No.10

FIELD MANUAL

YOUR AGENTS CAN WRITE CODE. THIS MAKES THEM SHIP IT.

THE STRIKE METHODOLOGY

One week. Three platforms. Two massive repos refactored in parallel by a fleet of agents that prove every claim before it lands. This is the exact operating system behind it: every rule, every gate, every receipt. Wire it in and run your builds like a strike team.

FERROX LABS

SEAN DONAHOE

JULY 2026

01 / THE PROBLEM

Parallel agents don't fail at writing code. They fail at integration.

Run five agents in five worktrees and watch what happens. Every one of them writes decent code. Every one of them passes its own tests. Then you merge, and the result is on fire.

If you've run multi-agent setups on anything bigger than a toy repo, you've lived this. Agent one and agent three both touched the migration sequence. Two of them regenerated the lockfile. Someone's tests passed against a baseline that stopped existing an hour ago. Each agent, individually, did its job. The combination is garbage.

The industry's answer so far has been vibes. Spin up more agents, point them at the backlog, hope the merge gods are kind. And when it breaks, the conclusion people reach is that parallel agents "aren't ready." Wrong conclusion. The agents are fine. The process around them is missing.

> **The bottleneck moved from writing code to proving code, and almost nobody updated their process to match.**

For the past week I've been running a major refactor across three platforms, two of them massive repos, using a methodology we call the Strike. It's now adopted across our entire team, and it's been cross-audited by two independent systems that both came back saying the same thing: this is how you do it.

This manual is the whole system. Not the sanitized conference-talk version. The actual operating procedure, including the boring rules that exist because we paid for them.

02 / THE CORE MOVE

Parallelize construction. Serialize integration. Make evidence the currency.

That's the entire trick, and I'm telling you on page three because it's the sentence everything else hangs off.

Most teams get the first part and skip the second. Agents build in parallel, then everything merges in parallel too, and the seams shred each other. The Strike splits the work into two zones with completely different physics:

ZONE 1 Construction. Wide open throttle.

Isolated worktrees, packets running concurrently, agents building full vertical slices with their own focused tests. This is where the speed lives. Push it as hard as your merge queue can absorb.

ZONE 2 Integration. One at a time. No exceptions.

A serial merge queue. Every packet lands alone, and the next one in line rebases and re-proves itself against the new HEAD before it gets to land. Parallelism ends at the queue. Always.

And underneath both zones, one rule that never bends: **claims are worthless, receipts are everything.** "All tests pass" is a sentence. A receipt is a commit hash, the exact command, the exit code, a log digest, and the environment it ran in. Anyone can re-run it and get the same answer. If they can't, it isn't evidence.

> **Speed and evidence aren't in tension. They're in tension only when you put the discipline in the wrong places.**

The loop, end to end:

ARM → FREEZE → PARTITION → BUILD → PROVE → INTEGRATE → ATTACK → PROMOTE

The rest of this manual walks each stage and, more importantly, tells you which disaster each rule exists to prevent. None of these rules are theoretical. Every one has a body behind it.

Arm and freeze

Arm: pin the world before the world moves

Before any agent writes a line, you record the environment digest. Container image, compiler versions, node, rust, python, the lot. "Passed locally, failed in CI" is not a mystery. It's an unpinned toolchain, and you kill it here, before it exists.

Then you create the single most underrated file in the whole system: **STRIKE.md**, committed at the repo root. It holds the packet table, the merge queue order, and every receipt. The strike's state lives in the repo, never in an agent's head. An agent that crashes or loses its context resumes from the file, and picks up exactly where the evidence says things stand. Not where it vaguely remembers.

Two numbers get set here too. A concurrency cap (we default to three packets at once) and a packet timebox (default 90 minutes). More on why the cap matters later, but the short version: ten parallel packets feeding a serial queue is nine packets waiting.

Freeze: one baseline, tagged, immutable

Identify the exact accepted baseline commit. Tag it. Record the hash in STRIKE.md. Every packet branches from that tag and nothing else.

This sounds trivially obvious and gets skipped constantly. Agents will happily branch from whatever HEAD they woke up on, and now your five packets were built against three different realities. When focused tests disagree at integration, you'll have no idea whether the code is wrong or the baselines never matched. Freeze first. Argue with evidence later.

