

Deterministic boundaries

The five laws of GTM engineering.

author

Jason Derr

audience

GTM engineers, RevOps leads,
operators running probabilistic systems

read time

~20 minutes

What's inside

Five laws for revenue systems with a model somewhere in the middle. Each comes with the failure you will recognize, the mechanism that enforces it, and a public repo that runs it.

	The happy path is not the system	3
	Preface. The question that separates a demo from a system.	
	What changed	5
	Preface. Why GTM needs engineering discipline now and did not before.	
	Where to start	6
	Preface. The system in one picture, and the order that installs it.	
01.	The AI does not own the invariant	7
	Probabilistic systems need deterministic boundaries.	
02.	Green is earned	10
	Green must be earned by observation.	
03.	Unknown is a state	13
	Unknown is a state. Guessing is never the default.	
04.	Failures produce controls	16
	Failures should produce controls.	
05.	Autonomy is earned	19
	Autonomy is earned through evidence.	
	Same laws, upstream	22
	Marketing is not the exception. The five laws applied before the pipeline.	
A.	Sixteen principles, GTM edition	25
	The engineering vocabulary, translated.	

The happy path is not the system

The question that separates a demo from a system.

A lead appears. The workflow enriches it, scores it, writes the opener, sends it, and updates the CRM. Someone runs one lead through it in a meeting and the room agrees it works.

That meeting was a demo of the happy path. The system is everything the demo did not touch.

What happens when enrichment returns a company that changed its name last quarter? Or when two signals for one account arrive in the same minute?

The CRM holds four versions of that company and the workflow picks one. The model says it is confident and the evidence it cites does not support the claim.

The sending tool returns 200 and the message never leaves the queue. Someone edits the scoring definition on a Friday.

The workflow runs twice on the same event. The enrichment vendor is down for four seconds.

None of those are exotic. Every one of them happened inside a working revenue system this year, and every one of them produced a green result on the dashboard while it happened.

Making the workflow work was the demo's question, and the demo answered it.

Under what conditions are you willing to claim that it worked?

That question is the whole discipline. The rest of this guide is the five answers to it.

where the question comes from

Production engineering has asked it for decades. A service is never done because the demo ran.

It is done when its failure modes are named, instrumented, and each one has a defined behavior.

GTM inherited the tools of that world and skipped the posture. The workflow builder gets a mock lead through the sequence and ships. Nobody writes down the ways it could be lying.

By late summer this year, one revenue engine had four to six agents building on it at once.

The constraint had moved from how much got built to how much of what they reported could be believed.

An agent will report a green test suite against an endpoint that does not exist. It will call a migration safe on an empty table.

It will read "no disagreement" off two reviewers who never opened the same file.

Each of those is a green light earned from nothing. A person reading eighty commits a day cannot catch them by hand, and a dashboard cannot catch them at all.

what the five laws do

Each law answers the engineer's question for one part of the system. What may the model decide. What observation earns a success claim.

What happens when the observation is inconclusive. What a failure changes in the system. When accumulated evidence justifies less supervision.

Read them in order. They move from a single decision to a running system to a system that improves to a system that runs on its own.

What changed

Why GTM needs engineering discipline now and did not before.

Most of the workflow automation GTM ran on for a decade was deterministic at the orchestration layer. A trigger fired, a rule evaluated, an action happened.

The middle of that chain was knowable.

If the rule was wrong, it was wrong the same way every time, and the first bad record told you.

AI puts new steps in the middle. A trigger fires, a model interprets what it saw, a model judges what it means, a model generates the response, and then the action happens.

Those middle steps are probabilistic. A plausible answer can be wrong, and the same input can come back different on Tuesday.

A bad output does not repeat itself in a way a person would notice.

The operator practices built for the old chain assume the middle is knowable. Check a few records, read the output, ship it.

Those practices have no move when the middle can lie fluently.

Engineering built its practices for systems whose components fail, disagree, arrive late, repeat work, and return incomplete observations.

None of it was written for marketing, and none of it was needed there until the middle of the chain changed.

The architecture of GTM changed. The operating discipline has to change with it.

