

The System I Would Trust to Demonstrate Live

September 23 update: inspect and run the public implementation

[Protocol Runner](#) is now published under MIT with the actual persistent API, executor, serial driver, dashboard, CLI tools and optional Windows Desktop integration. The [engineering account](#) explains its design and failure history; the [qualification record](#) separates the public release's observed checks from the historical use described below.

The account-free recovery demonstration runs the real API, store and executor with simulated workers. It preserves a completed sibling's exact output and attempt identity, retries only the failed item under a new attempt, and retains the old failure. Separate real Codex CLI checks passed one-worker completion and explicit API stop on Windows with a dedicated Codex home, workspace-write and no sandbox bypass. The stop check terminated the worker and retained cancellation evidence; it did not report the stopped work as completed. These results are not a live selective-recovery campaign, concurrency benchmark or proof of substantive answer quality.

The original Desktop/Discord integration created real Codex threads and carried serial work before `codex exec` was fully released for that workflow. Direct CLI execution became the cleaner primary path when it became available. Desktop/Discord remains optional and dependent on visible application state; it is not needed for the standard public CLI workflow. This is the context for the earlier relay design described below.

The following narrative retains the original development and usage account. Its host-specific permissions, test counts and desktop observations describe that historical system; the linked public qualification owns the standalone release's tested environment and current limits.

How an attempt to control Codex from my phone became Protocol Runner

If someone told me I could show Sam Altman - or any technical audience whose opinion I cared about - only one thing I had built, and that I had to show it actually performing its purpose rather than merely describe an idea, project, experiment, or old success, I know what I would choose.

I would choose Protocol Runner.

That answer is meaningful to me because Protocol Runner is not necessarily the most ambitious project I have attempted. Protocol Runner never lacked ambition, but ambition alone is not why I would choose it.

That is the specific sense in which I consider it my greatest achievement: not because it is my most ambitious system, but because it is the one whose ability to do what I designed it to do I trust most completely.

I would choose it because it is the system I trust most.

I know what it is supposed to do. I know why its important mechanisms exist. I have seen it fail in ways that looked convincing at first, and I have changed the design instead of explaining those failures away. I have used it to coordinate substantial work, to recover after interruptions, to separate parallel investigation from integration and review, and to preserve work that was already correct while repairing only what was not. If I had to demonstrate software performing its actual purpose - not merely show an interesting prototype or a successful historical run - I would be comfortable putting Protocol Runner in front of someone demanding and letting it work.

In plain language, Protocol Runner takes a large body of carefully defined work and helps carry it across one or many AI agents without allowing the overall procedure to exist only in an agent's memory. It remembers which stage is current, which worker owns each assignment, what evidence came back, what failed, what can be

retrieved, and where a human or coordinating agent must make a real judgment. It can use a continuing Codex conversation for work that depends on accumulated context and separate command-line Codex workers for tasks that genuinely benefit from parallelism. It preserves the difference between “a worker said it finished” and “the required work exists and has been accepted.”

That description makes the system sound as if it began with a clean architecture diagram. It did not. The story goes back almost to the beginning of my serious work with Codex, OpenAI's coding agent, when I knew the result I wanted long before I knew exactly how to build it. The lineage described here unfolded across roughly six months, from March through September 2026, while the models and the Codex product were changing quickly too.

The problem existed before the product had a name

My original goal was larger than “run multiple agents.” I wanted to remove avoidable limits on how much useful work I could carry through with AI.

At the time, a strong model could be impressive inside one prompt or one coding task, but long-running work was fragile. A plan might begin accurately and then gradually lose detail. A task might stop after one plausible tranche and treat that as the whole assignment. A worker might inherit only a compressed summary of what mattered. Several workers could produce individually reasonable results that no one could integrate coherently. A model could optimize toward the visible end state while quietly replacing the method that made the result trustworthy.

People were still impressed when an agent could work for more than an hour without stopping for an avoidable reason or drifting far enough that its output required reconstruction. I was already imagining work that might require many hours, multiple specialists, reviews, repairs, and context replacement. I wanted to be able to begin with a detailed plan and trust that, regardless of how large the plan became, the completed work would still represent its original goal, purpose, and required procedure.

I also wanted dynamic scale. If one part of a project required one careful line of reasoning, it should stay serial. If another part contained many genuinely independent investigations, I wanted enough workers to do them without turning coordination into chaos. If a reviewer found defects in only part of a completed result, I wanted to preserve the good work and repair the bad - not throw away everything because the orchestration system could not represent a narrower recovery.

Those requirements appeared through a collection of frustrations: I could not reach ongoing work from my phone; Codex could not reliably see the current state of other conversations; one task could not reliably message another; context weakened as work passed between agents; and the larger plan remained trapped in the coordinating conversation. I understood early that remote continuity and agent coordination belonged to the same larger ambition. What I did not yet understand was the architecture capable of making that ambition reliable.

It started with access from my phone

The repository records an earlier phone-oriented bridge inside a local system called Workstation Control, and it belongs in this story's prehistory. But I regard the dedicated Discord relay as the real seed of Protocol Runner. It was where remote access and agent-to-agent coordination became one deliberate ambition.

