

Protocol Runner: preserving a consequential procedure across AI workers

Evidence companion prepared September 16, 2026. Historical campaign: August-September 2026.

Implementation excerpts are pinned to repository revision ef8546a; they are source inspection, not a newly executed qualification. The review excerpt is an independently retained historical work product whose complete-file digest matches the accepted maintenance record.

Protocol Runner is a local control system for work whose procedure matters. A coordinating agent and Justin define contracts, bounded assignments and transitions. Runner validates and snapshots the work plan, records execution state, dispatches work to existing Codex Desktop conversations or fresh command-line workers, and follows the declared transitions. It does not decide whether the resulting analysis is true.

These exhibits show why that distinction matters in actual use. They connect a substantial research-and-review campaign to the implementation that keeps process state outside individual conversations, then show what happens when part of the work needs correction. The emphasis is preservation of useful work and accountable continuation, rather than how many agents can be launched.

Public implementation and observed qualification - September 23

The actual [Protocol Runner repository](#) is available under MIT. Its [engineering account](#) explains the decisions, contribution and original Desktop/Discord constraint; its [public evidence record](#) identifies the checked runtime revisions and environment.

- **Recovery with simulated workers:** the real authenticated API, persistent store and executor retain the completed sibling's attempt and exact output bytes, retry only the failed item with a new attempt identity, and preserve the failed attempt. The timeout is injected; this does not claim interruption or recovery of a live model process.
- **Real Codex CLI execution:** a separate Windows check completed one assigned worker, wrote the expected output and reported the group and run completed. An explicit stop check cancelled the attempt and lease, stopped the group, terminated the worker process and retained cancellation evidence. It correctly left that run blocked rather than counting stopped work as complete.
- **Standalone software:** the recorded Ubuntu qualification passed the package/dashboard build, focused tests, lint, recovery demo and startup/API-boundary checks. The live Windows checks used a dedicated Codex home, workspace-write and no sandbox bypass. These are scoped software, completion and cancellation results, not reasoning-quality or multi-worker scalability measurements.

The standard public workflow uses direct CLI workers. The optional Desktop/Discord path was originally built to create real Codex threads and run serial work before `codex exec` was fully released for that workflow. It remains sensitive to visible application state and has separate qualification needs.

The fixed documentation links above retain the public source account inspected for this update. The original historical campaign and source exhibits follow unchanged in their own evidence scope.

1. A real campaign that could reject its own output

A development-history update covered a fixed interval of **595 commits**. Twelve screening packets divided the interval, and **25 investigation assignments** examined coherent development families. Their results were integrated into **40 historical accounts containing 190 claims**. This required technical interpretation: whether a change introduced a capability or hardened an existing one, which system owned it, what had actually worked, and which limitations remained.

The following is a readable reconstruction of the recorded sequence. It is a curator-authored summary, not a copied execution log or a claim that one monolithic work plan performed every stage.

Stage	Concrete work and resulting boundary
Screen	Classify the fixed 595-commit interval once, in order, into twelve reviewable packets.
Investigate	Run 25 bounded investigations with declared inputs and output artifacts. The investigation plan allows up to 25 concurrent items; that setting alone does not prove all 25 ran simultaneously.
Integrate	Assemble 40 accounts and their claim/evidence mappings. Completed worker assignments still leave a draft.
Challenge	Separate AI review finds three substantive narrative defects.
Repair	Correct those three accounts, retain the other 37, and preserve completed investigations and interval classifications.
Review again	A fresh AI reviewer examines the entire corrected draft, including unchanged accounts, without the prior review's conclusions.
Accept and retain	The coordinating process accepts the corrected record through DRCE, the development-record system. Runner's execution state can close without deleting the retained review.

The defects were meaningful. One account described the creation of a capability that already existed; the evidence supported subsequent hardening. Another folded too much earlier Runner development into a narrower update. A third missed the particular identity-and-timestamp issue behind a procedural correction. Fixing them required judgment about the evidence and the claims, not rerunning a validator until its indicator became green.