> **Current repo evidence outranks any prior summary. Including your own. If a summary and the repo disagree, the repo wins.**

That rule sits at the top of the recon step for a reason. Agents love their own previous conclusions. A week into a big refactor, the summaries pile up, and the drift between what an agent believes about the codebase and what's actually in it becomes its own bug class. The fix is cheap: re-inspect before you trust, every time.

04 / PARTITION

Strike packets: contracts, not tickets

A packet is not a task on a board. It's a contract, declared in STRIKE.md before construction starts, and every field exists because its absence has burned someone.

- **File ownership.** Which files this packet owns exclusively. Nobody else touches them.
- **Authority boundaries.** What this packet may decide, and what it may not.
- **Invariants.** What must stay true the entire time it's working.
- **Non-claims.** What the packet explicitly does NOT assert or deliver. Writing down what you're not claiming is half the audit done in advance.
- **Tests and acceptance evidence.** What proves it, and what receipt closes it.
- **Timebox.** Blow past it and you escalate. You do not go quiet.

The seams that lie to you

Here's the trap that catches everyone: disjoint file lists do not mean disjoint work. Two packets can own completely separate files and still destroy each other, because some artifacts are shared by nature no matter who "owns" them:

```
Package lockfiles          // two regenerations, one survivor
Database migrations        // both grab the next ID, boom
Schema registries          // shared definitions, silent conflicts
Generated code             // codegen output is one namespace
DI wiring / shared config  // env vars, service registration
Shared test fixtures       // fixture registries drift apart
```

The migration one deserves its own paragraph because it's the classic. Packet A and packet B each create a migration. Each grabs the next sequential number. Each passes its focused tests beautifully, because in its own worktree, its number IS the next number. Integration detonates. Both packets did everything right by their own lights, and the combination is still broken.

> **These seams are radioactive. They integrate sequentially, always, regardless of what the ownership declarations say.**

The parallelization rule falls straight out of this: run packets concurrently only when they share zero files, zero state, zero schemas, zero authority, and zero seams. When in doubt, serialize. A packet that waits an hour costs you an hour. A seam collision costs you a day of forensics.

05 / BUILD

Vertical slices, heartbeats, and the two rules agents hate

Each packet builds in its own isolated worktree, branched from the frozen baseline, and delivers a complete vertical slice. Implementation, schema and fixtures, hostile tests, drift protection, defined failure behavior, and a factual receipt. Not "the code plus tests later." The whole slice, or the packet isn't done.

Hostile tests are worth a beat here. Happy-path tests prove your code works when the world cooperates. The world does not cooperate. Every packet ships tests that actively attack its own inputs: malformed evidence, out-of-order events, duplicates, the works. If your fail-closed behavior isn't tested, you don't have fail-closed behavior. You have a hope.

Heartbeats: silence is a failure state

Every packet reports status to STRIKE.md at least every 30 minutes of active work. A packet that goes silent past its timebox is STALLED, and stalled means escalate. It does not mean sit there blocking the dependency chain while everyone assumes it's fine. On a multi-repo strike, one silently spinning agent can stall an entire chain for hours before anyone notices. The heartbeat makes silence visible, and visible problems get fixed.

Rule one agents hate: no drive-by fixes

Mid-packet, your agent finds an adjacent bug outside its file ownership. Every instinct it has says fix it, it's right there, it's two lines. **It does not fix it.** It logs the find as a new packet proposal in STRIKE.md, with severity and evidence, and gets back inside its own boundary.

Brutal? Slightly. But the drive-by fix is exactly the untracked drift this whole methodology exists to prevent. It's a change nobody owns, nobody declared, and nobody will remember at audit time. The bug gets fixed either way. The difference is whether it gets fixed inside the system or as a landmine.

Rule two: a flake is a defect

> **Rerun-until-green is evidence laundering. It is prohibited.**

A test that fails and then passes on rerun never counts as passing. Quarantine it, open a packet for it, count it as a finding. I know how tempting the rerun is. Everyone knows. That's precisely why it has to be a hard rule and never a judgment call, because the judgment always goes the same way at 2am, and every laundered flake corrupts the one thing the whole system runs on.

06 / INTEGRATE

The merge queue. This is the part that saves you.

If you take one mechanism from this manual, take this one. It's the least obvious piece and it carries the most weight.

Integration is serial. Packets enter the queue in dependency order with a green focused proof, and then:

1. One packet lands onto the integration branch. Alone.
2. The next packet in queue rebases onto the new HEAD.
3. The rebased packet RE-RUNS its focused proof.
4. Green? It lands. Not green? It stays out. No debate.
5. Every landing logged: packet, commit before, commit after, re-proof receipt.