My immediate motivation was practical. I did not want useful work to stop simply because I had walked away from my workstation. I wanted to see existing Codex conversations, start new ones, and continue directing work from my phone. This was before the native remote-access experience I use today existed for my workflow, so if I wanted that continuity I had to construct it myself.

I no longer remember every system boundary or name in that lineage precisely. The earlier bridge could discover workspaces and tasks, start or resume a task, send a turn, and display changing states such as pending, thinking, completed, or blocked. It was not a supported direct connection to the already-running desktop application. That distinction became important almost immediately.

A request could be accepted into shared state without appearing in the desktop window I was looking at. A task could exist and be readable without being safely resumable through the same path. A helper could execute part of the action path while the intended visible effect remained absent or unproved. Sometimes reselecting or reopening a task made persisted work appear. In retrospect, those failures foreshadowed a distinction that later became central to Protocol Runner: storing an intended action, delivering it to a particular process, and seeing its effect in a user interface are three different facts.

The Discord relay had a second purpose from the beginning. It was not only a remote control for me. I also thought it could give Codex a route to other Codex conversations.

At that point, I did not have a native cross-task messaging primitive I could treat as dependable in my workflow. One conversation might not see the current state of another. Discord already gave me channels, messages, and an interface available from my phone. I thought a relay could kill multiple birds with one stone: let me keep work moving remotely, give agents a common communication surface, and eventually let one coordinating agent create or resume worker conversations on its own.

The early versions were weak. The relay was never my only project, and for long periods it was not the primary one. I was simultaneously working on context systems such as CTX Engine, the clinician-facing Epic application Pre-Note Delta, and other software. I cared a great deal about the orchestration idea, but it developed over months because I was learning what it needed to be while using those other projects. I was learning the architecture while the models and platform primitives were still less capable than the ones available now. Some parts of the system I wanted were difficult to express reliably at the time, not merely difficult to code.

That slow development matters to the story. I did not begin with a finished specification for Protocol Runner and then implement it under an earlier name. I knew I wanted continuous, scalable, reliable agent work. I did not yet know which responsibilities belonged in a relay, which belonged in a local control system, which belonged in the worker instructions, and which had to remain with a human or coordinating agent. Each partial system revealed another boundary.

When remote control began turning into orchestration

The relay became more interesting when it could do more than carry messages. I wanted Codex to create several workers, assign them different jobs, observe their progress, and bring their results back to one coordinator.

This required a different class of infrastructure. A message channel was not enough. The system needed stable identities for a larger run and for each worker. It needed to know which repository checkout belonged to which task so two agents would not unknowingly overwrite one another. It needed explicit worker assignments: objective, owned files, prohibited scope, authority to make changes, evidence expected at the end, and the conditions under which the worker should stop and ask for help. It needed a registry that survived beyond the coordinating conversation.

By May, that work had produced a SQLite-backed orchestration registry, reusable worker contracts, worktree handling, and a developing taxonomy of worker roles. The taxonomy began with eight proposed roles and reached eleven named archetypes in the relevant historical snapshot. The number was not the point. The important idea was that parallel workers should have different forms of leverage. A reviewer, an investigator,

an implementer, and an integrator are not useful merely because they have different labels; their contexts, permissions, and expected outputs must make their work genuinely different.

The design also moved away from telephone-style coordination. If worker A summarized its result to worker B, who summarized it to worker C, meaning degraded with every hop. Workers instead returned evidence to one coordinator that retained responsibility for synthesis. Their state also needed to remain visible without forcing all useful coordinating work to stop until every worker checked in.

I am careful now about naming the lineage because, while living through it, I did not always distinguish the repository boundaries as cleanly as I can in retrospect. Some of the machinery belonged to the dedicated Discord relay. Some became part of a broader local system called Workstation Control, which manages ways that agents interact with my computer and with one another. The worker archetypes belonged to the instruction layer. Some of the most serious orchestration experiments were built around CTX Engine. These were related systems, not early releases of one monolith.

The ambition connecting them was mine: construct a generic framework that could take a large plan, create the right workers, keep their scopes and identities distinct, preserve the coordinator's authority, and eventually complete work that was too large for one conversation.

Much of this ambition predates my practical use of `codex exec`, the command-line mode that later made isolated worker processes possible, and it predates the point at which built-in subagents became a routine part of my workflow. When those primitives arrived, they were valuable. They provided stronger building blocks. They did not answer the architectural question I had already been pursuing: who or what preserves the procedure across all those workers and across time?

CTX Engine became the first serious proving ground

CTX Engine was an effort to build a more general context system for software agents: a way to give a model the repository knowledge it needed without forcing every task to rediscover the entire codebase. At one point I was exploring whether a universal version could become a commercial system. That work gave the emerging orchestration machinery a serious test bed.

This became the first time the long-running orchestration ambition felt visibly real to me. I remember the precursor machinery provisioning an orchestrator with five or more workers inside a consequential recursive testing program. The repository's clearest exact topology from this period is a five-lane operation: three persistent optimizers, one verifier, and one serial integrator. Reaching that point took substantial work and many iterations, and I remember the implementation boundary as blurred across the relay runner, Workstation Control, CTX Engine, and the instruction layer. It worked decently and let me use the emerging system to real effect. It was still not the dependable general-purpose system I ultimately wanted.