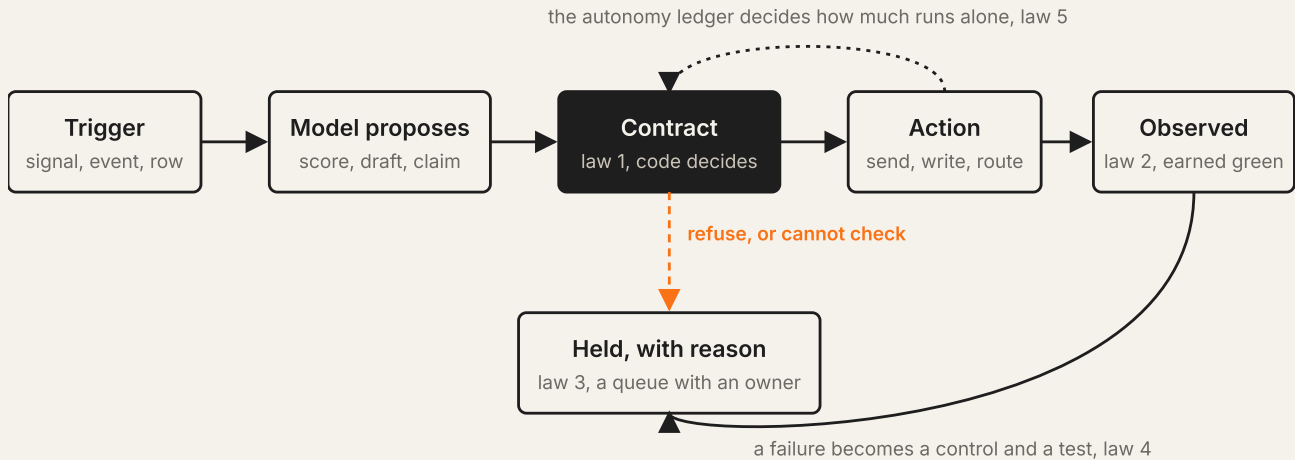
None of the five laws is new. Engineering wrote them for systems with unreliable components in them, decades before a model was one of the components.

AI engineering learned them again the moment a model entered the software loop, and sharpened two of them, evaluation and earned autonomy, into a discipline.

GTM is where they land next, for that reason. This guide wrote them from one production revenue engine and five public gates, and offers them as transferable, never as surveyed.

Where to start

The system in one picture, and the order that installs it.



Start at the riskiest boundary, which is usually the send. Work inward from there, one law at a time, and let each one run before the next goes in.

- First, the contract on the send. Six conditions, the model owns one, and a refusal that names what is missing. Law 1.
- Then earned green on the one dashboard number you already quote to leadership. Name the observation that turns it green, and refuse the state until that observation exists. Law 2.
- Then unknown as a state on enrichment. A third value, a queue with an owner and a clock, and nothing routing on a guess. Law 3.
- Then the postmortem loop. One four-line record per incident, and a test that fails if the control is removed. Law 4.
- Last, the autonomy ledger. Until it runs, no agent climbs past draft. Law 5.

Each step is a week of work for one engineer, and each one pays for itself the first time it refuses.

1

The AI does not own the invariant

Probabilistic systems need deterministic boundaries.

the failure you already recognize

An account enters your outbound sequence because the model said it fit. Nobody wrote down what the model was allowed to decide.

Your agents now produce more drafts, scores, and sends in a day than anyone on the team can read in a week. So the output goes through, and you hope.

The usual fix is more AI. A reviewer agent over the writer agent. A better model. A sharper prompt. The pile gets taller and more fluent.

A miss reads exactly like a hit, and no layer in that stack can say no and mean it. A reviewer agent can be argued with, and a prompt drifts.

the law

Probabilistic systems need deterministic boundaries.

Split the work in two. Models research, score, draft, classify, and judge. Deterministic code sits at each edge and decides whether the output proceeds.

Let the model answer whether this evidence suggests a hiring problem. Keep the decision to enter a sequence in code, where its conditions are written down and refused when unmet.

lineage

Engineers call this a contract, or an invariant, enforced at a policy point. What must always be true is written once, checked by code, and kept out of the model's hands.

Permission to act is a contract

the mechanism

The model proposes. Code verifies, gates, caps, routes, or refuses.

Every action with a cost, a send, a CRM write, a routed lead, gets a permission block written as plain conditions. Each condition is something the system can observe.

The block below lets an account into a sequence. The model contributes one line of it, the judge status. Five of the six conditions are facts the model never touches.

When a condition is unmet the block refuses and names what is missing. A warning beside an action that already happened has protected nothing.

