

Pre-Note Delta: building the clinical product and the tools needed to build it

Justin Sublette · Technical and product case study · 2026

A useful question hidden inside a large chart

Pre-Note Delta began with a question a returning clinician should be able to answer quickly: **what changed with this patient since I last saw them?** An electronic health record contains the underlying information, but finding it can require reconstructing a story across visits, laboratory results, medication orders, specialist reports and other parts of the chart. The organizing principle I wanted was the clinician's point of return to the patient's care.

An illustrative case makes the distinction concrete. A clinician last saw a patient six months ago. Since then, the patient has had a hospital admission, new medication orders, laboratory results and a specialist visit. The useful preparation view brings those events together, states where the comparison starts and lets the clinician inspect the source of each item. Another clinician might need a different starting point for the same patient.

I created Pre-Note Delta as a clinician-launched application integrated with Epic, a widely used electronic health record. Its core was an evidence-linked account of intervening chart activity, with optional AI-assisted drafting and clinician review. It was read-only: it did not independently place orders or write a note into the patient's chart. A clinician could inspect the evidence and copy preparation text into their own workflow.

The development became much broader than the visible application. I had to make authenticated Epic developer information accessible to coding agents, build ways to operate and observe browser and desktop workflows, evaluate AI-generated clinical prose, and construct a credible commercial deployment path. I led the product, architecture, integration and acceptance work with extensive AI assistance in implementation, investigation and testing. The engineering described here was work I directed and integrated; it was not an unaided hand-written codebase.

The repository records a formal product thesis in January 2026, an executable foundation later that month, a richer clinical application in February, and substantial live sandbox, evaluation and deployment work through the spring. I ultimately chose not to pursue the remaining commercial opportunity against Epic's native offering. The system was retired into historical reference in July. That commercial outcome is part of the case, alongside the substantial software and tooling it produced.

Making the product's meaning survive integration

The phrase “since I last saw this patient” sounds simple until it becomes a data contract. The application has to establish who the clinician is, identify a relevant prior encounter, exclude the current encounter from that historical search, and choose a defensible time boundary. Epic launch context and FHIR resources can represent the clinician through different identities and relationships. A field that works for one sandbox scenario may be insufficient for another.

I built around SMART on FHIR: the authorization and clinical-data standards that allow an application to launch with electronic-record context and retrieve permitted resources. The implementation normalized clinician and practitioner-role references, used bounded identity expansion and examined additional visit context when necessary. Ambiguous context had to remain visible rather than becoming a confident match simply because one candidate appeared first.

The clinician's identity selected the baseline; it did not filter out subsequent work by other clinicians. Once the starting encounter was resolved, the summary gathered patient activity since that time. Otherwise, the tool could omit precisely the specialist visit or hospital event the returning clinician needed to know about. I recorded that distinction explicitly because it was easy for development or testing to compress the product into the wrong behavior.

The baseline model later supported manually selected historical horizons and the current admission. Those were separate meanings, not interchangeable ways to obtain more rows. A manual horizon sought an appropriate encounter at or before the selected date. Admission mode relied on the current visit context. The interface disclosed fallback or missing-baseline conditions so that a clinician could understand what the displayed interval represented.

The implementation primarily retrieved and normalized events after that boundary. It did not maintain two complete chart snapshots and calculate a universal semantic difference between them. For example, medication output represented supported medication activity since the baseline; it was not a claim to have reconciled every possible medication-state change. That distinction made the product promise more precise and the tests more useful.

Building a clinical evidence application

The early product separated web, API and shared core responsibilities. It grew from seven summary sections to twenty, spanning common review domains such as encounters, medications, laboratory results, imaging, procedures and problems, alongside less uniformly populated domains such as care plans, goals and devices. The point of the fixed section model was consistent meaning: each section had an expected output shape, evidence references and a way to report its own limitations.

I wanted a useful summary to remain challengeable. A compact item could lead to normalized evidence, a source document or a more detailed encounter, procedure, laboratory or report view. Where related resources were assembled into a review packet, the application followed explicit clinical-data relationships. Two records occurring on the same day were not enough to invent an association between them.