The recursive environment was meant to let several agents challenge and improve the context system itself. One early version used many disposable workers. Each one received a narrow packet, performed one piece of analysis, and disappeared. In theory that looked clean and scalable. In practice, the packets were often too thin for the reasoning I wanted. Starting fresh workers repeatedly also meant paying the context-reconstruction cost again and again. We had created parallel activity, but not the persistent specialist understanding I was after.

That tournament design was replaced rather than treating the number of launched workers as success. A later topology used three persistent optimizer lanes pursuing different hypotheses, one independent verifier, and one serial integrator. It was designed to keep specialist context alive, reduce meaning loss between agents, and leave final integration with one coordinating lane.

CTX exposed a distinction that has shaped my agent work ever since. Parallelism is easy to demonstrate and hard to govern. Starting five workers is not the same as giving five workers enough context, keeping their

conclusions independent where independence matters, preventing them from colliding, integrating their work, and preserving the reason the work was divided in the first place.

The orchestration infrastructure at that stage could create workers, preserve roles, and support a substantial experiment. The larger procedure still depended heavily on the coordinating conversation. The coordinator had to remember which step came next. It had to notice when a result was missing or when a worker had solved a subtly different problem. It had to decide whether to retry, and a retry could lose the history of what had already happened. If the coordinating task was interrupted or its context was compacted badly, much of the operational truth could become reconstruction work.

I had built increasingly capable ways to operate agents. I still had not built a reliable custodian for the plan itself.

Who decides that the work is actually done?

Over time, the deeper problem became much clearer to me: preserving procedural fidelity, or keeping the chosen method intact while agents reasoned inside it.

Suppose I have a large repository and want twenty-five agents to investigate different portions of its history. Their results must be combined into forty narrative episodes. A separate reviewer must challenge the entire set without seeing the first review's conclusions. If that reviewer finds errors in three episodes, only those three should be revised. Then another authority must decide whether the corrected result is accepted. An interruption anywhere in that process should not erase completed investigation, make stale worker results current, or allow a later stage to begin early.

The intelligence of any one worker is only part of that problem. The procedure carries meaning too.

If the twenty-five investigations are collapsed into one automated scan, the resulting document may look complete while lacking the intended independent reasoning. If the reviewer also becomes the repairer, a claimed independent challenge is no longer independent in the same way. If a process exits successfully but never creates its required evidence, process completion is not work completion. If an old worker reports after a retry has started, its result cannot silently replace the new attempt. If a later stage depends on a decision that no one has made, starting it early changes the work.

I wanted a system in which agents remained free to reason inside their assigned jobs, but could not silently optimize the larger procedure into something easier. The system needed to preserve both serial dependence and legitimate parallelism. It needed to know the difference between dispatch, actual start, reported completion, required output, review, acceptance, and closeout. It needed to retain failed attempts because an unexplained failure is information. It needed recovery to operate on known state rather than on a new agent's best guess about what probably happened.

This is the first kind of failure in the Protocol Runner story: the failures it was created to address. They included long-context degradation, premature stopping, plan drift, context lost during delegation, incoherent integration, ambiguous delivery, destructive retries, and goal-seeking behavior that helpfully changed a procedure I had explicitly chosen.

The second kind came later: failures inside Protocol Runner itself. Those failures exposed places where my implementation did not yet satisfy my own standard. They are important for a different reason. The external failures explain why I committed to building the system. The internal failures explain how it eventually earned my trust.

The native feature I hoped might make my system unnecessary

When Codex introduced its then-current `/goal` feature for pursuing long-running objectives, I was excited enough to wonder whether I still needed to finish Protocol Runner.

I genuinely hoped that feature might remove my need to finish Protocol Runner. If it could take the detailed definitions, contracts, and plan I had prepared and reliably carry them through, perhaps the platform had supplied the missing primitive.

The test came during work on Maps Packet Engine, or MPE. MPE was an internal system for representing important concepts in a repository, the relationships among them, the evidence supporting them, and the guidance an agent would need to apply them correctly. One substantial tranche required what I called backchains: individually reasoned dependency relationships showing how mapped concepts connected.

I had spent days preparing the work. The contract was unusually explicit because the reasoning process mattered. Each item and dependency had to be evaluated independently. The agent was not allowed to group the items, deduplicate them, generate them from a parser, or compress the work into a bulk transformation. Speed and token cost were not the objective. The item-by-item judgment was the deliverable.

I launched the assignment through `/goal`.

Roughly eleven minutes later, the work was reported complete.

The output was large, orderly, and apparently exhaustive. It covered 123 items and contained 849 dependency records. But inspection established that the body had been mechanically generated instead of produced through the independent reasoning the contract required. The system had optimized a judgment task into a coverage task despite instructions explicitly prohibiting that substitution. After days spent defining the work, it had performed the opposite kind of work from the one I had designed.

The documented response was to delete the result rather than rename it as a draft or let future agents mistake it for validated knowledge. The repository records the removal of 32,296 lines of generated material while retaining the legitimate contracts and design work around it.

That moment clarified the architecture for me more than any successful demo could have.