Step three is the one everything hinges on, so let me say it plainly. The moment packet A lands, packet B's proof is no longer evidence about the codebase. B's tests passed against a commit that stopped being HEAD the second A merged. That proof is now history. A fact about the past. And merging on the strength of it is merging on faith.

> **A proof against a commit that is no longer HEAD is evidence about history, not about the present.**

This is also where the concurrency cap from the Arm phase earns its keep. The queue absorbs one packet at a time, and each landing forces a rebase-and-re-prove on the next in line. Throw ten concurrent packets at that and you haven't bought speed, you've bought a car park. Nine packets idling, burning tokens, going stale against a HEAD that moves every landing. Three packets feeding a smooth queue will finish a strike faster than ten packets fighting over it. We've watched it happen.

Yes, the re-proof step costs minutes per landing. Agents will try to skip it, because their proof "just passed" and surely nothing relevant changed. Sometimes they're even right. But the times they're wrong are exactly the times you needed the check, and you cannot tell the two apart in advance. That's the entire point.

After the final packet lands, the aggregate gates run against the exact integrated HEAD: full test suite, lint, typecheck, security scan, build, deployment proof. All of it pinned. Producer versions, schema digests, fixture digests, correlation rules, and the environment digest from day one.

07 / THE EVIDENCE SPINE

Receipts, real fixtures, and why prose proves nothing

The receipt schema

Every proof in the system, focused or aggregate, produces a receipt in this exact shape:

```
commit:      exact hash
command:     exact command executed, verbatim
exit_code:   integer
log_digest:  sha256 of the full output log
timestamp:   ISO 8601 UTC
env_digest:  toolchain digest from the Arm phase
```

A receipt is valid only if anyone can re-run that command at that commit in that environment and reproduce the result. That's the whole test. An agent writing "all 847 tests passed, the module is production-ready" has written you a claim. Possibly a true one. But claims don't compound and receipts do, because a receipt survives the agent that made it.

Fixtures: reality outranks assumption

Two rules here, and the second one is the sneaky one.

First: fixtures get replayed through the real consumer, the actual reducer, the genuine pipeline. "It deserializes" proves the shape parses. It proves nothing about behavior, and behavior is what you're shipping.

Second: for anything provider-facing (cost, usage, billing receipts), the critical-path fixtures must be **captured from real provider responses**, not handcrafted. A handcrafted fixture encodes the author's understanding of what the provider sends. If that understanding is wrong, your tests now verify the mistake, at 100% pass rate, forever. Beautiful green checkmarks confirming a fantasy. Capture from reality, record the source and digest, and let the provider's actual weirdness into your test suite where it belongs.

Fail closed. On all of it.

Malformed evidence, version drift, gaps in the event stream, conflicting duplicates, out-of-order events, post-terminal events, unknown critical fields, missing authoritative usage or cost. Any of these appears, the system stops and says so. It does not guess, coerce, patch over, or continue on best effort.

And one line on money, because for us this system routes real billing: authoritative cost and usage come only from observed provider receipts, correlated across the full chain. Request, attempt, retry, fallback, response, ledger entry, customer debit. A model's own claim about what

08 / ATTACK

The author never audits its own work

Once everything's integrated and the aggregate gates are green, you attack the result. And the single most important word in this phase is "independent."

An agent auditing its own packet will find nothing. Not because it's lazy or dishonest, but because it's grading its own homework with its own blind spots. Whatever misunderstanding produced the bug also produces the audit that misses it. Same weights, same gaps.

So the cross-auditor is always a different agent, and preferably a different model entirely. Different training, different failure modes, different blind spots. What one model glides past, another trips over, and the tripping is the value. We run this as a standing pattern across our stack and it catches things single-model review reliably misses.

Severity belongs to the auditor

The auditor assigns severity. The author never downgrades a finding. This rule exists because without it, every HIGH mysteriously becomes a MEDIUM around the time reopening the packet starts looking like work. Disputes escalate to a human. They do not get negotiated between the two agents.

BLOCKER
Corrupts accounting truth, receipt authority, correlation, security, or rollback. Reopen the packet. Now.

HIGH
Wrong behavior on realistic inputs. Missing fail-closed handling. Drift exposure. Reopen the packet.