External integration required more than implementing the nominal API. Epic's advertised capabilities, supported search parameters and actual responses did not always line up neatly. The FHIR client needed bounded pagination, timeouts, retry behavior and resource-specific compatibility handling. Replay identified requests consistently so a captured interaction could be reproduced without requiring a live sandbox for every development step.

One useful example was care-plan handling. A successful response containing no usable care plans was being treated as a retrieval failure. The correction distinguished an empty result from an unsupported query or a failed request, while retaining a positive case to ensure the repair did not suppress actual data. These distinctions mattered directly to the clinician: “there were no matching rows” and “this part of the chart could not be evaluated” are different statements.

Performance work preserved the same model. Logical section work could run concurrently while outbound requests remained constrained by a shared limit. Progressive delivery showed the baseline and section structure, then filled sections as their results arrived. Request reuse and bounded evidence warmup reduced unnecessary repeated work. These were concrete responsiveness mechanisms; I do not have a controlled before-and-after latency result for the entire application.

The later evidence-navigation test was substantial. A selected sixteen-patient Epic sandbox cohort exercised all twenty section slots per patient and loaded all 3,599 expected evidence references. It contained 203 positive section instances and 117 truthful empty instances, with no failed section slots in the final proof. Eighteen

sections had positive examples in at least five patients; care-team and consent/code-status coverage remained thinner. This demonstrated a functioning sandbox application and its evidence paths, while leaving broader clinical validation as a separate undertaking.

Giving coding agents access to the real vendor environment

A coding agent cannot implement an integration well if the important specifications are trapped in an authenticated browser it cannot use. That was a practical development constraint for me. My Epic developer account exposed selected API information, rendered guidance and other material that was not conveniently available as a clean public documentation package. Native browser access was not available to my coding agents in the way the work required, so I built the missing access path.

The progression began with a concrete interface problem. The Epic API-selection list did not copy and paste reliably. A February snapshot documented extraction from the browser's document structure so the selected APIs could become an inspectable input. It was a small tool serving a material purpose: the agent needed to know what the actual developer environment offered.

The tooling then grew into authenticated browser capture. A crawler could attach to an existing signed-in Edge session through the browser debugging protocol, capture the current page without forcing a reload, and traverse a bounded set of relevant pages. That preserved access to information already available through my session while giving the coding workflow local material it could search and read. This was controlled developer-document capture, not a bypass of Epic authentication.

By April, the documentation library covered selected API payloads, rendered guidance and a broader official reference corpus. It retained original responses alongside readable representations and a record of what had or had not been captured. One recorded collection included 191 of 192 selected API documents, six guidance pages and 59 of 60 broader reference pages. The remaining gaps stayed explicit.

This work improved the inputs to implementation. Instead of asking an agent to infer Epic behavior from a generic FHIR description or from a few manually pasted excerpts, I could give it the relevant vendor contract, examples and observed constraints. The application and its development environment advanced together: better access made more precise implementation possible, and implementation failures identified which information the agents still needed.

Building automation that could be tested

The next challenge was getting through the real launch workflow. A mocked browser test could exercise my interface without proving that the application launched correctly through Epic. LaunchPad supplied a browser-based sandbox route. Hyperspace introduced a stateful desktop application, patient selection, integration setup and a handoff into the browser. Each layer could appear successful while the next layer was using stale or incorrect state.

I developed browser and desktop harnesses around those realities. The work included a Windows virtual-machine route and a desktop-lab route, browser-session attachment, screenshots, UI state inspection and shared application validation. It also included a custom AI desktop pilot: a model could inspect a screenshot and structured state, classify the situation, and choose among a bounded set of actions. The controller imposed step and confidence limits and retained action evidence, with fallback when the model could not make progress.

That model was acting as a test operator, distinct from the AI that could draft a clinical summary inside the product. Its ability to choose an action was not sufficient proof that the application had reached the intended state. The surrounding harness separately attached to the launched app, observed requests and validated the

resulting workflow. Unit tests for the controller used mocked model transport; live product-model evaluations answered a different question.