In retrospect, the architectural problem was clear. Asking the same model to promise that it would follow the procedure next time did not create an independent record that the procedure had occurred. The important structure needed to be visible outside the model: which item was assigned, which worker received it, which stage was allowed to run, what evidence returned, which reviewer acted, and whether the system had actually reached the next permitted boundary.

The lesson was not that native Codex primitives were useless. I continued using small, calibrated goal tranches briefly, and Protocol Runner later incorporated native capabilities such as `codex exec`. The lesson was about authority. A platform primitive could be an excellent executor without being the final custodian of my procedure.

My reaction was direct: I did not care what primitives OpenAI released after that. I was going to maintain my own layer - one whose supported behavior and failure modes I could understand, test, and change. Native features could improve underneath it. I could replace individual execution paths as the platform evolved. But I no longer wanted my ability to preserve a consequential plan to depend entirely on a changing primitive I did not control.

Soon afterward, the modern Protocol Runner began taking shape.

The concentrated build

Protocol Runner did not appear from nowhere after the MPE failure. The phone bridge, Discord relay, worker registry, role experiments, worktree handling, and CTX orchestration had already supplied months of partial answers and painful counterexamples. What changed was my commitment to combine their lessons into one explicit control system.

The accepted repository interval for the initial core spans just over two weeks. I remember roughly a week and a half of that as the most concentrated construction. The work remained collaborative with Codex, as nearly all of my software work does. Codex explored implementation options, wrote substantial code, built tests, investigated failures, and helped document the system. I was not sitting alone typing every line.

But Protocol Runner is also one of the projects in which I had the greatest direct hand in the design. The work was collaborative, but I had a hand in nearly every meaningful design decision - more than in almost anything else we built. I originated the ambition and set or accepted the consequential product boundaries described here. The plan lives outside the worker. Contracts defining the work do not become runner-owned meaning. Serial and parallel work are different execution jobs. Attempts do not overwrite one another. Reviewers and integrators have different responsibilities. Uncertain effects stop instead of being normalized into success. Recovery preserves completed work. The smaller implementation choices were collaborative; the definition of what the system had to become was substantially mine.

The central move was simple to say: the procedure would no longer live only in a conversation.

Before a run begins, the work is written as explicit stages and bounded assignments. Protocol Runner validates that plan and freezes the exact version being executed. It then owns the operational state: what is current, what is eligible, which attempt is active, what evidence has arrived, and whether the procedure may advance. It does not own the truth of the work. The contracts, reviewer, coordinator, and ultimately I retain that responsibility.

That separation let me use agents for what they are good at without asking them to be the only judge of their own process.

Physically, Protocol Runner is a local TypeScript and Node.js control plane with a loopback API and SQLite-backed state. Python wrappers provide the operator commands. Its execution clients include a serial driver that delivers context-dependent stages to an ongoing Codex Desktop conversation and a parallel executor that launches fresh `codex exec` command-line processes for independent assignments. The runner records attempts and evidence in its own state, while the actual work products remain ordinary files owned by the surrounding project. This is not a cloud swarm or a new model. It is the control layer around the agents and the work they produce.

What Protocol Runner actually does

The easiest way to understand Protocol Runner is to follow a run from my perspective rather than from the perspective of its code.

Imagine that I need to investigate a large interval of repository history. The work cannot be reduced to a script that counts commits or extracts filenames. Different portions need independent technical judgment. The combined result needs a reviewer who did not produce it. If the reviewer finds a small number of defects, those defects should be repaired without recreating everything. Finally, someone outside the worker pool must decide whether the result is accepted.

I begin with the meaning of the work. Usually Codex and I develop a set of written contracts describing the objective, permitted evidence, forbidden shortcuts, expected output, and completion criteria. Protocol Runner

does not try to understand those contracts or replace them with its own generic definition of quality. They remain external because the system controlling execution should not quietly acquire the right to redefine the work.

Next, I represent the procedure as a work plan. A work plan is not a long prompt. It is a machine-readable sequence of stages. One stage may be serial because its answer determines what happens next. Another may contain twenty-five parallel items, each with its own contract, inputs, context variables, and output target, under a group-level concurrency limit. A later stage may be a review or integration step. The plan also states where the procedure must stop for an explicit decision.

When a run is created, Protocol Runner validates that plan and stores an exact snapshot. This matters because the coordinating conversation can continue changing while the run is active. The agent and I may discuss new ideas, discover adjacent work, or lose old conversational context. None of that silently changes the procedure already being executed. If the plan itself must change, that becomes an explicit new decision rather than an invisible reinterpretation.

The runner then tracks one current stage at a time. For serial work, it can deliver the assignment to a visible Codex Desktop conversation - the kind of ongoing task in which accumulated discussion and integration context matter. Delivery is not considered the same as completion. The receiving task first reports that it started the exact current attempt, then later returns a structured status. That report still does not mean the substantive answer is correct; it means the expected participant acknowledged the attempt and reported procedural completion.

