Task Resolution Protocol

Take a ticket from assigned to merged: ground it in the real code, get the plan approved before anything moves, then build, prove, and close it out.

Produces	A merged PR that meets the acceptance criteria, plus a two-layer ticket update — a plain-language summary for the business and the technical detail for engineers.
Use it when	A ticket is assigned and needs delivering to a professional standard: understood against the real repo, planned, approved, implemented, verified, reviewed, and closed.
Never	Execute before approval. No branch, no write, no head-start on the code until a human says go.

What it is

Delivering a ticket well is not reading the title and starting to type. It is understanding what the ticket actually asks against the code that exists, writing down exactly what you intend to change and why, getting a human to say yes, and only then building it. You build it with the discipline that what you ship matches what you promised, every check is green before you push, and the ticket ends up readable by the person who filed it. The protocol is a sequence with one hard stop in the middle: you present a plan and you wait. Nothing that is hard to undo happens on the near side of that stop.

You reach for this whenever a ticket is yours to deliver to a real standard: a feature, a bug fix, an infra change, a security fix. It is the difference between "I closed the ticket" and "I delivered the thing the ticket asked for, proved it, and left a trail anyone can follow." What you hold at the end is concrete: a pull request merged into the trunk that meets the acceptance criteria, and a ticket carrying two layers of update. Those two layers are a summary a project manager can read without knowing the codebase, and the technical detail an engineer needs.

The hard boundary is the approval gate. Everything before it is read-only: you read the ticket, you read the code, you write the plan. You do not create a branch, you do not write a line, you do not spin off a subagent to get a head-start on the "obvious" part. The plan is presented and the work stops until the owner approves it. This is not ceremony. A plan grounded in real code, seen before any code exists, means a course-correction costs an edit to a document instead of a day of thrown-away work. Once approved, the second half runs: implement, verify, self-review, open the PR, close the ticket.

How you'd do it by hand

The scripts in the skill wrap three boring, deterministic steps — pull the ticket, gate the plan, post the update — around the judgment that can't be scripted. Strip them away and the method is the same. Here it is end to end, as a person with `git`, a git host CLI, and a ticket login would do it.

The whole thing is a phase machine: 0 through 5, with a stop at 1. Walk it in order.

Phase 0 — Ground it in the real code

Pull the ticket and read *all* of it, comments included. The half-line that redefines the scope usually lives in a comment three weeks down the thread, not in the description. On a ticket tracker with a REST API that is one authenticated GET; in a browser it is opening the ticket and scrolling. Either way you read it whole, and you don't write to it yet. If the ticket names an evidence file — a spec, a gap-list, a linked doc — read that too before you scope anything.

Then ground the ask in the actual repo. Open the files it touches, read the functions, and for every claim you're going to make, be able to cite `file:line`. Where you can, confirm the described behaviour in the running artifact rather than trusting the description. A root cause you can only call "suspected" or "likely"

or "probably" is not a root cause — it is a guess, and a guess does not ship. Either you traced the failure to a line and reproduced it, or you keep digging until you can.

The load-bearing habit here is *discover, don't punt.* Every question the code, the config, the telemetry, or the pipeline can answer, you answer yourself. You do not hand it back to the owner. The only thing that surfaces to a human is a decision that genuinely lives in someone's head: a business rule, a priority call, an access grant, the wording of something a stakeholder will read. And even that surfaces as a decision with options and a recommended default, never as an open-ended "what do you want me to do here." When the ticket's wording contradicts what the code actually does, you make the call and say so out loud: "Decision, mine not yours: the ticket says X, the code does Y, here's the evidence, I'm going with Y."

Phase 0 is done when every acceptance criterion is mapped to a real piece of code, and every gap is either closed with evidence or surfaced as a named decision.

Phase 1 — Present the full plan, and stop

Now you write the plan: what changes, in which files, and why; the tests that will prove it; the risks; where relevant, the cost. You restate the ticket's acceptance criteria against their original source, so nothing gets quietly narrowed. A criterion that reads "the plan shows it and the report confirms the line item" is not met by showing only the plan.

The plan carries every mandatory piece. How environment variables and config are handled. How the change gets verified end to end, on the real surface. What *can't* be verified locally, named honestly, with what stands in as proof. How the PR gets opened. How the ticket gets updated. A plan missing one of these is not a rough draft to be filled in later; it is incomplete, and presenting it as ready is the violation.

