

Building with AI, and building the systems around it

Justin Sublette · AI Systems Engineer - Agents, Context, Execution and Evaluation · Updated September 23, 2026

Open source: [Mathematical Research](#) · [Protocol Runner](#)

I build systems that help AI agents carry complex technical work from an initial objective through investigation, implementation, evaluation and operation. My work spans practical products, research environments and the development infrastructure supporting them, with particular depth in context, persistent execution and agent evaluation. Working extensively with AI, I own problem definition, architecture, integration, experimental direction and acceptance of results.

Two kinds of engineering have developed together throughout that work. One concerns the application or research capability I want to create. The other concerns how people and AI agents can actually build it: what information an agent can obtain, what tools it can operate, how work survives interruptions, and how we determine whether the result is useful. Building substantial projects has given those questions concrete consequences. Improving the development process has made it possible to attempt more demanding projects.

I started in June 2025 with no formal software education, no technical work experience and no practical programming or Git knowledge. Much of my early work used ChatGPT's browser models. The repository preserves a remarkably literal starting point: the first commit contained a 110-line strategy fragment in a file without an extension. Early code assumed other components existed when they did not. I was learning how generated pieces related to an actual software system, including what had to be implemented, connected, deployed and checked before an apparent solution could work.

The initial ambition was automated trading. That brought me into contact with broker interfaces, strategy execution, risk and position handling, persistence and cloud deployment. By July, the repository contained explicit boundaries between those responsibilities and deterministic tests of selected trading behavior. A retained application log dated August 28, 2025 records an order progressing through broker-reported executions to a complete fill and matching portfolio update in the account I was using for paper trading. About ten weeks after the first commit, I had evidence of a real integration effect. It was a meaningful early milestone within a system that still had substantial defects; it did not establish a profitable or fully reliable trading operation.

As the work expanded, keeping an AI collaborator adequately informed became an engineering problem of its own. Long browser conversations accumulated useful understanding alongside stale assumptions. Important decisions and current implementation could become separated. I began developing ways to preserve and retrieve project knowledge outside the conversation. Over time, that work extended into source access, context construction, task orchestration and evaluation. My responsibilities increasingly concerned the relationship among the objective, the information available to the agent, the actions it could take and the evidence left afterward.

Pre-Note Delta brought these concerns together around a product idea: help a returning clinician see what changed in a patient's chart since that clinician last saw them. That question gave the application its organizing principle. It needed to identify the relevant earlier encounter, collect intervening activity across the chart and present an account whose sources the clinician could inspect. I built it as an Epic-integrated application using SMART on FHIR, the authorization and data standards used by applications inside electronic health records.

The core contained twenty deterministic summary sections with source evidence, explicit empty and error states, and clinician-readable output. In a sixteen-patient sandbox cohort, all 3,599 evidence references loaded, establishing a bounded result for integration and source inspection. Optional AI drafting operated on that evidence, with source inspection and clinician review; it did not write autonomously into the chart. This division mattered to the product. Establishing which events belonged in the comparison was an application responsibility. Producing useful prose was a separate capability to evaluate. The clinician retained the medical decision.

Building the application also required building access to its technical environment. The assistants I was using could not directly work through my authenticated Epic developer browser. I developed tooling that progressed from extracting a difficult API-selection interface to authenticated browser capture and a searchable local documentation library. An April 2026 collection retained 191 of 192 selected API documents, alongside guidance

and broader reference material. That gave coding agents concrete vendor contracts to work against while making missing captures visible.

Testing required another layer. I developed harnesses for Epic's browser and desktop environments, combining scripted navigation with an optional AI pilot using screenshots, interface state and bounded actions. Routine navigation became deterministic; model-driven control served diagnosis and recovery. When shared replay recordings made the patient provenance of an initial test corpus unreliable, I introduced isolated evidence collection, stable identities, content hashes and observed browser requests. A test needed trustworthy inputs before its output could say anything useful about the application.

I also evaluated the AI inside the product. A 96-case matrix distinguished functioning output streams, adequate source inputs and faithful generated prose. Many cases were limited by their inputs, and source-level investigation exposed a generation pipeline that could time out or fail to use available detail. We replaced it with bounded packets of source material sent directly to the drafting model. A smaller diagnostic rerun captured all twelve outputs and showed additional source detail in some drafts, while other drafts remained effectively summary-only or lacked enough source material. That rerun supplied specific observations about the revision, without establishing quality across the full matrix. Those distinctions guided further work without turning successful execution into an automatic quality claim.