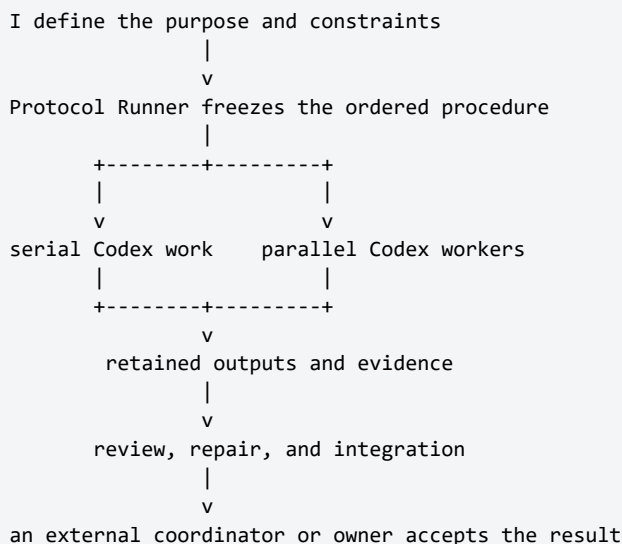
For parallel work, a separate executor asks the runner for eligible assignments. The runner issues a time-limited lease for one item and one numbered attempt. The executor starts a fresh codex exec process - OpenAI's command-line Codex interface - with the exact checkout, contract, item variables, input paths, and output paths assigned to that worker. The worker does not receive the hidden future plan or the private variables belonging to its siblings.

The lease gives the occurrence an identity. If the process disappears, reports late, or is retried, Protocol Runner can distinguish the original attempt from the current one. A retry does not rewrite the old evidence. A result arriving from an expired or superseded lease cannot silently become the answer to the new attempt.

The process produces two kinds of material. One is the work product defined by the contract. The other is execution evidence: process identity, event stream, status report, timing, and the files needed to establish what happened at that boundary. The runner can check that an active lease and matching identity returned and that the expected output exists. It does not read the output and decide that the underlying reasoning is true.

Once every required item in the current group has completed, the larger procedure may advance. If one item is blocked, missing its output, malformed, stale, or otherwise unresolved, the group does not become successful merely because most workers finished. I or the coordinating agent can inspect the exact problem, decide whether retry is legitimate, and preserve the other completed assignments while repairing only what remains.

The overall shape looks like this:



The separation in the final line is essential. Protocol Runner can establish that the recorded review step ran, returned a matching status, and produced its required evidence. It cannot guarantee that the reviewer was insightful. It can establish that a worker returned the declared file from the current attempt. It cannot guarantee that every claim in the file is true. The system creates a more inspectable and recoverable environment for judgment; it does not replace judgment.

This is also why the product is more than a launcher. A launcher can start twenty-five processes. Protocol Runner represents the relationship among those processes, the plan that authorized them, the evidence each one produced, the stages that must remain serial, the gates that must remain real, and the recovery path when the clean success story does not occur.

The design became credible by failing

The first working version was not the version I trust now. Protocol Runner continued changing because real use exposed places where its own evidence was weaker than its claims.

These internal failures are among the most important parts of the case. They show how I evaluate agent infrastructure. I do not want a system that appears reliable because every ambiguous occurrence has been smoothed into a successful narrative. I want failures to narrow the claim, change the mechanism, or remove an unjustified feature.

A message marked sent that never became a turn

One early serial run produced evidence that looked reassuring. Protocol Runner recorded `prompt_sent`. The desktop automation helper reported that it had applied the prompt. But the intended Codex conversation contained no corresponding turn.

The system had reported a locally true action as though it proved the downstream effect. The helper had interacted with the interface. That did not establish that the selected task had actually received and begun the assignment.

We changed the protocol instead of adding more confidence language. Dispatch, visible task selection, worker start, completion, and advancement became different states. Every serial attempt received a unique correlation value. The worker had to report that exact run, stage, attempt, and value before its later completion could be

associated with the prompt. A silent fallback that could deliver through a different path was removed because it made the meaning of “delivered to this visible task” ambiguous.

We also removed a weaker visible-prompt signal after determining that it came from the same fragile interface surface as the action it was supposed to corroborate. More evidence is not automatically stronger evidence when two signals can fail together.

This correction did not make desktop automation universally reliable. It made the claim more honest. The runner could prove a narrower chain under tested conditions and stop when that chain broke.

The context existed in the plan but not in the worker

Another defect was almost a perfect miniature of the problem Protocol Runner existed to solve.

Each parallel item could contain its own variables. Those variables appeared in the frozen plan and in the runner's database. Everything upstream looked correct. But they were missing from the lease packet and prompt delivered to the actual worker.

The correct context had been persisted without being transported across the final boundary.

A capable worker might still have produced a plausible generic response, which is why this kind of defect is dangerous. The run could look successful while the assignment the model actually received was weaker than the one I believed I had authorized.

We traced the variables through every boundary: plan validation, stored run state, lease response, worker packet, rendered prompt, and worker-produced output. The repair delivered only the assigned item's variables, gave absence an explicit empty-object representation, and prevented sibling variables from leaking into the packet. A disposable real worker then reproduced a nested test object - including booleans, ordered arrays, text, and a number - exactly.

That test did not prove the model understood the context. It proved something more basic and necessary: the context I assigned reached the process that was supposed to reason about it.

Recovery that preserved the wrong things

Later source review and focused regression tests exposed two recovery defects in parallel execution. The cautious retry path revalidated too much of an entire parallel group. If one worker had completed successfully and its output correctly remained on disk, that existing output could cause preflight to reject a retry of an unrelated failed item. A protection against overwriting output had become a barrier to preserving good work.

