The Build Discipline That Ships

RIG Operating Procedures — the packet chain, the energy math, and the proof gates that move AI work from a demo to production.

Mike Rodgers, Rodgers Intelligence Group Field doctrine, v1 ~3,400 words · 15-min read

Most enterprise AI never ships. The number that gets quoted — roughly ninety-five percent of generative-AI pilots return nothing measurable — isn't a model problem. The models are fine. The pilots die in the gap between "it worked in the demo" and "I can re-run it tomorrow and bet a customer on the result." That gap is process, not intelligence. A model that scaffolds an app, writes a migration, or answers a support ticket will do it differently every time you ask. You cannot reproduce it, you cannot prove it, and you cannot hand it to someone who will be on the hook when it breaks.

RIG Operating Procedures is the build discipline I use to close that gap. It is not a framework you bolt on top of a model and hope. It is a chain of typed artifacts, statistical gates, and a single hard rule: **nothing is done until a real command's output says so.** This paper walks the whole machine — the spine, the chain, the math, the approval gate, and the proof object — with the actual numbers and the failures that earned every rule. If you build with AI and you've watched a "10/10 passing" run turn out to be eight stubs, this is the antidote.

> **Code owns decisions. LLMs assist transformation. Gates decide if it ships. ProofPacket or it did not happen.**

Those four lines are the operating law. Everything below is how they're enforced — mechanically, on the real tree, with the gate flipping red when you break what it guards. No vibes. No theater. Let's build.

PART I · THE SPINE

IQRSQPI: confidence is the gate, not the calendar

Every mission runs the same eight-step spine. The acronym is ugly and I keep it that way because each letter is a checkpoint you are not allowed to skip:

THE IQRSQPI SPINE — RUN ONCE PER STEP, RECURSIVELY PER LAYER

Step	Name	Done when
I	**Intent**	Delete it and the thing has no reason to exist. One sentence.
Q	**Question**	The unknowns are surfaced from a world-class persona panel — not guessed.
R	**Research**	Every load-bearing question has an answer, a source, and a grade (HIGH/MED/LOW).
S	**Solution / Spec**	Every line resolved on paper and tagged AUTO / DRAFT / FLAG; deterministic-vs-probabilistic boundary declared.
Q	**Quality**	The spec is interrogated again — the residual-question list is written, nothing DRAFT treated as settled.
(R)	**Research**	The Q⇄R loop runs **at least three times** until a full pass surfaces no new load-bearing unknown.
P	**Proof / Plan**	The plan is locked, dependency-ordered, every chunk independently verifiable by command.
I	**Integrate**	The slice ships, passes its proof gates, and seals into a ProofPacket.

The heart of it is the **Q⇄R loop**. You ask, you research, you ask harder, you research again — and you do not advance until you are "very, very comfortable you have good answers." Two research rounds is the minimum; you loop until the loop is *dry*, meaning a full pass turns up no new unknown. Confidence is the gate. Not the deadline, not the token budget, not how tired you are.

Two things keep the loop honest. The first is that it's **prototype-led**: instead of arguing about an abstract question, you build a throwaway mock, interrogate *that*, and research how the best tools actually solve it. A mock answers experiential questions a survey can't — "what would this even feel like?" — and it's cheaper to throw away than the spec it de-risks. The second is the **red team**: every load-bearing answer gets an adversarial pass that tries to *refute* it before it's accepted. If independent skeptics can break a claim, it's downgraded — never the reverse.

And the whole thing is fractal. You run IQRSQPI once for the macro mission, then again for each step, then again for each layer inside that step. Go as many layers deep as the intent requires, then do it again, every time.

PART II · THE CHAIN

The packet chain: no packet skipped, no fact invented

The spine produces artifacts, and those artifacts form a chain. Each is a typed JSON object. Each is a hard block: if it's missing, the build stops. The root law:

```
Intent → Question Ledger → Research → Spec → UX Workflow
   → Plan → Test Plan → EVOLVE Route → ProofPacket → DoneContract
```

Every mission advances left-to-right. A later packet may never invent a fact an earlier packet didn't establish.