Keep the two halves apart. A checker that reuses the model's own reasoning can only confirm what the model already decided, so its pass carries no information.

```
may_enter_sequence(account) =  
    qualifying_evidence ≥ 1  
    AND evidence_age_days ≤ 60  
    AND suppressed = false  
    AND identity_resolved = true  
    AND judge_status = pass  
    AND required_fields_present = true  
  
if any condition fails  
    refuse  
    name the failing condition  
    hold the account
```

evidence

- September 2026. Five boundaries ran on their own construction for a week. Before merge, the gates caught a rule that would have green-lit a fabricated fifty million in funding.
- The same week they caught a dashboard field emitting live markup past 241 green tests, and a benchmark judge whose prompt had been told the answer. Neither reached production.
- A redaction check that reused the redactor's own matcher was rebuilt after adversarial input proved it could not fail. A test client name had walked through every run.
- The rebuild uses two deliberately different matchers, and its suite holds every detector to one case where the redactor provably misses and the gate catches.

install

redaction-gate is the egress boundary. Public, MIT, zero dependencies, Node 20 or newer. Clone it and point the roster at your own client names.

```
git clone https://github.com/derrtaderr/redaction-gate.git
cd redaction-gate && npm test
```

Exit code 2 makes it a CI gate as is. Wrap your agent's tool table with `guard` and every exit it has, email, Slack, webhook, passes through one door.

In a no-code stack this law is a required-properties rule on the enrollment trigger, and a record that fails it lands in a review list instead of the sequence.

diagnostic

- Name one action in your stack that proceeds on a model's say-so with nothing deterministic between the two. What would a refusal look like there?
- For your send step, write the conditions that must be true before a send. How many does the system observe, and how many does it believe?
- Which check in your pipeline has never failed, and can you prove it is able to?
- If the model is wrong about fit tomorrow, what stops the account from being contacted?

2

Green is earned

A passing check proves only what it observed.

the failure you already recognize

Your dashboard says 400 sent. The sending tool accepted 400 requests, and that is the whole number.

Whether 400 messages left the provider, let alone reached an inbox, is a fact nobody collected.

The enrichment workflow finished with no errors. Half the accounts came back with the company field blank, and the scoring step downstream read every blank as a zero.

The API returned 200. The record in your CRM never changed, because the write went to a sandbox key that was still sitting in the environment file.

Each is a success claim the system made about itself, granted for free by a step that finished, an accepted request, or a log with nothing in it.

the law

Green must be earned by observation.

A state may turn green when something independent of the step observed the outcome the state names.

Every state earns exactly the word it claims, and the words are a ladder. Accepted, then sent, then delivered, then read, then worked.

The provider's record earns only the rung it describes.

Hold one question against every green light on your board. What did the system look at before it made this claim, and could that thing have said no?

lineage

Testing and observability settled this decades ago. A passing check proves exactly what it observed, and a check that could never fail was never a check.

The earned-green function

one function decides

Each state that carries weight gets one function. It names the observation that turns the state green, and returns unresolved when that is missing.

The function stays small. Sent reads the sender's own delivery record. Qualified reads the evidence rows the rule requires. Enriched checks the fields the next step will read.

Published fetches the artifact from where it was meant to land. Working divides an outcome by a denominator written down before the run.

the five false greens

Green lights that lie do so in five ways. Emptiness, where the check ran on nothing. Disagreement, where two reviewers answered differently and the workflow read one.

Missing state, where the field the check needed was never populated. Failed observation, where the check itself errored and the error was swallowed.

The fifth is the check that has never returned red. Until you make it fail on purpose, you have no evidence it can.

Write the five into the function as refusals, and the board stops inheriting green from silence.

```
earnedGreen(state, evidence)

  sent      needs  the provider's send record
  delivered needs  the provider's delivery event
  qualified  needs  evidence rows the rule names
  enriched  needs  fields the next step reads
  published  needs  artifact fetched at destination
  working    needs  outcome and denominator, both recorded

  refuse when
    evidence is empty           → unresolved
    reviewers disagree         → unresolved
    required field missing     → unresolved
    observation itself errored → unresolved
    check has never failed once → untested

  return green only on the named observation
```

evidence

- 2026-09-20. On one production lane the independent review blocked twelve times on a single error class before the merge, and every block cited the file.
- 2026-09-12. The same review caught a dormant bug that would have charged a hundred times the intended amount. The lane that armed it had a green suite.
- 2026-09-11. Paging tests had passed against an API that did not exist, so nothing in the suite could fail. The independent review blocked the merge and named the file.

install

ship-check is public at github.com/derrtaderr/ship-check, a reviewer contract plus a mechanical gate.