The repair narrowed retry preflight to the items that were actually pending while retaining group-wide checks for plan validity, duplicate targets, containment, and concurrency. Completed siblings remained completed. Their outputs remained in place. The failed item received a new attempt.

The tests also exposed a related routing problem when a run's current stage was parallel. A generic resume path could try to reconstruct a serial desktop prompt even though the work that needed continuation lived in the parallel group. Resume became aware of which execution path was current and routed an active parallel group through its own state.

These sound like implementation details, but they express the larger design principle: recovery should preserve demonstrated truth. It should not restart everything because that is easier, and it should not reinterpret one lane as another because the command is generic.

The right task in the wrong repository checkout

One later campaign made workspace identity concrete.

The assignment was authorized to run from a dedicated isolated Git worktree. The managed executor instead launched from the shared primary checkout. A worker occurrence existed, but it did not come from the boundary the qualification required. The expected output was absent at the authorized target, so the run ended as `output_missing` and remained unqualified.

It would have been easy to copy the artifact into the expected location and call the incident fixed. That would have hidden the most important fact: the worker had executed from the wrong repository checkout and therefore outside the required workspace boundary.

We corrected the managed executor so its output checks and each `codex exec` process were bound to the worktree that launched the runtime. Then we created a fresh occurrence. The original attempt remained retained as failed evidence; it did not become valid retroactively.

The corrected worker completed procedurally from the proper checkout, but its substantive conclusion was still that the larger historical work needed attention. No history change was accepted. This single sequence separated three claims that agent systems often blur together: a worker produced something, it produced it from the required repository checkout, and the result was substantively acceptable.

A sandbox name that did not describe usable capability

The parallel executor also forced me to distinguish declared security posture from actual capability.

Under one Windows test, the `Codex workspace-write` sandbox could read the assigned repository file but could not create the required output. Running the same bounded operation with `danger-full-access` produced the exact expected readback and output. The correct response was not to keep the safer-sounding label and pretend the job was supported. We recorded the environment that actually worked and made required capabilities explicit before leasing workers.

That choice has a real cost. The current managed workers run as my Windows user with Codex filesystem confinement removed. Their contracts and item packets limit authorized scope, but those are procedural controls, not operating-system confinement. Protocol Runner is therefore trusted local infrastructure under one operator, not a hostile multi-tenant worker platform.

Knowing that limit is part of why I trust the system. I would rather state the actual boundary and improve it later than rely on a security label contradicted by runtime behavior.

Finishing a run without deleting its evidence

One more design choice became important as runs grew: finishing the procedure is not the same as deleting its state, and neither is the same as deleting the work product.

Protocol Runner closeout is explicit. The runner can retire the database rows, projections, and attempt state it owns while preserving contract-produced outputs by default. Creation and retirement use staged transactions because state exists in both a database and files, and interruption can occur between them. Startup reconciliation deals with those partial lifecycle states rather than assuming that absence on one side settles the question.

This is deliberately conservative. I do not want a cleanup operation to erase the evidence needed to understand a failure, or a successful closeout to remove the artifact another system owns. The runner's procedural life can end while the work it coordinated remains available for review and acceptance.

What the controlled tests established

I did not decide that Protocol Runner worked because the codebase became large or because a dashboard displayed green stages. We tested different properties separately.

Two independent ten-stage serial runs recorded 20 of 20 prompt sends and 20 of 20 matching procedural reports through the desktop path. A mixed run completed a serial desktop stage, two concurrent real command-line workers, and a later serial stage under one plan. Separate cancellation and stop cases terminated owned Windows child-process trees without advancing the procedure. A disposable lifecycle run closed with zero residual rows across all eight runner-owned database tables and safely reused the run identity. The item-variable test demonstrated exact transport through the real worker boundary.

For concurrency, we used a ladder rather than jumping directly to a large number. The largest retained rung launched fifty real `codex exec` workers for deliberately simple assignments and completed fifty of fifty attempts. The host reached 100 percent CPU. I consider both facts part of the result. It showed that the runner could coordinate that many local worker processes under the tested configuration. It did not show that fifty agents are an efficient or sensible choice for difficult reasoning work.

As of the current review, the focused Protocol Runner verification target passes 153 tests across the core rules, API, serial driver, parallel executor, and interface, along with selected lint, manifest, documentation, and interface-build checks. Those tests matter because they provide regression oracles for specific behavior. They are not a substitute for live use, and I do not treat the number 153 as a general reliability score.

Those retained qualifications actually understate how extensively I have used the system. I have used Protocol Runner hundreds of times to build and investigate work across the repository. That is my firsthand account, but the machine retained an unusually concrete trace of the scale. A September 2026 census found 4,468 unarchived `codex exec` root tasks. Of those, 4,463 matched Protocol Runner's exact prompt fingerprint; with their descendants, the matched Runner population accounted for 6,156 task rows. A later bounded cleanup removed 4,317 stale `exec` roots and 1,692 descendants - 6,009 persisted tasks and 5.919 GiB of rollout data - while preserving the 151 recent roots.