That last clause is the one people underestimate. **A later packet is not allowed to invent facts the earlier packets didn't contain.** The Plan can't introduce a requirement the Spec never carried; the Spec can't claim a fact the Research never sourced. The chain is the audit trail. If a fact appears in the ProofPacket, you can walk it back through every link to where it was first established and graded.

THE HARD BLOCKS — EACH MISSING PACKET STOPS THE BUILD

Packet	Blocks until
`IntentPacket`	present, or no mission exists
`QuestionLedger`	three grill-loop rounds recorded; every item answered or explicitly escalated
`ResearchPacket`	every load-bearing claim has a source and a confidence grade
`SpecPacket`	every acceptance criterion is observable — no forced guesses
`UXWorkflowPacket`	every screen traces to a state transition (no orphan screens)
`PlanPacket`	every chunk is independently verifiable by command
`TestPlanPacket`	every acceptance criterion has at least one check
`EvolveRoutePacket`	exactly one risk profile selected (B1–B4 / hotfix)
`ProofPacket`	command output + artifact hashes exist; the hash chain self-verifies
`DoneContract`	"done" is computed from the evidence — never asserted by the agent

The Triple-QR grill loop

The Question Ledger has its own minimum: **three Q/R cycles**, in order.

1. **First pass** — expose the missing facts and the buried assumptions.
2. **Second pass** — challenge the *strongest* answer, not the weakest. Steel-man the opposition.
3. **Third pass** — close, escalate, or block every remaining open item.

An open question is permitted only if it is *explicitly escalated*. Silent ambiguity is a failed gate. This is the difference between a spec that looks complete and one that is.

PART III · THE MATH

The energy gate: honest sigma, or no sigma at all

"Energy" is the residual risk in a mission — every unanswered question, uncited claim, orphan screen, skipped gate, and unapproved outward action counts as positive energy. A mission advances only when residual energy is below the gate for its stage. Most of the gates are boolean (is the claim sourced? does the screen trace?). The interesting one is statistical, and it has a history worth telling.

The first version scored spec counts with a robust Z-score (MAD-Z) and talked about "30-sigma" outliers. A consensus pass across ~38 papers found that math was **statistically invalid** for our inputs — page and integration counts are skewed, zero-inflated, and small-n. MAD-Z and sigma bands assume something our data isn't. So we rebuilt it. **Calibration v2** is the honest version:

```
# Counts are scored by UPPER-TAIL PROBABILITY under a fitted model.
# Poisson by default; switch to Negative-Binomial when overdispersed (var

  Poisson:        P(X ≥ k)  via exact complementary-CDF sum
  Negative-Binom: r = mean² / (var − mean),   p = r / (r + mean)

# Two HONEST hard cutoffs per feature — policy values, NOT sigma bands:
  over-max   :  observed > 3.0 × baseline_max         → BLOCK
  surprising :  tail-prob   < 0.01 (fitted model)    → BLOCK

# Aggregate = MAX-SCORE: any single feature that blocks fails the gate.
# A feature that is constant across the baseline is DROPPED, not epsilon-f
```

The Calibration-v2 count gate (`k_factor = 3.0`, `p_floor = 0.01`). Every number replays from a SHA-locked frozen baseline — no scipy, no wall-clock, no LLM.

The point of the rewrite isn't the distribution choice. It's the **relabeling**. We stopped calling policy cutoffs "sigma." A "tail probability below one percent" is a real probabilistic statement. A "30-sigma deviation" on skewed count data is theater dressed up as rigor. The robust scale we kept (Qn, with Akinshin's finite-sample correction, ~82% efficiency versus MAD's ~37% at n≈22) is used only as a cross-check, never as the primary signal. If you can't derive a number from first principles, you don't get to wear it.

There's a parallel **plan-energy** gate that scores chunk risk so a fat, un-splittable plan chunk gets caught before it's built:

```
chunk_energy = 1·layers + 0.5·files + 3·contract_edits
             + 4·(no tests) + 2·(no rollback)

plan_energy  = max(chunk_energy)        # worst chunk drives the verdict
gate FAILs (must split) if plan_energy > 6.0
```

E_plan — weights are heuristic and honestly labeled as such. A chunk that edits a contract with no tests and no rollback (3 + 4 + 2 = 9) blocks on its own.