A reviewer that did not build the change walks it as a first-time user against the flow you promised, and returns BLOCK or BLESS with evidence quoted.

Point it at the one workflow your team trusts most. Write the promise page first, one page on what a first-time user should be able to do.

Hand the contract to a reviewer with no shared context. Only an explicit pass proceeds, and anything hedged parks for a human.

In a no-code stack this law is a property that reads the provider's delivery event, never the workflow's completion step.

diagnostic

- Pick the greenest state on your board. What observation made it green, and who collected it?
- Which of your checks has never once returned red? What would it take to make it fail on purpose?
- When two reviewers or two runs disagree, what does the workflow do with the second answer?

3

Unknown is a state

An answer the system could not check is a value of its own.

the failure you already recognize

Your enrichment call comes back without the field. The workflow needs a value, so it takes the best guess and moves on. The guess routes the account.

A job posting has been open 192 days. Nobody has looked at it since the day it was collected. It ranks first every week because nothing in the stack asks whether it is still true.

Your dashboard has two colors. A check that could not run shows the same green as a check that ran and passed.

Each of those is the same move. The system met something it did not know and treated it as something it did.

That move is where the guess gets in, as a default, as a rank, or as a pass.

the law

| Unknown is a state.

Give it a name, store it as a value, and let it fail closed. A row the system could not verify may be held and reviewed. It may never route, rank, or send.

lineage

Security engineering settled this long ago under the name fail closed. When the lock cannot read the badge, the door stays shut.

GTM stacks mostly fail open, because open keeps the week moving.

Three states, and how they combine

Every claim the system holds gets one of three states. Matched means evidence agrees.

Contradicted means it disagrees. Unresolved means nobody looked, or the check could not run.

The third state is the whole design. Somebody looked and found nothing is evidence, and what it means depends on the claim.

For an open role it contradicts. For a suppression list it matches.

Nobody looked is the absence of evidence, and it never shares a value with either.

Unresolved is a first-class value with rights and limits. It can sit in a review queue and print why it could not be checked. It cannot rank, route, or enter a send batch.

Where several checks cover one claim, a contradiction outranks everything and an unresolved check outranks a match.

Two clean checks and one that could not run is an unresolved claim.

The design holds rows out instead of dropping them, because a gate that stops the week gets deleted by Friday. A queue that names what nobody has looked at gets worked.

```
evaluate(claim, evidence)
  evidence could not be checked    → UNRESOLVED
  evidence supports the claim      → MATCHED
  evidence contradicts the claim   → CONTRADICTED

combine(checks)
  any CONTRADICTED                → CONTRADICTED
  else any UNRESOLVED             → UNRESOLVED
  else                            → MATCHED

route(verdict)
  UNRESOLVED    → hold, print why
  CONTRADICTED → exclude
  MATCHED      → eligible to rank
```

evidence

- 2026-09-17. An evidence read over ninety scored leads returned zero with coverage. It traced upstream to 133 of 163 accounts held behind a threshold written for a retired customer definition. The gate was re-pointed and the accounts released that week.
- 2026-08-30. The top account on an outbound list ranked on a job posting open 192 days, for a seat filled months before. A person caught it, and the detector below shipped eleven days later.
- 2026-09-10. The detector's first run over 257 signals retired three on age and held 254 as unresolved until each had a source to check against. None entered a send batch on a guess.

the week still moves

A gate that stops the week gets deleted by Friday, so the queue needs an owner and a clock. Every held row names the check it needs, its owner, and a due time.

When the detector held 254 of 257 signals, the batch ran on the ones that resolved and the queue drained as each signal got a source. The pipe kept running, smaller and verified.

install

signal-desk is public at github.com/derrtaderr/signal-desk. Eight stages from capture to outreach, every one fails closed. An uncheckable claim is held with its reason printed.

Point it at the signals your ranking already uses. Give each one the fact that has to still be true, a source to check, and a last-checked date.

In a no-code stack this law is a third value on the property, and a view filtered to it is the queue somebody works.

diagnostic

- Which field in your routing takes a default when enrichment returns nothing, and what does that default decide?
- Pick your top five ranked accounts. When was the signal under each one last checked against anything?
- How many rows sit in your unresolved queue today? If zero, is that because everything was checked, or because nothing can land there?