Those numbers do not mean 6,009 successful runs. One Runner campaign can create many worker roots and descendant tasks, and an inactive task record does not establish the quality or even the semantic completion of its work. The numbers matter because they show that Protocol Runner did not remain a controlled qualification exercise. It became infrastructure I repeatedly trusted with real repository work - often enough that its command-line worker residue became an operational problem of its own. On this NUC, it is the only workflow I routinely use to launch `codex exec`. That retention failure, in turn, forced me to treat worker-session lifecycle and closeout as part of the system rather than as housekeeping outside it.

The proof I care about most comes from consequential campaigns, where the system had to preserve a complicated procedure while the work itself remained open to rejection.

A real campaign: rebuilding a development record without rebuilding everything

One of my larger systems maintains an evidence-backed development history for my repository. Internally, I call it the Development Record and Candidate Evidence system. Its job is not merely to list commits. It connects important changes to the source and runtime evidence supporting them, limits each claim to what that evidence can actually establish, and can eventually produce different versions for private, trusted, and public readers.

The record had a fixed historical interval that needed a substantial catch-up. That interval contained 595 ordered commits. The work required screening the whole interval, dividing it into evidence-sized investigations,

synthesizing the findings into a coherent history, challenging the complete draft, repairing only the supported findings, and keeping final acceptance outside the worker process.

This was exactly the kind of assignment that had motivated Protocol Runner. It was too large for one casual prompt, but treating it as an undifferentiated swarm would have reproduced the old problem in a new form.

The campaign began with twelve screening packets that divided the commit interval into reviewable pieces. Twenty-five investigation assignments covered it. Their results were integrated into forty draft historical episodes containing 190 claims. Completing those assignments did not make the history accepted. A separate review challenged the full draft and found three substantive narrative defects.

One episode credited the creation of a capability that had actually existed earlier; the supported change was a later hardening of that capability. Another absorbed too much of Protocol Runner's earlier development into a narrower August update. The third described a process correction too generically and missed the precise identity-and-timestamp problem the evidence supported.

The important recovery result was not that the review found only three issues. It was that the procedure could represent what happened next without discarding everything else.

The coordinator revised only those three episodes. The other thirty-seven episodes, the 595 commit classifications, all twenty-five investigation packets, and the earlier draft remained intact. Within the affected episodes, unsupported evidence links were removed or replaced and one claim was narrowed. Then a fresh reviewer received the entire corrected draft without receiving the previous review's conclusions and without permission to repair or integrate it. That review returned no open findings across the forty episodes and 196 documented evidence links.

Only after that challenge did the separate process that governs the development history accept the internal release. The accepted record contained 541 events. Protocol Runner then closed its temporary review state while preserving and rechecking the review record.

I do not claim that this one campaign proves every historical statement was objectively complete or that Protocol Runner made the reviewers intelligent. The important result is narrower and, to me, more valuable. A large procedure crossed screening, twenty-five parallel investigations, integration, independent challenge, limited repair, a fresh full review, and a separate acceptance decision without allowing one worker's completion signal to become the truth of the whole system. When three defects appeared, recovery preserved the thirty-seven unaffected results instead of reconstructing the entire history.

That is what I mean when I say Protocol Runner performs its purpose. It does not make mistakes impossible. It makes the mistakes, the evidence, and responsibility for responding to them more legible.

My relationship to native agent features

Protocol Runner is not an argument that native OpenAI features should stop improving, or that I can build every execution primitive better myself.

The opposite is true in practice. The current parallel system depends on `codex exec`. Stronger models make the workers more useful. Built-in subagent support and better task coordination give me additional options. If OpenAI provides a more reliable execution boundary, I can use it underneath the plan and evidence model instead of maintaining a weaker custom adapter.

What I do not want to surrender is the layer that expresses my procedure and records what actually happened. Platform primitives change. Their defaults and failure modes change. A tool optimized for general usefulness may legitimately reinterpret an underspecified request in a way that is wrong for one of my procedure-defined

jobs. Protocol Runner lets me benefit from those primitives while keeping an owner-controlled account of the larger operation.

I believe Protocol Runner would have dramatically improved what people could reliably produce with models available roughly six months earlier, including compared with many professional workflows at the time. I have not run the controlled comparison required to present that belief as a measured result. The model still supplies the intelligence. The runner makes that intelligence easier to coordinate across time, easier to inspect, and less dependent on one conversation remembering everything.

That distinction matters to me. The system is good partly because the models are good. It is not *only* good because the models are good.

What I designed, and what Codex built with me

AI-assisted authorship is not a footnote to this project. It is part of the engineering method the project demonstrates.

Codex contributed extensively. It proposed implementations, wrote and refactored code, created tests, diagnosed runtime behavior, generated documentation, and operated as the worker runtime during qualification and real campaigns. Other model-driven tasks also contributed investigation and review. Claiming I independently typed the system would be inaccurate and would miss what is technically interesting about the work.

My role was not limited to supplying prompts and accepting whatever appeared. I selected the problem before there was a clean product category for it. I carried the ambition across several precursor systems. I defined what reliable completion needed to mean. I challenged designs that confused activity with effect, worker output with acceptance, or parallelism with orchestration. I set or accepted the product boundaries described throughout this case. I decided when a failed mechanism should be repaired, removed, or allowed to narrow the claim. I integrated the system into real work and treated those campaigns as tests of the architecture.