Broad execution changed where AI belonged. Flexible model-driven navigation helped with unfamiliar states and recovery, but deterministic desktop navigation became the default for bulk collection. Integration setup became a cohort-level preparation step rather than something rediscovered on every patient. Transient failures received bounded retries; persistent failures stayed attached to their own cases so that one bad transition did not erase the rest of a run.

The harness also became more faithful to the user experience. It observed the page's real summary stream, copy request and AI-note request instead of calling convenient API endpoints and treating those responses as browser success. Later work handled requests that started in the background before the test opened an AI tab, avoiding false missing-request verdicts and duplicate generation attempts. The question became whether the actual user path had worked.

Learning to trust the test corpus

Scaling testing required a reliable population and reliable evidence. The sandbox roster could not depend on a display-name heuristic that silently lost unusual labels or counted the same identity twice. A dotted label exposed a parsing defect; stable launcher identities replaced weaker matching. The resulting discovery roster contained 1,747 unique identities, with display variants merged rather than treated as extra patients.

A more serious problem appeared in the captured corpus. Shared replay recordings could be inherited or misattributed, so a patient-labeled result did not necessarily prove that its underlying recording belonged to that patient. Such a corpus would be a poor foundation for evaluating either the application or AI output. I froze the suspect collection and built a stricter recollection route instead of silently rewriting the old evidence into apparent correctness.

The replacement required recordings local to each run, stable patient identity, complete content hashes, per-test records and requests actually observed in the page. Its April ledger recorded 1,733 successful captures and eleven failures. That ledger contained 1,744 entries; the larger discovery roster and the recollection ledger were different populations. A capture with no usable baseline could be valid evidence of that condition while still being unsuitable for a later comparison requiring an anchor.

Realism testing then used eligible evidence to construct controlled scenarios. One early idea tried to target an arbitrary older encounter. A live canary showed that the real automatic baseline instead selected the provider's latest qualifying encounter. I removed the incompatible assumption and narrowed the seed model to behavior the application could actually produce.

This separated several outcomes that otherwise look like a single green test: the deterministic encounter matched, the AI run completed, the capture was complete, and the generated text was good. The larger corpus supported discovery and test selection; it did not mean thousands of AI outputs had received a completed clinical-quality assessment. Broader automated judging remained unfinished. That distinction let the corpus be useful without making it evidence for a result it had not measured.

Evaluating AI inside the product

The optional drafting feature introduced another engineering problem: readable prose had to stay connected to what the application actually knew. The deterministic summary remained the factual starting point. The browser submitted the displayed summary with a session-bound attestation, and the server verified that input before requesting generation. Selected note and report material could add context through a bounded source packet. The clinician still reviewed the result.

I built a matrix that crossed eight scenario profiles, six baseline choices and two drafting modes: 96 captured cases. The evaluation separated route and interface behavior from input adequacy and content quality. It examined whether the prose respected the available facts, chronology, baseline and requested form, instead of treating a completed response as sufficient.

The first quality audit classified 53 cases as passing, 42 as input-limited and one as a major issue because the output was corrupted. The sparse cases mattered: a thin sandbox history is not automatically a model defect. Equally, a stream that finishes successfully can contain unusable text. These classifications came from the project's internal assessment of retained inputs and outputs. The surviving records do not identify whether a person or model assigned the individual quality and source-enrichment verdicts; they do not establish independent clinician validation.

Source-aware generation exposed a deeper architectural issue. An intermediate digest was supposed to make richer notes useful to the final draft. Diagnostics showed that source text reached the pipeline, yet the final stage could fail to use it. Large cases exhausted the digest deadline; other cases underused a valid digest. In twelve representative cases, the first approach produced ten complete captures, two partial captures and no clearly enriched final results.

The repair replaced the digest-and-merge arrangement with bounded direct source packets. The revised twelve-case run completed every capture: four showed source-enriched prose, six remained effectively summary-only and two lacked sufficient source material. The result was valuable because it tied a change in architecture to an observed failure mode. It did not establish a universal improvement across all clinical cases.