PART IV · THE ROUTING

EVOLVE profiles and BMS build modes

Not every change deserves the same ceremony. A typo fix and a payments migration should not run the same gate-set — one is over-process, the other is under-process, and both are how you get hurt. Two routers decide the rigor.

EVOLVE: how risky is this change?

The senior-engineer instinct "this is going to break" made computable. Build energy is a weighted blend, scored 0–100:

```
E_build = 0.25·test_risk        + 0.25·blast_radius
        + 0.20·complexity_delta + 0.15·spec_drift
        + 0.10·rollback_difficulty + 0.05·review_history_risk
```

Then the route resolves in strict order. A **hard ceiling** (irreversible migration without a backup, a secret in the commit, an untested path to prod) is infinite energy — it *blocks before execution*, full stop. **Hard triggers** (schema migration, auth, payments, infra, public-API break) force B4 regardless of the score. A production incident takes the compressed hotfix lane. Brownfield (touches existing contracts) raises the floor to B3 — it is never trivial. Otherwise you band the score:

EVOLVE PROFILES — THE GATE-SET SCALES WITH THE BLAST RADIUS

Profile	Band / trigger	What it requires to seal
B1	E ≤ 20, greenfield	spec executable, tests green
B2	E ≤ 45, additive	+ grill ledger empty, interface review, coverage floor, feature flag, no secrets
B3	E ≤ 70, or brownfield	+ blast-radius map, contract tests, regression replay, rollback tested
B4	E > 70, or hard trigger	+ expand/contract migration, rollback rehearsal, change-advisory gate, **human approval (Gate-D)**
hotfix	production incident	compressed grill + rollback tested, then mandatory post-hoc full grill + retro

BMS: how much of this can deterministic code own?

The other axis is the Build Mode Score — a confidence number that decides how much of the work an LLM is allowed to touch versus how much is owned by deterministic code. High confidence means code owns it; low confidence means you're in agent territory and the guardrails tighten.

A1–A4 BUILD MODES — ROUTED BY BMS CONFIDENCE

Mode	Name	BMS
A1	Python only — deterministic	≥ 0.75
A2	Hybrid — code core, narrow AI slots	0.45 – 0.74
A3	Agent bounded	0.25 – 0.44
A4	LLM agent, free	< 0.25

The fallback rules matter as much as the thresholds: an A1 task that hits an UNKNOWN drops to A3.1. An A2 whose confidence collapses falls to A3. An A3 with unresolved ambiguity escalates to A4. And any A3 or A4 path that takes a public action requires human approval. The system degrades *toward* caution, never away from it.

> **Deterministic before agentic. The generator is the moat; the AI fills only narrow, named slots.**

This is the principle under both routers. The generator core is an **AI-free zone** — same input, byte-identical output — even when the product it emits runs an LLM at runtime. Compile-time AI, never AI in the control or verification path. Structure, schema, migrations, CI: 100% deterministic templates. The probabilistic generation is confined to small, pre-validated business-logic holes — never structure, never authz, never the gate that decides if you ship.

PART V · THE BRAKE

Gate-D: nothing goes outward without a typed human "yes"

Every irreversible action — deploy, git push, production migration, external API write, customer email, payment, destructive delete, credential movement — is a Gate-D action. The rule is absolute: **no outward action without explicit, typed human approval.** The approval is an exact phrase,

`APPROVE <run_id>`, bound to the mission. There is no `--yes`, no `--force`, no env var, no non-interactive bypass.

The mechanism is the *generate / operate split*. Outward artifacts — deploy workflows, infrastructure-as-code — ship **dormant**: `workflow_dispatch` only, behind a fail-closed `if armed` gate. The only path from dormant to live is a separate **arm** step that requires the typed phrase. The generator can emit a complete deploy bundle and arm nothing — and we prove zero outbound with a socket trap across the whole generate→arm path before trusting it.

> THE BUG THIS RULE EARNED
>
> **The whitespace-padded approval**
>
> An adversarial pass caught an `arm` verb that accepted an approval phrase with a trailing space — it *would* have armed a deploy on a near-miss. Fixed to exact-match, no leniency. This is the rule that keeps a generator from becoming a foot-gun.