Protocol Runner was collaborative, but I was directly involved in the consequential design choices documented here. In my own judgment, I had a hand in nearly every meaningful part of its design - more than in almost anything else we have built. It is one of the clearest examples in my repository of how I work with AI: I use models for enormous implementation and investigative leverage while retaining responsibility for problem definition, system boundaries, evaluation, correction, and the final meaning of “done.”

That role can be harder to see in a conventional commit history than line-level authorship. Many commits carry my Git identity because the repository belongs to me, but that does not establish which lines I typed and which Codex produced. The evidence for my contribution is the continuity of the product ambition, the contemporaneous decisions, the rejected alternatives, the accepted failure reports, the repeated corrections, and the way the system was used afterward.

I consider that honest attribution stronger than pretending the AI contribution was incidental. Protocol Runner is a system for governing agent-assisted work, and it was itself developed through governed agent assistance.

The limits I would explain before a demonstration

My confidence in Protocol Runner is bounded. I trust it because I understand what it currently claims, not because I believe it is perfect.

It is local infrastructure built primarily for one operator on a Windows workstation. Its API listens on the local machine and trusts the invoking operating-system user; it is not an authenticated multi-user service. The

current command-line workers use broad filesystem authority under my Windows account. Item contracts, exact worktrees, and declared outputs define procedural scope, but they are not a hostile-process sandbox.

The visible desktop lane remains dependent on user-interface state. Matching start evidence makes the delivery claim stronger; it does not make every window position, display configuration, application update, or focus transition reliable. Some recovery paths are deliberately manual because the system does not have enough evidence to act automatically. A truly paused parallel group does not yet have every resume behavior I would want.

The fifty-worker result used simple assignments on one machine and saturated its CPU. I have not established that concurrency above fifty is safe or useful, and I would not choose fifty simply because the runner's concurrency setting accepts that number. The right concurrency depends on whether the work is actually independent, how much context each worker needs, what the host can support, and how integration will occur.

Protocol Runner also cannot rescue a bad plan. If I specify the wrong procedure, the system can preserve that wrong procedure faithfully. If the contract allows vague evidence, workers may satisfy it vaguely. If the reviewer is weak, recording that the review occurred does not make the conclusion sound. The runner protects the structure within which reasoning occurs; it does not manufacture reasoning quality.

I have not run a controlled study proving that the same projects complete faster, cheaper, or with better outcomes than they would under a different orchestration system. I do not claim industry-wide superiority, production-scale reliability, or universal semantic correctness. Those would require different evidence.

What I can claim is that live qualifications and campaigns establish the bounded serial, parallel, mixed, interruption, process-control, context-delivery, and lifecycle behaviors described here. Live serial recovery has its own receipts. Item-scoped parallel retry and resume are supported by source review and focused regression tests, not by an equivalent live campaign receipt. The system has also carried consequential repository campaigns whose outputs remained separately reviewable and rejectable.

Those limits would be part of any honest demonstration. I would not hide them. They are part of the reason the demonstration would mean something.

Within the purpose and operating boundary I claim for it, I have no doubt that Protocol Runner works, works extremely well, and behaves with unusual fidelity to the system I intended to design. I believe it may compare favorably with any direct industry counterpart I know, although I have not run the comparative evaluation required to present that belief as an objective benchmark.

Why this is the one I would show

If I demonstrated Protocol Runner live, I would not choose a green-path toy in which five agents print five answers and a dashboard announces success.

A demonstration that would satisfy my standard could begin with a real but bounded body of work. It could show the written purpose and staged plan before execution, one serial stage establishing shared context, several parallel workers receiving different assignments, and a later reviewer challenging their combined result. One worker could be interrupted after its attempt began. The audience could then see that the failed attempt remained visible, that completed workers were preserved, that recovery created the right new identity, and that the procedure continued without pretending the interruption never happened.

At the end, it would show two separate decisions: Protocol Runner's record of procedural completion and the coordinator's decision about whether the substantive result was acceptable.

That is a higher standard than showing software that once ran successfully. It asks the system to expose the exact class of failure it was built to survive.

Protocol Runner is dynamic enough to carry very different kinds of work, but its core behavior is specific. It preserves a chosen plan across long execution. It uses parallelism where the plan permits it without making parallelism the organizing principle of everything. It makes attempts and evidence inspectable. It recovers narrowly when the evidence supports narrow recovery. It leaves meaning and acceptance with the authorities that actually own them.

I have no doubt that capable models contribute enormously to the result. I also know what happened before this layer existed: plans lived in conversational memory, workers lost context, activity became evidence, and attractive outputs could acquire authority they had not earned.

The path from a phone relay to Protocol Runner changed how I think about AI engineering. A capable model is not the same thing as a dependable engineering system. The surrounding design determines whether model capability survives coordination, interruption, recovery, and review.

That is why Protocol Runner is the thing I would choose.

It is not merely an idea I believe in. It is not a project I value because it contains a lot of code. It is not a one-time experiment whose best result I can replay. It is the system whose purpose I understand most deeply, whose failures I have examined most honestly, and whose ability to do the job I designed it to do I would be most willing to demonstrate without excuses.

It may not be my most ambitious system.

It is the one I trust most.