4

Failures produce controls

Fix the incident once. Fix the class permanently.

the failure you already recognize

An outreach message invents a claim about the account, so someone edits the prompt. A duplicate record gets contacted twice, so someone suppresses the second row.

A top-ranked account turns out to be a job posting filled months ago, so someone removes it from the list.

Each fix takes ten minutes and each one works. A month later the same shape is back with a different name in it.

Your team is now spending its week on instances, and the system has learned nothing from any of them.

A patch changes one record. The failure lived in the system, in a threshold nobody remembered writing, in a signal with no expiry, in a channel nothing was reading.

the law

Failures should produce controls.

Every production failure either strengthens an invariant your system already enforces or creates a new one. Whether the record got fixed is the least interesting question.

lineage

Engineering has run this loop for decades, under the names blameless postmortem and regression suite. The incident is the input. The output is a rule the system holds from then on.

The loop, in four steps

the mechanism

The loop turns an instance into a control. It runs the same way for a wrong claim in an email as for a wrong bucket in a scorer.

Name the instance exactly, with the row, the timestamp and the observation that caught it. Then ask what class of failure this row is one member of.

The class is the real finding. A held account behind a stale threshold is one member of "a gate implements a decision nobody re-reads."

Write the invariant that would make the class impossible, and put it where the system enforces it rather than in a document.

Then write the test that fails if the invariant is ever removed. Without it the control is a memory.

Check the control against the class, too. Naming the decision a threshold serves makes a stale gate findable.

Reading the threshold from a versioned policy object, and failing the build when a gate names a retired version, makes it impossible. The second is what the law asks for.

The record below is the whole artifact. Four lines per failure, kept where the next person will find it.

```
# failure record, one per incident
instance  133 of 163 accounts held on
          "fit below threshold" (09-17)
class     a gate implements a decision
          nobody re-reads after it changes
control   every threshold names the
          decision it serves, in code,
          so grep finds each gate to move
test      fails if a threshold exists
          with no decision name attached
---
instance  192-day posting ranked first
          (08-30, caught by a reader)
class     recency mistaken for validity
control   age ceiling plus evidence check,
          fails closed on SUSPECT
test      a signal past ceiling with no
          evidence cannot enter a batch
```

evidence

- September 2026. A customer definition changed and the enrichment gate kept the old one. 133 of 163 accounts held, 130 on a retired reason. The control names the decision every threshold serves.
- August 2026. The top outbound account carried a job posting open 192 days. The control gives every signal an age and fails closed. First run held 254 of 257 signals out of send batches.
- September 2026. A channel had run three weeks with no written question. The control is a register with a gate date, and the next run paused itself under its cap.

install

The same loop, applied to inbound events, produced webhook-engine. Four controls, each born from a failure class the ten-minute receiver still carries.

A signature over the raw bytes replaces trusting the sender. An idempotency reserve replaces hoping the network never delivers twice.

A bounded retry replaces giving up on the first error. A replayable dead-letter queue replaces losing the event in a log.

Point it at the endpoint your CRM sync calls. github.com/derrtaderr/webhook-engine.

In a no-code stack the record is a note on the workflow, and the test is a scheduled check that fails loudly when the rule is missing.

diagnostic

- Take your last three incidents. Can you name the class each belongs to in a sentence that never mentions the record?
- For each class, where does the system enforce the rule today? If the answer is a document, it is a patch.
- Which of your invariants has a test that would fail if someone removed it on a Friday afternoon?
- If the same event reached your pipeline twice in one second, what would happen, and have you ever watched it happen?

5

Autonomy is earned

Accumulated evidence, and only that, justifies less supervision.

the failure you already recognize

Your agent nailed fifty drafts in a row, so you stopped reading them. Around draft thirty the review became a skim, then a stamp, then a Friday decision to flip it to auto.

Nobody wrote that decision down. Nothing in the stack can take it back. The agent that aced the demo now writes CRM rows at 2am, weeks after anyone last looked.

That is how most unattended agents got their clearance. A feeling about the agent turned into a setting nobody owns.

the law

Autonomy is earned through evidence.

The question under every unattended agent has two halves. What evidence would justify letting it act without supervision, and what evidence takes that permission back.

Nothing in the stack owns that decision today. Eval tools score outputs. Dashboards chart them. A human still converts a score into permission, informally, once, and it sticks.

lineage

Production engineering calls this progressive delivery. A change runs in shadow, then on a canary slice, then everywhere, and each step is promoted on observed behavior.