MEDIUM
Correct but fragile. Observability gaps. Missing hostile test. Follow-up packet; blocks only if clustered.

LOW
Cosmetic. Naming. Docs. Logged and moved past.

The loop is simple and non-negotiable: every HIGH or BLOCKER reopens the responsible packet, the fix goes back through the merge queue like any other change, and the audit repeats against the new exact HEAD. You are done when the auditor finds zero HIGH or BLOCKER findings against the current HEAD. Not when the list "looks manageable." Zero.

Cross-audit is also how this methodology itself got hardened, for what it's worth. Two independent systems went through it, both found real gaps, and the version in your hands is what survived. The process eats its own cooking.

09 / PROMOTE

"Production-ready" is six separate states, not a feeling

The phrase "production-ready" has ruined more weekends than any bug ever written. Here it isn't a judgment call at all. It's a ladder, and every rung is earned independently, in order.

1 Implementation — the code exists and its proofs are green
2 Fixture acceptance — real fixtures replay clean through real consumers
3 Packaging — it builds into a deployable artifact
4 Deployment — the artifact actually deploys
5 Live canary — it survives real traffic, measured
6 Release acceptance — a human signs off, on evidence

Code presence proves state one. Local tests prove state one. Neither says a single thing about states two through six, and the collapse of all six into one vibes-based "yeah it's ready" is where production incidents come from.

Canary criteria get written before the canary flies

Numeric thresholds for error rate, latency percentiles, cost-correlation match rate, plus whatever invariants the specific packets carry. Defined in advance, in writing. A minimum soak window (we default to 60 minutes of representative traffic) with no early promotion because it "looks fine." And an auto-rollback trigger: any threshold breach rolls back immediately, without waiting for a human to decide, and then escalates.

Define the criteria after the canary is live and you'll define whatever the canary happens to be doing. That's not a gate. That's a rubber stamp with a dashboard.

Schrodinger's rollback

> **A rollback that has never run is a rollback that does not exist. Exercise it, for real, before release acceptance.**

Every team has a documented rollback procedure. Almost no team has recently run one. Which means almost every team's rollback exists in a superposition of working and not working, and you find out which during the worst possible fifteen minutes of your quarter. So the Strike makes it a gate: before release acceptance, the rollback path executes once, for real, against staging or the canary environment, and produces its own receipt. Boring when it works. That's the point. You want the boring version, on your schedule, not the exciting version on production's.

10 / WHERE THE SPEED ACTUALLY COMES FROM

The human is a gate, not a bottleneck

Everything so far sounds like discipline, and discipline sounds slow. So here's the part where the speed comes from, because this system is fast, and it's fast for reasons most "move fast" setups get exactly backwards.

Two-tier authorization

The naive safe version of multi-agent work puts a human approval on every merge. Congratulations: you built a fast highway with one toll booth. Your parallel worktrees all queue up behind one person's attention span, and the strike serializes at exactly the point where you needed it not to.

So the gate splits in two:

TIER 1 **Pre-authorized. Zero human wait.**

A packet merges to the integration branch the moment three things are true: its post-rebase proof is green against current HEAD, it stayed inside its declared file ownership, and its receipt is complete. Machines verify all three. Nobody waits for a human to read a diff at midnight.

TIER 2 **Human only. Hard stop.**

Merge to main. Closing coordination issues. Release. Deploy. Canary promotion to full traffic. These wait for explicit sign-off, every time, and passing the gates earns a request for authorization. It never earns the authorization itself. No "it passed so I shipped it."

Human judgment gets spent where it's irreplaceable, on the promotion decisions, and gets taken entirely off the critical path everywhere machines can verify the evidence themselves. That's the trade, and it's the whole reason the parallelism survives contact with reality.

And the rest of the speed is subtraction

The frozen baseline deletes "which reality was this built against" forensics. The seam rules delete migration collisions. The re-proof step deletes the broken-integration debugging day. Heartbeats delete the silent four-hour stall. Receipts delete the re-litigation of what already passed. None of these make construction faster. All of them delete the rework that eats multi-agent projects alive, and rework is where the time actually goes. A strike feels slower per packet and finishes dramatically sooner per project. A week into running this across three platforms at once, I'll take that trade every single day.