Then you stop. This is the gate, and it is absolute. No branch, no write, no implementation subagent, nothing — until the owner approves. (The read-only research pass that wrote the plan may itself run in a subagent; it returns the plan and nothing else.) Approval is for this plan; it does not roll forward to the next, separate piece of work. You never offer a menu of options where one of them is a scope-drop dressed as a choice ("we could defer this part to a follow-up"). Offering the wrong option is itself the failure, even when the human picks the right one.

Phase 1.5 — Break up anything too big for one PR

A single-PR ticket skips this: you write a short table of the commits and the files each touches, and move on. When the change is bigger, you split it into small PRs that each merge to the trunk safely on their own, in dependency order. You do not open a tall stack of dependent branches and hold them all open — that guarantees drift. Where the platform has a feature-flag system, incomplete work can land behind a default-off flag. Where it doesn't, each PR has to be *correct on merge*, so the breakdown is shaped so nothing half-built ever reaches the trunk. State which case you're in.

Phase 2 — Implement in small, verified steps

Branch off the default branch, following the repo's own naming. Then work in task-then-verify pairs: make one change, check it against the specific plan item it satisfies, then the next. You define success for each step and loop until it's met, rather than marching down a fixed list and hoping the sum is right.

Match the codebase's conventions even where your taste differs — naming, structure, test style. Conformance beats taste inside someone else's repo. Every item you got approved is mandatory; none of them quietly become "optional" or "for now" or "a follow-up" when you hit friction. Sometimes something turns out harder than the plan assumed — an unfamiliar config shape, a missing fixture, an API that doesn't behave as expected. The response is to dig into the docs and source until you can do it properly, not to shrink the scope. If you delegate any of this, the instructions you hand off carry the gates in full, because a delegate follows only what it was told, not the protocol in your head.

Phase 3 — Verify against the real gates

Before *any* push, run the project's own configured checks and require them actually green. Discover the commands from the repo — the formatter, the linter, the affected tests, the build, the typecheck — don't assume them. Run them and read the real result.

```
# The exact commands come from the repo (package.json scripts, Makefile, CI config).
# The shape is always: format, lint, affected tests, build, typecheck — all green.
<formatter> --check          # e.g. prettier --check, gofmt -l, black --check
<linter>                     # the repo's lint task
<test-runner> <affected>     # the affected suites — passing, not just "ran"
<build>                      # the production build
<typecheck>                  # where the repo runs one
```

The trap to know: many repos run tests as continue-on-error in CI, so a green CI badge can sit on top of a failing test. A green check is not proof. Gate on the real verdict — the actual test output — not the colour of the check. A failing test is a hard block; you do not push red. The only exception is a failure you can *prove* is pre-existing on the trunk baseline and unrelated to your change. You prove it by stashing your diff and watching it fail without you. Scan the commit message for AI attribution before every push (the exact grep is in the cheatsheet below).

Then the change-specific battery. Exercise the *real* surface the change lives on — if the change is about a save path, drive a real save, don't test the DOM around it. Where a flag exists, prove the change is inert with the flag off. Enumerate every entrypoint and call-site of the behaviour you changed, across the *whole* repo — scripts, tooling, migrations, seeds, not just the `src/` directory. A grep scoped to `src/` structurally cannot see a prod-capable seed script wired as an npm task, and that is exactly where a live-data surprise hides. State the search you ran, so the scope itself is reviewable. Anything you genuinely can't prove, you report as unproven — never silently passed.

Phase 3.5 — Confirm, don't discover

This phase exists so that when a human reviewer first looks at your PR, they *confirm* your work rather than *discover* its bugs. Two passes, before the diff is review-visible where possible.

First, read your own diff cold, like a hostile external bug-hunter who wants to find something wrong with it. This is not "tests pass and CI is green" — that is a different, weaker thing. You are hunting for newly-introduced defects against the classes that recur:

- read-modify-write with no concurrency token (two writers clobber each other)
- check-then-act across a boundary (two callers both pass the guard)
- an error-mapping change that flips a status but leaves every consumer of the old signal unhardened
- empty-string input that slips past an `if (value)` filter and widens a query
- a CLI guard that fires on boot once the script is bundled When you fixed a bug of some class, check specifically that the fix didn't spawn a sibling of the same class — fixes are the single highest-risk place for that.