Deployment and commercial preparation extended through AWS customer-environment templates, private application and data services, container deployment, migrations, signed sessions, audit retention and hospital review and support materials. Demonstration deployment evidence showed healthy web and API services and an audit event reaching durable storage. Pre-Note did not achieve a hospital rollout or commercial success. I had developed the concept without knowing a comparable Epic capability was coming; seeing the native offering address much the same workflow reinforced my confidence in the product insight. I eventually decided that pursuing the remaining opportunity against the platform owner was not the best use of my effort, and retired the system in July 2026.

Life Links explored a different product need: giving an AI assistant access to selected, persistent context about a person's physical world. I designed records for places, containers, objects and equipment, with physical location kept distinct from collections organized around an activity. Later additions included attachments, routines and calendars. A future plan needed to remain editable without rewriting the record of an activity already completed. An external calendar needed to remain authoritative for its own events. These were product decisions expressed through the data model and application behavior.

The application supported both a human interface and agent access through shared operations. OAuth-authorized MCP access let compatible assistants work with permitted records across conversations while the site was closed; MCP is the protocol through which an assistant discovers and calls a service's tools. Deployed tests used Codex and ordinary ChatGPT as actual clients, exercising creation, retrieval, editing, deletion confirmation, replay and revocation. Separate calendar tests exercised real provider round trips with designated test accounts. Defects in metadata handling, interpretation after writes and time-zone preservation were repaired before the affected checks passed. A matched browser benchmark also showed fewer startup requests and faster initial content after a Collection-loading redesign. The application received a public MIT-licensed release and was submitted to the OpenAI WebMCP Challenge.

Work like this kept raising a broader question: what makes additional information useful to an agent? CTX Engine was my most sustained investigation of that problem. It combined keyword and semantic search with dependency analysis to provide coding assistants with relevant sources and a reading guide. Version-aware indexing and synchronization addressed changing files; recovery after a failed rebuild preserved the prior searchable data while identifying it as out of date. In an internal thirty-task navigation benchmark, the combined approach found 69% of the reference set of relevant files, compared with 34% for keyword search, with 85% precision.

The harder question was whether the agent produced a better answer. I developed comparative evaluations against ordinary repository search, separating file-finding accuracy, answer quality, verification and resource measurements. Across forty-six internal tasks, the standard configuration recorded nine wins, one loss and thirty-six ties or mixed results against repository search under coarse answer scoring. Reference answers were protected from trial agents. Holdout conditions fixed the source, model, prompt and scoring. When two attempts

were invalidated by execution or output-contract problems, we repaired the conditions and reran them. An experimental replacement that looked promising during development did not demonstrate better answers on twenty-four held-out tasks from the same repository. I retained the standard configuration.

The investigation also reached into the evaluator itself. In a broader comparison, an experimental format for presenting source evidence led agents to report opening more relevant material without improving the scored answers. Several explanations remained possible: failure to use the evidence, extra information with little task value, or scoring too coarse to reveal meaningful differences. We added diagnostics separating evidence exposed by the system, files opened by the agent and the sequence in which useful sources were reached. Missing timestamps and token records remained missing; new instrumentation could not retrospectively supply them. This work developed my ability to investigate where a promising local improvement stops translating into a better outcome. CTX is now retained as a legacy system; its context infrastructure, bounded results and evaluation work remain substantial parts of my technical experience.

Context was only one requirement for longer work. Protocol Runner addressed recurring failures in multi-stage plans, handoffs and completion reporting. I designed it to retain plans and execution state outside conversations, deliver the appropriate step and choose subsequent actions within the plan. Work that depended on accumulated conversational context could stay with the same agent conversation; independent assignments could use separate workers. Judgment of work quality remained with the coordinating agent under my direction. When an investigation ran in the wrong checkout, the repair bound worker execution to the launching worktree. A fresh run then completed the assignment. Sustained use made these operational distinctions consequential: a retained census identified more than 4,400 worker tasks matching Runner's prompt signature.

I have now released the [real Protocol Runner implementation](#) under MIT, with an account-free recovery demonstration and [public qualification evidence](#). The demonstration uses simulated workers through the actual API and executor; separate real Codex CLI checks passed completion and explicit stop. The Desktop/Discord path reflects an earlier execution constraint: I built it to create real Codex threads and carry serial work before codex exec was fully released for that workflow. Once available, direct CLI execution became the cleaner primary path. The durable procedure and recovery model remained useful as that execution mechanism changed.

Workstation Control supplied practical operating capabilities across my Windows machines, workers, browsers and local services. Agents needed to discover available operations, deliver configuration, reach the intended session and inspect what actually happened. A responding browser port did not prove that the browser belonged to the task. A prompt persisted in conversation storage did not prove it had appeared in the open desktop application. Those findings led to explicit browser ownership and a separate desktop delivery path. Related repository-wide tool work made native entrypoints explain their commands, selectors, effects and errors through a non-mutating discovery operation. A closing audit covered 81 stable tools without interface findings or probe failures, providing a checked standard for how agents discover and call tools.