And one corollary that sounds obvious until you watch it fail: **a skip is never a pass.** A gate that can't run — no Docker, no load-test tool, no browser for visual capture — returns a distinct non-pass status that *blocks* "done." It never silently counts green. An honest "I couldn't run this" is infinitely more valuable than a fake "it passed."

PART VI · THE RECEIPT

The ProofPacket: done is computed, not claimed

This is the object the whole machine exists to produce. The ProofPacket is not a report — a report is a hypothesis. The ProofPacket is the **evidence object**: hash-bound to the actual bytes on disk, self-verifying, and the sole place "done" is decided.

```
ProofPacket {
  schema_version    : "2.0.0"
  artifact { // the bytes, not a number passed in
    artifact_sha256 : <canonical tree hash of the real output>
    file_count, load_bearing_present
  }
  reproducibility {
    tier1_byte      : { expected, replay, match }   // byte-identical
    tier2_ai_slots  : { slots, all_cache_hits, all_properties_passed }
  }
  gates : [ {
    gate_id, status, blocking, exit_code,
    bound_to        : <artifact_sha256 this gate verified>,   // MANDATORY
    evidence_sha256 : <hash of captured stdout/stderr>,
    tool, started_at, duration_ms
  } ... ]
  dod : {
    // done = ALL blocking gates passed AND each bound_to matches
    //        the artifact hash AND tier-1 byte-match holds.
    done : <pure boolean AND — never an agent's opinion>
    first_failure, blocking_gates_passed / total
  }
  economics, supply_chain,
  packet_sha256     : <computed LAST over the packet minus this field>
}
```

ProofPacket v2 — written to `.rig/proofpacket.json`. The packet hash makes it tamper-evident; recompute it and it has to match.

Four laws hold this together. **Verify the artifact, not the report** — when two agents disagree on the test count (and under concurrency they will), the working tree and the command output are the only truth. **A green must be reproducible by command** — if you can't paste the command and show its exit code, it isn't proven. **"Done" is computed** — `done = all(blocking_gates_pass)`, a hard AND, one red gate means not done. **Hash-bind the proof** — a gate result carries `bound_to`, the exact artifact hash it ran against; a claim floating free of the bytes it describes is not proof.

Reproducibility itself is two honest tiers, and you never claim a tier you can't prove. **Tier 1** is byte-identical — the deterministic frontend, hash-checked. **Tier 2** is lockfile-pinned and version-stamped — the backend and AI slots, never claimed byte-identical. Calling Tier 2 "byte-identical" is a lie the determinism gate must reject. Exact pins only, no `^`, no `~`, no `latest`; base images by real `@sha256` digest; secrets by reference, never inline.

PART VII · THE WALKTHROUGH

One mission, start to seal

Here's the whole machine on a single brownfield change: "add a billing page to an existing app and wire it to Stripe." Watch where it stops you.

1. **Intent.** One sentence: "Authenticated users can view and pay an invoice." Delete it and the feature has no reason to exist — it survives the test.
2. **Question.** The persona panel surfaces the unknowns: how does authz scope the invoice? what's the idempotency story on a double-submit? what happens on a failed charge? Three grill rounds. The idempotency question gets *escalated*, not guessed.
3. **Research.** Stripe's idempotency-key pattern, sourced and graded HIGH. Existing auth middleware, read from the real tree, graded HIGH. Nothing invented.
4. **Spec.** Acceptance criteria, all observable. The det/prob boundary is declared: the schema, the route, the Stripe client wiring are deterministic; nothing here is an AI slot. Tagged AUTO.
5. **Quality ⇄ Research.** The spec is interrogated. "What proves the failed-charge path?" becomes a new test requirement. Loop until dry.
6. **EVOLVE route.** The router sees `payments = true` → hard trigger → **B4**, no matter the energy score. The gate-set now demands contract tests, regression replay, rollback rehearsal, and a human approval gate.
7. **Plan.** Dependency-ordered: schema → Stripe wiring → route → page → tests. E_plan scores each chunk; the "Stripe wiring + migration" chunk is split so no single chunk edits a contract with no rollback.
8. **Implement.** Each phase: build → adversarial-verify (plant the failure, watch it go red, restore) → verify the real tree → commit on green. The decisions log records every fork with its reversibility.
9. **Proof.** Gates run on the real tree. The ProofPacket computes `done` as a hard AND. The deploy bundle is generated **dormant**.
10. **Gate-D.** Because it's B4 and it deploys, it stops. It waits for me to type `APPROVE <run_id>`. Until then, zero outbound — proven by the socket trap.