Second, run the automated reviewer if the repo has one configured, locally, before you push, and fix every actionable finding to zero. Then, once the PR is open, run a full review pass over it and fix *every* finding — blocking and not. A question-finding gets answered in the PR body or a comment. Push the fixes, re-review at the new head, repeat until it comes back clean. If the reviewer's first pass on the visible PR turns up a real defect, that means this gate didn't run well enough. It is a gate failure to strengthen, not "the process working." The bar is confirm, not discover.

Phase 4 — Open the PR and hit the done-bar

Detect the platform from the repo's `origin` remote and route accordingly — never mix the two. A `github.com` remote means the `gh` CLI. A `dev.azure.com` remote means an `az` bearer token against the Azure DevOps REST API, because the `az repos` extension is often broken and REST is reliable.

```
# GitHub
gh pr create --base <default-branch> --title "<id> <desc>" --body-file pr.md
gh pr checks <n>       # gate on the real result, not the badge colour
```

```
# Azure DevOps — mint a bearer, hit REST with org/project/repo as variables
TOKEN=$(az account get-access-token \
  --resource 499b84ac-1321-427f-aa17-267ca6975798 --query accessToken -o tsv)
BASE="https://dev.azure.com/<ORG>/<PROJECT>/_apis/git/repositories/<REPO-ID>"
curl -s -H "Authorization: Bearer $TOKEN" "$BASE/pullrequests?api-version=7.1"
```

A couple of platform edges worth knowing. Azure DevOps caps a PR description at 4000 characters, so `wc -c` the body and trim before the call, cutting the least load-bearing sections first. When an Azure DevOps merge is blocked, query the branch policies before guessing the cause. Only a policy marked blocking can stop a merge; "required reviewers" is usually advisory, and the real gate is often a minimum-reviewers count that any one approval satisfies. And don't force-push after a reviewer has voted; on ADO that stales the vote even when policy says it shouldn't. Add a follow-up commit and re-request instead.

The PR is not done until command output confirms all of it:

- assignee set and reviewers requested
- checks green with no new failures versus the baseline
- zero unresolved review threads, the automated reviewer's included, on the current head
- affected tests passing locally
- every number in the PR body re-measured against the actual head Fill the body from the repo's template, with real detail — the changelog, what testing was done and its results, screenshots where they help.

Phase 5 — Close the ticket, both layers

Two mechanical actions, both required, every round. Not one.

First, move the status to match where the work really is — in progress when you start, in review when the PR opens, complete only after it's merged and verified. Second, post a two-layer comment. First a plain-language business summary a project manager can read: what this does for users, why it matters, what's left, no file paths, no jargon. Then the technical detail: the PR link, the changelog, the verification evidence. The ticket has to be legible to a non-engineer *and* an engineer.

```
**Summary (non-technical):**
<2–5 sentences a PM reads: what this does, why it matters, status, what's left.>

---

**Technical detail:**
- PR: <link> (<state>)
- Changelog: <per-file bullets>
- Verification: <gates + real-surface evidence, review results>
```

Scan the comment for AI attribution before you post it, the same as a commit message. Then verify it actually landed by re-reading the ticket's latest comment: if what comes back is a system-generated event and not your two-layer write, the phase isn't done. The recurring failure here is doing the status flip and skipping the comment — both actions, every round, verified.

Why it's built this way

Every rule is scar tissue from a specific, expensive way delivery goes wrong.

The approval gate is absolute. The whole value of a plan is that it's cheap to change before any code exists and expensive to change after. Let a subagent "get a head start on the obvious part" and you've spent the budget you were trying to protect, on work that a five-minute correction to the plan might have deleted. Worse, you've taken a decision that was the owner's to take. Nothing hard-to-reverse happens before a

human says go, and that includes the parts that look too obvious to need approval. Those are exactly the ones that turn out to have been misread.

Discover, don't punt. Handing the owner a question the repo could have answered is offloading your work onto them, and it's the fastest way to lose their trust in the process. Their attention is a scarce resource; spend it only on the decisions that genuinely require their judgment. Everything the code, config, telemetry, or pipeline can tell you, you find out yourself. The corollary is that when you *do* surface something, it lands as a real decision with a default, so answering it takes them a moment, not an investigation.

Evidence first. A "suspected" root cause that turns out wrong doesn't just waste a round — it teaches everyone downstream to discount your diagnoses. Tracing the failure to a line and reproducing it in the real artifact is the price of being believed. A fix built on a guess about the cause fixes nothing and hides the real bug for the next person.