Steal this

Strip away every detail and the Strike compresses to one line:

FREEZE → PARTITION → BUILD VERTICALLY → PROVE LOCALLY → INTEGRATE EXACTLY

PROVE GLOBALLY → ATTACK THE RESULT → PROMOTE ONLY THE EVIDENCE-BACKED STATE

Nothing in that line is specific to my stack, my repos, or my agents. It works for a solo builder running two worktrees on a side project and it works for a team running a fleet across three platforms, because the failure modes it targets are universal. Baselines drift everywhere. Seams collide everywhere. Proofs go stale everywhere. The physics don't care how big you are.

Ten rules, if you want the fridge-magnet version:

- Current repo evidence outranks any summary, including your own
- Strike state lives in a committed file, never in an agent's memory
- Integration is serial; every landing forces the next packet to rebase and re-prove
- No fixes outside your packet's ownership. Discovered bugs become new packets
- A flake is a defect. Rerun-until-green is evidence laundering
- The auditor is never the author, and the auditor owns severity
- Model claims never mint authoritative receipts
- Rollback is exercised before release, not merely documented
- Machines gate integration; humans gate promotion
- Fail closed on everything malformed, drifted, duplicated, or missing

The full methodology is packaged as a drop-in skill that works with Claude Code, agent frameworks, and anything else that reads a SKILL.md. The complete file is in the appendix, right after this page. Copy it, paste it into your systems, run your first strike on something real, and come tell us in The Forge what broke. That's how this version got good.

Sean Donahoe is the founder of Ferrox Labs and runs The Forge, a community for AI builders and creators shipping real systems. He is the CEO and founder of Flux Router and the creator of the Wayland desktop agent.

APPENDIX / THE SKILL

Copy. Paste. Strike.

Everything below is the complete skill file, verbatim. Select it all, copy it, and save it as **wayland-strike/SKILL.md** in your skills directory. Works with Claude Code, Cowork, and any agent framework that reads SKILL.md files. The frontmatter description handles triggering; the body is the operating procedure your agents will follow.

```
---
name: wayland-strike
description: |
  The Wayland Strike methodology for executing remaining project work with parallel
  agents under strict evidence discipline: freeze baseline, partition into strike
  packets, build in isolated worktrees, integrate through a serial merge queue,
  adversarially cross-audit, and promote only evidence-backed states. Use this skill
  whenever Sean says "run a strike", "strike this", "Wayland Strike", "adopt strike
  methodology", asks to parallelize remaining work on a repo, asks to coordinate
  multiple agents or worktrees on a codebase, wants dependency-ordered work packets,
  or asks how to finish a project fast without weakening receipts, accounting truth,
  rollback, correlation, security, or release gates. Also use when planning any
  multi-packet build, integration sequence, canary, or release acceptance process.
---

# Wayland Strike

You are executing a strike: a fast, parallel, evidence-locked completion of remaining work. Speed is the goal. Evidence is the constraint. When they conflict, evidence wins, and you find speed somewhere else.

The full methodology lives in `references/methodology.md`. Read it once at strike start. This file is your operating procedure.

## The loop

arm → freeze → partition → build vertically → prove locally → integrate exactly → prove globally → attack the result → promote only the evidence-backed state.

## Non-negotiables (memorize these first)

1. Current repo evidence outranks any prior summary, including your own.
2. The strike state lives in `STRIKE.md` at the repo root, committed. Never in your memory. Crashed agents resume from the file.
3. Integration is serial. Parallelism ends at the merge queue. One packet lands at a time; the next rebases and RE-RUNS its focused proof before landing. Stale proofs are history, not evidence.
4. You never fix anything outside your packet's file ownership. Discovered bugs become new packet proposals in `STRIKE.md`. No drive-by fixes, ever.
5. A flaky test is a defect. Rerun-until-green is evidence laundering and is prohibited.
6. The auditor is never the author. Cross-audit uses a different agent, preferably a different model. The auditor assigns severity; the author never downgrades a finding.
7. Candidate or model claims never mint authoritative receipts. Authoritative cost and usage come only from observed provider receipts, correlated across request → attempt → retry → fallback → response → ledger entry → customer debit.
8. Rollback must be exercised once for real before release acceptance. Documented rollback does not count.
9. Two-tier authorization. Tier 1 (pre-authorized, no waiting): packet merges to the integration branch with a green post-rebase proof, inside declared ownership, with a complete receipt. Tier 2 (Sean only, hard stop): merge to main, closing coordination issues, release, deploy, canary promotion. Passing earns a request for authorization, never authorization itself.
10. Fail closed on: malformed evidence, version drift, event-stream gaps, conflicting duplicates, out-of-order events, post-terminal events, unknown critical fields, missing authoritative usage or cost.

## Phase procedure

### Phase 0: Arm
- Record the environment digest (container digest, language/toolchain versions) in `STRIKE.md`.
- Create `STRIKE.md` if absent: packet table, merge queue, receipts log.
```

- Set concurrency cap (default 3 concurrent packets) and packet timebox (default 90 min; oversized packets get split).