A failed step rolls back. The same ladder fits an agent.

The ladder

how permission moves

Six rungs. Observe, recommend, draft, act with approval, act within limits, act alone. An agent starts at the bottom and climbs one rung per clean stretch.

The evidence is a streak with coverage. Every check writes a row that names the case class it exercised.

A set number of consecutive passes under the same rules, spread across the named classes, earns the next rung.

A streak measures consistency. Coverage says what that consistency applies to. Fifty easy cases in a row prove nothing about the hard class the agent never saw.

One block resets the streak to zero.

A rule change resets it too. Yesterday's clearance was earned under yesterday's rules, and a streak that survives a rule change measures nothing.

Demotion is automatic. A failed run at any rung drops the agent one rung and restarts the count.

The ladder is a ledger, and the kit's CI enforces the reset. If you will not run the ledger, keep the agent at draft.

Three demos and a good feeling never move an agent up. The logged streak is the only thing that does.

rung	evidence required to hold it
1 observe	runs, writes nothing, rows logged
2 recommend	N passes at rung 1, human reads all
3 draft	N passes at rung 2, human edits
4 act w/ approval	N passes at rung 3, human clicks
5 act within limits	N passes at rung 4, volume capped
6 act alone	N passes at rung 5, floor rules on
promotion	N consecutive passes, same rules, every named case class exercised
hold	any high-risk class still unresolved
reset	one block, or any rule change
demotion	one failed run drops one rung

evidence

- Production eval framework, 2026. Rule checks, then per-axis LLM judgments, each axis pass or fail. Five consecutive passing runs before any agent runs alone. Bad output fell 80% across the fleet.
- The same rule extended to the builders, 2026-08 to 09. Four to six coding agents in parallel, each earning a merge through an independent check.
- The packaged gate blocked its own release twice, 2026-08. Sixty-two tests were green while a documented safety guarantee was false.

install

gtm-agent-evals is the kit, MIT, on github.com/derrtaderr. Deterministic rules run first and cannot be overridden. A fail-closed LLM rubric scores the rest.

An autonomy gate turns the streak into a yes or a no.

Golden-trajectory regression catches an agent drifting from a known-good run. A CI gate blocks the merge that would change it silently.

Point it at whichever agent runs unattended today. Wire the exit code into the one branch that decides whether the action fires.

In a no-code stack the ladder is a property on the agent's config plus a counter that resets. If nothing resets it, the agent stays at draft.

diagnostic

- Which of your agents acts unattended right now, and what row of evidence earned that?
- When a rule changed last, did the agent's clearance reset, or did it carry over?
- What single event would demote an agent today? If you cannot name one, nothing can.
- How many clean runs does your team require before flipping to auto, and is that number written down?

upstream

Marketing is not the exception

The five laws applied before the pipeline.

the failure you already recognize

On 2026-09-08 a content pipeline passed ten drafts in a row through its own rubric. Every one came back clean.

Read side by side, the ten had converged on one skeleton, and a reader could infer each takeaway from the first two lines.

The grader had been reading the writer's own moves and calling them good. Nothing in the loop could see the sameness, because nothing in the loop stood outside it.

That is the default shape of an AI content stack. A model researches the segment, turns it into messaging, writes the campaign, grades it, and reports how it performed.

Every step passes. The same model family validates its own assumptions all the way down, and the report at the end reads as a result.

the principle

The five laws govern every probabilistic decision in the motion, upstream of agents too.

Research, positioning, copy, and creative are model outputs now, and they need the same boundaries as a routing decision.

The judgment layer here decides four things. What is true. What is good enough. What ships. What the response teaches you next.

lineage

Independent measurement. The generation step never supplies the evidence for its own grade, and the grade never stands in for the outcome.

Marketing is not the exception

keep four things apart

The writer produces the draft. A blind judge grades it against a rubric it did not write, with no memory of the draft's history.

The audience responds, or does not. A calibration pass compares what the judge predicted with what the audience did.

Four parties, and the boundary between each one is deterministic. The judge cannot see the writer's reasoning. The calibration pass reads the judge's verdicts as data.

the judge is an instrument

An LLM judge is itself a probabilistic component, so it gets the same treatment as the writer. Measure it before trusting it.

On one draft, nine runs of the same judge on identical input returned four clean verdicts and five kills.