Subsequent browser-faithful revalidation passed eight behavior scenarios, and the earlier corrupted output did not reproduce in its fresh rerun. The full 96-case quality matrix was not rescored at that step. I kept those scopes separate: successful later behavior and a promising source-path repair were earned results, while broader prose-quality requalification remained its own work.

Building the deployment a hospital would need

Commercial deployment imposed requirements that a local demonstration could ignore. A hospital would need to understand where the application ran, how it reached Epic, where credentials and clinical data traveled, who could support it and what evidence survived an incident. I built that deployment and operating model alongside the product.

Early Railway staging supplied a place to exercise service health, storage, migrations and rollback procedures. The customer design moved to dedicated AWS environments, with separate accounts for customer nonproduction and production. Terraform composed networking, container services, image repositories, database and Redis services, secrets, ingress and audit infrastructure. The tooling included account provisioning and repeatable environment operations rather than assuming someone would recreate the stack by hand.

The public web entry passed through a load balancer and web application firewall. Application tasks and data services remained on private networking; the separate API service and database were not offered as public endpoints. Database and Redis protections, encrypted secrets and customer-specific configuration were part of the implemented templates. Fixed-egress and site-to-site VPN variants addressed different hospital network arrangements, although the retained evidence does not establish a live hospital VPN installation.

The rollout sequence separated provisioning, image deployment, database migration and service activation. Application controls included signed sessions, constrained read-only SMART scopes and explicit origin checks on selected authenticated browser operations. Optional AI required authorization for the particular customer and environment as well as feature enablement. When enabled, that provider path could carry clinical information; it belonged in the customer data-flow and approval discussion.

Operational evidence had its own design. PHI-free audit events flowed through AWS logging and delivery services into an encrypted, customer-isolated S3 archive. Deployment and proof documents had a separate archive path. Versioning, access restrictions and six-year Governance Object Lock retention supported retrieval of the evidence needed to explain a rollout or investigate a problem. Patient-bearing support exports were not ordinary operational logs.

This reached observed infrastructure operation. April vendor demonstration receipts recorded stable services with two web tasks and two API tasks, healthy public responses, and readback of a known synthetic audit event from durable storage. Later proof included the associated firewall and archived objects with encryption and retention metadata. Those are concrete deployment achievements. This was the demonstration environment of my company, Vector Mosaic; a hospital production rollout had not been established.

I also prepared the non-code work a sale would require: hospital and Epic analyst review materials, deployment and data-flow explanations, support procedures, cost and pricing work, and a bounded pilot proposal. The pilot was intended to measure reliability, workflow value and clinician trust. Agreements, institutional approvals and patient benefit could not be assumed from a running stack; commercial readiness still had work remaining when I stopped pursuing the opportunity.

Product validation and the decision to stop

I developed the idea without knowing that Epic had a comparable native capability coming. I continued pursuing it while I believed there could be a worthwhile reason for a hospital to adopt it. By April, my commercial assessment explicitly treated Epic-native AI as a serious competitive threat and asked what Pre-Note Delta could deliver better or sooner.

Seeing a native Epic offering closely match the workflow I had targeted, and seeing physicians respond positively to that kind of tool, reinforced my confidence in the idea. Epic now describes built-in summaries of patient history and recent developments. A published study of its encounter-summary tool also reported favorable clinician feedback: in a small pilot, 64% preferred a practice with the tool available. These are external signs that the problem and workflow have value, rather than evidence of Pre-Note's own clinical outcomes. [Epic's product description](#) and [the JAMIA Open study](#) provide that market context.

My commercial conclusion was that a good product idea did not necessarily justify competing with the platform owner. A small market might still have existed, but I decided that pursuing it was not the battle I wanted to fight. The timing here is my knowledge and decision sequence; I am not claiming that Epic copied the product or that I preceded every native chart-summary effort.

Pre-Note Delta did not become a commercial success. It did produce a substantial integrated application, a way for agents to work with authenticated vendor information, browser and desktop automation, distinct evaluations of automation and generated prose, and a demonstrated cloud deployment. It also strengthened my judgment about when to use model flexibility, when to replace it with deterministic execution, and how to make an external integration testable without confusing an attractive demonstration with proven clinical value.