The correction changed five evidence routes by removal, replaced or repaired fourteen others, and narrowed one proof limit. The final draft retained 196 evidence routes. The important preservation result is the scope of the repair: three affected accounts changed while the remaining 37 and the investigation work survived. This was coordinator-directed editorial recovery across a campaign, distinct from the mechanical item retry shown later.

Actual work-plan extract, investigation stage (X1):

```
"max_concurrency": 25,
"required_worker_capabilities": [
  "json_transform"
],
```

Only these adjacent fields are reproduced; the private input, output and execution bindings are omitted. The full plan contains 25 item declarations. Contracts supply the meaning of an investigation; this machine-readable layer supplies execution structure and capacity.

2. Review was an assignment, not automatic acceptance

The final review used a fresh command-line process with an explicitly separate role. Its plan prohibited repairing or integrating the draft, and excluded prior or sibling review conclusions. It could return a review result; it could not accept the historical record. The point was to expose the corrected work to another substantive challenge, including the supposedly unaffected portions.

Actual final-review plan extract (X2):

```
"on_completed": {  
  "action": "stop"  
},  
"on_blocked": {  
  "action": "pause"  
}
```

This is a contiguous excerpt from a larger JSON object; its opening fields and closing object brace are omitted. A completed procedural assignment stops this plan. A blocked assignment pauses it. Neither branch says that the historical claims have become accepted.

The retained final review provides something more useful than a bare success indicator: it describes the actual scope it challenged. The following paragraphs are exact excerpts, separated here from their private identity and source-location sections.

Historical AI-review output, scope (X7):

I read all twenty-five investigation packets and the complete forty-episode E161-E200 candidate, including the thirty-seven unchanged episode packages. I challenged every one of the 190 claims for proposition strength, evidence class/status, ownership, identity continuity, effect and disclosure boundaries, authorship limits, counterevidence, and neighboring dependents.

Historical AI-review output, disposition (X8):

Open semantic findings: none.

Historical AI-review output, authority boundary (X9):

Those are expected post-review lifecycle actions, not needs_attention findings. This review does not accept history, advance a selector, mutate canonical content, enable recurrence, access a provider, publish, or create a recipient effect.

These are the reviewer's recorded findings, not an independent rerun conducted while preparing this exhibit. The durable acceptance record separately establishes that the corrected candidate was accepted and that prior history remained preserved. It also records ordinary Runner closeout with the sealed review retained and its digest unchanged. The control system, reviewer and record owner did different jobs.

3. The implementation keeps the procedure outside the model

The transition function checks that a worker's reported step matches the step currently being executed. It then selects the work plan's rule for the reported status. Later branches implement pause, stop, explicit routing and advancement to the next declared step.

Actual transition implementation (X3):

```
export function resolveTransition(
  plan: WorkPlan,
  step: ResolvedStep,
  completion: StatusReport,
): TransitionResolution {
  if (completion.step_id !== step.step_id) {
    return {
      action: 'block',
      reason: `Completion step_id ${completion.step_id} does not match current step ${step.step_id}.`,
    }
  }

  const rule = completion.status === 'completed' ? step.on_completed : step.on_blocked
```

This excerpt ends before the rule-specific branches. The worker supplies a procedural status; it does not supply an arbitrary replacement sequence. That design addresses a recurring agent failure: producing plausible output while compressing, skipping or redefining the intended work. It makes the assigned structure explicit and inspectable, although a badly authored plan can still preserve a bad procedure.

Runner's API owns state mutation. A separate executor receives leases for specific parallel items, launches the workers and reports their attempts. The serial driver asks the API to dispatch eligible work rather than independently selecting transitions. Existing Desktop conversations can retain their available conversational context; persistent Runner state records the plan, cursor, attempts and evidence. These are complementary forms of continuity, not a claim to reconstruct every lost conversation.

4. Retrying one item preserves its completed sibling

A useful recovery mechanism must distinguish a failed attempt from the assignment it was attempting. The store refuses a new retry while the previous attempt is active, derives a new attempt identity, and creates a distinct evidence directory. The following excerpt shows that boundary and identity construction; the subsequent source inserts a new attempt row and updates only the selected item's current pointer.