The confidence scores ranged from 0.64 to 0.76 with no relationship to the verdict.

A single verdict was one sample. Now the judge runs three times. Three of a kind is a verdict.

A two-one split is a state of its own and goes to a person, and every past verdict got re-read as one draw.

loop, with the boundaries marked

```
writer      → draft
              | boundary: judge never sees writer reasoning
judge x3    → 3-0 is a verdict, 2-1 is a split held for a person
              | boundary: verdict is data, never the outcome
audience   → response (48h read)
              | boundary: recognition, never impressions alone
calibrate   → judge predicted vs audience did
              | human relabels a blind set, reasons first
              | dimensions are clustered from reasons after
rubric      → versioned, changed only on a replay set
```

upstream evidence + practice

evidence

- 2026-09-08. Ten drafts passed the rubric clean while converging on one skeleton whose takeaways a reader could infer from the first two lines. The rubric gained an inference test the same day.
- 2026-09-09. Nine judge runs on identical input, four clean. Three runs became the rule, a split became a recorded state, and every prior verdict was reclassified as one sample.
- 2026-09-25. Twenty-five post bodies, order randomized, every verdict stripped, labeled by a human with a one-sentence reason each. The categories got clustered out of the reasons afterward. Deciding the dimensions first had already failed on replay.

the same laws, named

- Boundary. The judge grades, and code decides what counts as a verdict.
- Green. A clean verdict is one draw until the run count says otherwise.
- Unknown. A split verdict is a state, and it routes to a human rather than to publish.
- Controls. Ten convergent drafts became an inference test, never ten edits.
- Autonomy. The unattended run was earned on weeks of judged kills, and kills are the test output.

diagnostic

- Which model wrote your last campaign, and which model told you it was good?
- If you ran your content grader three times on the same draft, would it agree with itself?
- When a draft passes, what did the audience do, and did anyone compare the two?
- What is the state of a draft the grader cannot decide on, and who sees it?

Sixteen principles, GTM edition

The engineering vocabulary, translated. Each line is what the principle means when the system is a revenue motion.

- **Contracts and invariants.** An account cannot route unless the evidence exists. A send cannot occur without approval, source, and provenance.
- **Fail closed.** Uncertainty never becomes success. Matched, contradicted, or unresolved, and unknown is never green.
- **Observability.** You can see which signal fired, what evidence was used, which model decided, what action ran, and what landed.
- **Testing.** Fixtures, golden sets, regression cases, adversarial cases, and false-green tests, before production.
- **Evaluation.** Rubrics with calibrated thresholds, repeated runs, disagreements surfaced rather than averaged.
- **Deterministic boundaries.** The model proposes. Code verifies, gates, caps, routes, or refuses.
- **Idempotency.** The same event can replay without a second email or a duplicate record.
- **State management.** Explicit statuses, receipts, timestamps, ownership, and lifecycle, in one place.
- **Provenance.** Source, timestamp, evidence, and transformation history travel with the claim. A CRM field is a claim.
- **Separation of concerns.** Detection, judgment, generation, approval, and execution are separate stages with separate owners.
- **Versioning.** Prompts, schemas, scoring logic, the customer definition, and eval sets carry versions. A result names the version that produced it.
- **Replayability.** The same historical event runs against the old system and the new one, and the outcomes get compared.
- **Graceful degradation.** When enrichment dies, the account is held. The missing field is never invented.
- **Operational metrics.** Accuracy, unresolved rate, false-green rate, latency, cost per action, duplicate rate, intervention rate. Volume alone tells you none of these.
- **Change control.** Stage, test, shadow, canary, production, with a rollback that has been exercised.
- **Postmortems.** The question after a failure is which control would have prevented the whole class.

The judgment layer

Everyone eventually gets the same models. The edge is the judgment you construct, not the capability you buy.

Each law in this guide is an engineering commonplace. Contracts, earned green, fail-closed defaults, postmortem loops, progressive delivery.

The contribution is the translation into a revenue motion and the evidence that it holds there.

The five repos are the installable half. gtm-agent-evals, ship-check, redaction-gate, webhook-engine, and signal-desk are public at github.com/derrtaderr.

Point each one at your own system.

The weekly writing is at derr.ai. It is where the next law gets tested before it earns a chapter.

receipts

Every number in this guide traces to a dated post, a public repo, or a recorded run. The list, with links, sits on the guide's page at derr.ai.