Phase 1: Recon
- Inspect repo, branches, PRs, CI, issues, live deployment state. Record findings with commit hashes and timestamps.

Phase 2: Freeze
- Tag the accepted baseline: `strike/baseline-<date>`. Record the hash. All packets branch from it.

Phase 3: Partition
Each packet declares in `STRIKE.md` before construction: file ownership, authority boundaries, invariants, non-claims, tests, acceptance evidence, timebox.

Always-sequential seams, never parallelized regardless of declared ownership: lockfiles, database migrations and their numbering, schema registries, generated code, DI/config wiring, shared fixtures. Parallelize only packets with zero shared files, state, schemas, authority, AND seams. When in doubt, serialize.

Phase 4: Build
- Isolated worktrees, one per packet, branched from baseline.
- Vertical slice per packet: implementation, schema/fixtures, hostile tests, drift protection, defined failure behavior, factual receipt.
- Heartbeat: status to `STRIKE.md` at least every 30 minutes. Silent past timebox = STALLED = escalate to Sean, never silently block.
- Apply the flake policy and discovered-work protocol from the non-negotiables.

Phase 5: Integrate (serial merge queue)
1. Enter queue in dependency order with green focused proof.
2. Land one packet onto the integration branch.
3. Next packet rebases onto new HEAD.
4. Rebased packet re-runs focused proof. Green? Land. Not green? It stays out.
5. Log every landing: packet, commit before, commit after, re-proof receipt.

Phase 6: Prove
Against the exact integrated HEAD: full test suite, lint, typecheck, security scan, build, deployment proof. Pin producer versions, schema digests, fixture digests, correlation rules, and the Phase 0 environment digest. Replay fixtures through the real consumer/reducer; deserialization proves nothing. Critical-path provider fixtures (cost, usage, receipts) must be captured from real provider responses, not handcrafted, with source and digest recorded.

Phase 7: Attack
Independent agent cross-audits the exact HEAD. Severity ladder:
- BLOCKER: corrupts accounting truth, receipt authority, correlation, security, or rollback. Reopen packet now.
- HIGH: wrong behavior on realistic inputs, missing fail-closed handling, drift exposure. Reopen packet.
- MEDIUM: fragile-but-correct, observability gaps, missing hostile test. Follow-up packet; blocks promotion only if clustered.
- LOW: cosmetic. Logged.

Loop until zero HIGH/BLOCKER against current exact HEAD.

Phase 8: Promote
Six independent states, earned in order, never conflated: implementation → fixture acceptance → packaging → deployment → live canary → release acceptance.

Canary criteria defined BEFORE canary starts: numeric thresholds for error rate, latency percentiles, cost-correlation match rate, and packet-specific invariants; minimum soak (default 60 min representative traffic, no early promotion on "looks fine"); auto-rollback on any threshold breach without waiting for human judgment, then escalate. Execute the rollback drill against staging or canary and capture its receipt before requesting release acceptance from Sean.

Receipt schema

"All tests pass" is a claim. This is a receipt:

```
commit:       <exact hash>
command:      <exact command, verbatim>
exit_code:    <integer>
log_digest:   <sha256 of full output log>
timestamp:    <ISO 8601 UTC>
env_digest:   <toolchain digest from Phase 0>
```

Valid only if reproducible by re-running the command at that commit in that environment. Prose is not a receipt.

Reporting format

Maintain live in `STRIKE.md`:

```
Packet  Dependency  Status  Exact commit  Focused proof  Aggregate proof  Remaining blocker
──────  ──────────  ──────  ────────────  ─────────────  ───────────────  ─────────────────
```

Status values: PLANNED, BUILDING, STALLED, QUEUED, LANDED, REOPENED, ACCEPTED. Proofs are receipt IDs, not assertions.

Optimization priority

Optimize for rapid completion, but speed never weakens: receipt authority, accounting truth, exercised rollback, correlation integrity, security gates, release gates, merge queue serialization, auditor independence.