Actual retry implementation (X4):

```
if (previous_attempt_id !== null) {
  const previous = this.loadParallelAttemptRowSync(db, safe_run_id, previous_attempt_id)
  if (isActiveAttemptStatus(previous.status)) {
    throw new RunnerStoreError(
      'store.parallel_attempt_still_active',
      `Cannot retry item while previous attempt status=${previous.status}.`,
      { run_instance_id: safe_run_id, attempt_id: previous_attempt_id },
    )
  }
}

const attempt_number = this.nextParallelAttemptNumber(db, safe_run_id, safe_group_id, safe_item_id)
const attempt_id = parallelAttemptId(safe_group_id, safe_item_id, attempt_number)
const evidence_dir = parallelAttemptRelativeDir(safe_group_id, safe_item_id, attempt_id)
```

The focused HTTP regression test sets up two items: one finishes and writes its output; the other blocks. An explicit request retries only the blocked item. The test also creates an unexpected file at the retry target and confirms that preflight rejects it, then removes that fixture obstruction and resumes the group.

Actual assertions after resuming (X5):

```
assert.equal(startedBody.preflight.passed, true)
assert.equal(startedBody.preflight.launchable_item_count, 1)
assert.equal(startedBody.run.state.status, 'running')
assert.equal(startedBody.group.items.find((item) => item.item_id === 'node_001')?.status,
'completed')
assert.equal(startedBody.group.items.find((item) => item.item_id === 'node_002')?.status, 'pending')
assert.equal(await fs.readFile(completedOutput, 'utf8'), 'retained completed output\n')
```

Actual assertions after acquiring the retry lease (X6):

```
const retryLeases = bodyAs<TestParallelPreflightEnvelope>(retryLeaseResponse).leases ?? []
assert.equal(retryLeases.length, 1)
assert.equal(retryLeases[0]?.item_id, 'node_002')
assert.equal(retryLeases[0]?.attempt_id, retryBody.attempt?.attempt_id)
assert.equal(await fs.readFile(completedOutput, 'utf8'), 'retained completed output\n')
```

These are synthetic fixture identifiers, not live operational handles. The assertions check the useful property directly: only the blocked item becomes launchable, its new lease has the expected attempt identity, and the completed sibling's output is still intact. They are selected test-source excerpts, not a newly generated passing test report. The historical campaign above demonstrates substantive preservation at a different level; it should not be substituted for a live receipt of this exact API retry path.

5. A failure that changed the execution boundary

A separate historical assignment exposed a wrong-checkout error. The managed executor launched from the shared checkout instead of the isolated worktree named by the assignment. Runner reported missing output. An artifact found elsewhere was not treated as proof that the intended operation had succeeded.

The executor was then permanently rebound to its launching worktree and the exact assignment ran again successfully. The source record distinguishes that procedural recovery from its substantive conclusion: the recovered analysis still reported incomplete readiness and retained unresolved holds. Fixing where an agent ran did not turn its answer into an approval.

This incident is represented here by the retained, committed incident summary, not by reproducing the failed worker transcript. Alongside the campaign and code, it demonstrates the recurring design question: what must be preserved or corrected before continuation means what it claims to mean?

Contribution and evidence limits

Justin originated the product need and set or accepted its consequential boundaries: external plans, externally owned contracts, distinct attempts, separate review and acceptance, and recovery that preserves useful work. Codex supplied extensive implementation, testing, investigation and operating work. The account does not attribute every source line or review finding to unaided human authorship.

The retained campaign supports a real application of this architecture. The source and focused regression case make its mechanics inspectable. They do not establish universal semantic correctness, industry-wide superiority, or a measured productivity advantage over another orchestrator. The separately dated public-source section adds the standalone recovery demonstration and scoped real completion/stop checks. The historical excerpts themselves were not rerun during this update, and the public checks do not supply a live selective-recovery campaign or a comparative benchmark.