Ten steps, and the machine refused to ship three times: at the escalated question, at the forced B4 route, and at the dormant deploy. That refusal *is* the product. A demo that ships in one step and breaks in production is worth less than a build that stops you twice and ships something you can bet on.

What broke, and the rule it earned

None of this is theoretical. Every gate above exists because something passed when it shouldn't have. The honest version of a methodology is the list of its own failures.

LESSON 01 · THE PHANTOM GREEN

"10/10 regression" was eight constant-returning stubs

A regression suite reported a perfect pass. It was eight functions that returned a constant and counted BLOCKED as PASS. Nobody caught it by reading the summary.

Fix: Adversarial verification — every gate ships a planted-failure test proving it can flip red. A green that won't go red under a planted failure is theater. The planted failure stays as a permanent regression case so the gate can never rot back to vacuous.

LESSON 02 · THE EMPTY-TREE PASS

The determinism gate read green on a generator emitting zero files

Same input, same output hash — except the output was nothing. An empty tree hashes consistently, so the gate was "passing" on a generator that produced no app at all.

Fix: Verify against ground truth empirically — boot the app, confirm the load-bearing files exist, don't grade by inspection. The ProofPacket records `file_count` and `load_bearing_present` for exactly this.

LESSON 03 · THE SALTED HASH

"Byte-identical" output was secretly salted by UUID + wall-clock

The `output_hash` claimed byte-reproducibility. A double-build exposed it: a UUID and a timestamp were salting every run. It was never reproducible; it just looked it.

Fix: Stamp, don't timestamp — use a content-derived spec hash, never wall-clock. Exclude the proof sidecar from the determinism hash. Prove Tier-1 with an actual double-build, not an assertion.

LESSON 04 · THE CONCURRENT-WRITE DRIFT

Subagents reported 20, then 21, then 22 tests — all "passing"

Parallel agents wrote to the tree concurrently and each reported a different count from its own stale view. Only re-running the regression on the real tree revealed the truth.

Fix: Reconcile before commit — re-read the working tree, resolve stray files, run the gate yourself, then commit. Reports drift under concurrency; the tree and the exit code don't.

LESSON 05 · THE CATHEDRAL TEMPTATION

We almost built 588 lattice cells and 40 poetically-named "engines"

A pile of process docs described an elaborate apparatus — a 588-cell lattice with no algorithm, forty engines ("Whale Spiral," "Casimir Pressure") that were metaphors with no formulas. It would have been a lot of impressive-looking nothing.

Fix: Define from first principles or don't build it. We kept the honest pieces (the weighted multi-criteria gate, the E_plan scorer) and cut the decorative ones. Process is not progress. Shipping the deliverable is.

The thread through all five: **the failure mode is always a claim that floats free of the artifact.** A summary that isn't the tree. A hash that isn't the bytes. A sigma that isn't derived. A green that was never broken to prove it could go red. The entire discipline is one stance applied relentlessly — make the claim and the evidence the same object, then refuse to call anything done until they agree.

Work with Mike

I build AI systems that survive contact with production — the deterministic floor, the gates, the proof object, the human brake. If your AI work demos beautifully and then can't be reproduced, audited, or trusted with a customer, that's the gap I close.

I'll teardown your build against this doctrine, show you exactly where the claims float free of the artifacts, and give you the packet chain and gate-set to fix it. No retainer required to start — a **60-minute working session** where we run your hardest mission through the spine and you keep the ProofPacket.

Bring me the thing that won't ship. We'll find out why — with proof, not opinion.

— Email **Mike@RodgersIntelligence.com** or call **(262) 343-5680**.