Repo Reader supplied direct, authenticated access to current source files and Git history. I kept repository inspection distinct from CTX's context recommendations: an agent could browse, search and read permitted source, while historical references resolved to immutable commits. The implementation included bounded Git operations, durable OAuth authorization, atomic persistence, refresh-token replay handling and large-file reads. Tests targeted restart, concurrency and authorization failure cases; continued authenticated access through a real ChatGPT client still required separate verification. Across these tools, the goal was to give an agent useful capabilities with understandable behavior and enough evidence to diagnose a failed interaction.

The mathematical research system applied related ideas to a more open-ended objective: pursuing a checkable proof or disproof of the Riemann Hypothesis with autonomous AI. Here the research executive is an AI agent responsible for choosing and revising mathematical strategies. My role is engineering the research system, not routinely adjudicating its mathematics. I separated the enduring objective, current strategy, exploratory branches, accumulated understanding, evidence and proposed results so that a successor could reconstruct the research position without depending on one permanent conversation.

This demanded care about what survived a failed approach. A counterexample to one method could coexist with a useful conditional result. An unsupported prerequisite could not silently support a later theorem. The work included dependency review and adversarial falsification, as well as mechanisms for preserving and reviewing

proposed results. In one campaign, an audit rejected an overbroad obstruction claim while retaining a narrower local lemma. Separately, three fresh remote replicas of a frozen mathematical-software suite produced matching substantive results, and an isolated staging rehearsal exercised retained-data continuity without model-provider calls. These are bounded research-process and software results. RH remains open. At the September 16 checkpoint, preparation of tests using retained research data still failed, and the newer software had not yet been fully verified or returned to autonomous research operation.

The September 23 [open-source Mathematical Research release](#) makes that engineering directly inspectable and independently runnable. A separate Linux installation passed real research interaction and stop/restart with fresh public state. [Four worked examples](#) connect research episodes to checkable mathematics and the durable store. One includes a live successor retrieving its predecessor's exact records, preserving a corrected assumption and tightening the retained localization bound from 10 to less than 9. The baseline proof was available as reference material; the other three examples are checked, authored reconstructions rather than additional live discoveries. The [engineering narrative](#) explains the design choices and a real sandbox failure and repair. These results add concrete public evidence without rewriting the earlier private restoration frontier or claiming that RH is solved.

I built the Development Record and Candidate Evidence system, DRCE, to make this body of work inspectable. It connects an evidence-grounded engineering history to versioned releases, readable documents, structured records and interfaces for people and AI agents. Claims retain routes to supporting evidence. Selected hosted releases are separated from the private development repository. Hosted agent access was qualified with synthetic material; publication of the career record remains pending. A historical update using Runner investigated 595 commits and produced forty accounts of significant developments; separate AI review identified three requiring correction, which were revised while the other thirty-seven were preserved. DRCE then retained and released the accepted internal record. The history itself became a practical application of coordinated investigation, review, selective correction and reproducible delivery.

Across these projects, my work has involved TypeScript and Python applications, databases, authentication, browser and desktop automation, cloud deployment, retrieval, agent orchestration and experimental infrastructure. AI agents contributed extensively to implementation, investigation and testing. My responsibility was to connect those contributions to the objective: establish what mattered, direct the architecture, investigate failures, integrate the parts and decide what the evidence justified accepting. The resulting work gives concrete examples through which those responsibilities can be examined.

My development is visible in the changing questions I could ask and the consequences I could handle. Early on, I was trying to make generated components run together. Later work required reasoning about source identity, interrupted execution, privacy, deployment, experiment validity and whether added machinery was earning its cost. The models improved during that period too. The record cannot isolate a percentage of progress attributable to my learning, but it can show decisions changing, defects producing concrete repairs and the scope of my responsibility expanding.

I use full-stack AI engineering to describe that ability to connect an objective to the information, model behavior, software, operating environment and evaluation needed to pursue it. My deepest experience is in the systems around existing models. I have not yet built or trained an ML model, and hands-on training is a direction I want to develop. My product experience includes deployment and commercial preparation, with less evidence of broad external adoption or sustained commercial operation. Those boundaries help define where I can contribute now and where further experience would add depth.

The work I want to continue concerns agents that can carry useful responsibilities over longer and more demanding tasks: obtaining the right evidence, acting through suitable tools, retaining understanding and learning from tests that reflect the intended outcome. Building applications gives me concrete reasons to improve those capabilities. Engineering the surrounding systems gives me ways to make that improvement explicit, testable and usable.