Every approved item is mandatory. The forbidden vocabulary — defer, out of scope, for now, follow-up, separate ticket, simplify to — is banned on approved work. Each of those words is a quiet scope-drop, and a scope-drop nobody agreed to is a broken promise. When approved work hits friction, the honest response is to investigate until it's done properly, not to shave the hard part into a future that never comes. Friction is not a licence to renegotiate the deal after the handshake.

Claims must match code. The PR body, the commit message, and the ticket are part of the deliverable, not decoration around it. A body that says a dimension is captured when the code leaves it at zero, a criterion marked met after it was quietly narrowed, a test count that moved after an amend and never got updated — each is a defect exactly like a code bug. After any amend or force-push, every number is stale until you re-measure it against the new head. Re-derive the baseline from the real merge-base, not your own previous head. If the body prints a command and a count, a reviewer will re-run the command, so you re-run it first.

No red check ever pushed. Continue-on-error CI means a green badge can sit on a failing test, so the badge is not the verdict — the test output is. Pushing red "to let CI sort it out" leaves a broken PR behind and burns a review round. Fix it green locally first. The only thing that excuses a red test is proof it was already red on the trunk, untouched by your change.

Confirm, don't discover. A reviewer whose first pass keeps finding real bugs learns to distrust the work. Their attention drains into defect-hunting instead of the design questions only a human should be asking. Reviewing your own diff adversarially first, and running the automated reviewer before the diff is visible, is what makes the human pass a confirmation. A backstop that keeps catching things isn't the process working — it's the primary gate failing, and it gets strengthened, not celebrated.

No AI attribution, anywhere, scanned mechanically. Nothing this work produces credits an AI — not the commit trailer, not the PR body, not the ticket, not a code comment. A tool default that wants to add a co-authored-by line does not override this. It's banned because the work goes out under a human's name and a machine credit is both wrong and, in this context, not allowed. Awareness of the rule has failed to enforce it before. That is why it's a mechanical grep over the full commit message, the PR body, and every comment before each push and post — not a thing you remember to check.

The through-line: a delivery is trustworthy exactly to the degree that nothing irreversible happened before a human approved it, every claim in it is provable against the code, and every gate actually ran instead of being recalled. Everything above serves that.

At a glance

```
# Detect the platform — route on the origin remote, never mix the two
git remote get-url origin        # github.com → gh ; dev.azure.com → az + REST

# Pre-push gate — the repo's OWN checks, all actually green (not the CI badge)
<formatter> --check ; <linter> ; <test-runner> <affected> ; <build> ; <typecheck>
```

```
# Attribution scan — before every commit push AND every ticket comment
git log -1 --format=%B \
  | grep -iE 'co-authored|generated with|claude|anthropic|opus|sonnet|haiku|🤖' \
  && echo FAIL || echo OK

# Phase 5 — TWO actions, both required, every round:
#   1. move the ticket status to match reality (→ in review on PR open)
#   2. post the two-layer comment (PM summary + technical), then re-read to confirm
it landed
```

The phase machine — walk it in order, stop at 1:

- **0 Ground** — read the ticket and all its comments, ground every claim in `file:line`, answer your own questions, surface only owner-only decisions.
- **1 Plan** — present the full plan with every mandatory section, then STOP. Nothing executes before approval.
- **1.5 Breakdown** — one PR, or several each safe to merge on their own in dependency order.
- **2 Implement** — task-then-verify pairs, match the repo's conventions, every approved item mandatory.
- **3 Verify** — real gates green, exercise the real surface, enumerate every call-site repo-wide, state the search.
- **3.5 Confirm-not-discover** — adversarial self-read, automated reviewer to zero, before the diff is visible.
- **4 PR** — open it, hit the done-bar by command output.
- **5 Close** — status transition AND two-layer comment, verified landed.

The PR-done bar — confirmed by command output, never assumed:

- Assignee set, reviewers requested.
- Checks green, no new failures versus the trunk baseline.
- Zero unresolved review threads on the current head.
- Affected tests pass locally.
- Every number in the body re-measured at the current head.

The approval gate — the one line that never bends: everything before the owner's approval is read-only. No branch, no write, no implementation subagent, no head-start on the "obvious" part. You always show the plan first.