PROJECT CORTEX

Case Study

Sep 2025 - Oct 2026



What a brain-inspired security engine taught me about evidence, software reliability, and knowing when to stop.

Written by: Hamza Hafeez

“An idea is at its most convincing when it has never been tested. Building it is how you discover what you were wrong about.”

— Hamza Hafeez

What this is all about

I built a system to audit code for security vulnerabilities. It had a research paper behind it, a custom orchestration architecture, and an ambition to make application security accessible to people who couldn't read source code.

How to read this. Margin notes explain terms and say where a fact comes from. The blank space beside the text is yours. Write in it.

It also had a habit of breaking in ways that had very little to do with artificial intelligence.

The most useful thing it ever did was point a human at a line of code worth reading.

This is an account of how I built it, what went wrong, and what I learned from the experience.

Please understand: Project Cortex was the name of the research I did, Cortex EDR was the name of the application I built on that research, and then later added two CLI tools named Cortex CLI and Cortex Attack on it.

Date	November 12, 2026
Period	September 2025 to October 2026
Status	Archived, October 2026
Reading time	About 20 minutes
Perfect if	You have called an LLM API and read a Postgres error message.

WHAT'S INSIDE

1. How it started 4
2. A Research Paper With No Pulse 5
3. What I built 7
4. The Ledger of Bad Nights 9
5. Three Months of Silence 11
6. The CLI and the attack tool 14
7. Lights Off 15

Chapter 1

How it started

In the summer of 2025, I wanted to build something great. You know, being young and into technology, everyone wants to build **Jarvis**, or the world's best neural network. Obviously, I was one of those enthusiasts, but instead of Jarvis, I wanted to build **AGI**.. don't creep, I know I'm a bit fictional, but it is what it is.. I wanted to build AGI, and I still want to build it.. fr..

Jarvis: is most famous Tony Stark's fictional artificial intelligence assistant in the Marvel Cinematic Universe

AGI: is a hypothetical type of artificial intelligence that can match or surpass human capabilities across most or all intellectual and cognitive tasks.

But the problem at that time was that I only knew some of the AI vocabulary: **RAG**, **MCP**, reinforcement learning, supervised fine-tuning, prompting, and some of the mathematics and coding behind them. I could follow conversations about AI systems, understand their architecture, and build applications around existing models. But I was still far from having the engineering depth I wanted.
I came to software from a medical background in Sep 2024, so at that time, I had been only one year into tech.. which left me with one advantage: I had some understanding of how the human body worked.

RAG: An AI technique that combines large language models (LLMs) with external knowledge bases.

MCP: An open-source standard created by Anthropic in 2024 to connect AI applications to external systems, data sources, and tools.

When I started Project Cortex, I was a **full stack dev**, I could and have made some full stack apps both for myself and for a few freelance clients as well, and was just breaking into learning AI as well.

A **full stack dev** is an engineer who can build both frontend & backend of an application.

Anyways, it was around one in the morning. I was alone in my room, watching the ceiling fan turn, trying to finish a thought that kept coming back.

Every major AI lab seemed to be pursuing the same general direction of building a larger, more capable model.

I thought, what if they are doing it wrong? We do have models, a lot of em, and specialized in one domain. What if we just make an environment and add those models in it to work like a team under a boss?

Each member would specialize in a particular task. One would handle research, another would evaluate information, another would write code. A central component would decide which specialist should work, what information it needed, and what should happen next.

The system would behave like an organization of specialized cognitive processes.

Being from medical, I instantly went towards digging into how human brain does that, cuz we are mimicking the brain..
I went back to what I knew about anatomy, and I got it there!. The **Pre Frontal Cortex** is the part of the brain that does the same thing I was imagining, acting like a boss and orchestrating other brain parts for a task.

Pre Frontal Cortex is the front part of the brain's frontal lobe, located directly behind the forehead and eyebrows. It acts as the brain's "control room" or central coordinator for higher-level thinking, behavior, and emotion.

It is involved in executive functions, including planning, working memory, decision-making, and behavioral control. It does not perform every cognitive operation itself. It helps regulate and coordinate activity across different parts of the brain.

I began thinking about it as a conductor.

The analogy became simplified. The prefrontal cortex does considerably more than coordinate activity, and the brain does not operate like an organized orchestra. It gave me the spark.
What if an AI system had a dedicated executive component whose responsibility was to coordinate specialized agents rather than perform every task itself?

I was nineteen, and I thought I had found something that would change the world (my world).

I didn't yet know how much of the idea would survive implementation.

Chapter 2

A Research Paper With No Pulse

The research took roughly three months, although it was never three uninterrupted months.
University consumed part of the time. **Freelance** clients consumed another part.

I was in in 2nd year of **university** studying computer science while building full-stack applications for **freelance** clients at that time, while trying to understand neuroscience, AI architecture, and this research behind multi-agent systems.

The result was a 36-page research paper dated November 28, 2025.

It contained 51 references, a hierarchical decision-process formalism, and a proposed Risk Agent capable of vetoing decisions made by the central **orchestrator**.
The paper described an architecture in which specialized agents handled individual responsibilities while a higher-level executive component managed their execution and evaluated their outputs.

An **orchestrator** is a system, software tool, or even a person that coordinates multiple independent parts into a single, smooth, and working whole.

I should make one thing clear about its authorship.
ChatGPT helped write the formal prose. The research direction, reading, architectural ideas, and work behind the paper were mine. I revised substantial sections myself, but I did not independently produce every sentence. I would rather be explicit about that than present the document as a thing it wasn't.

I was and am proud of it.

I gave copies to professors at my university. They said it was impressive. Friends were encouraging. But nobody challenged the assumptions, questioned the formalism, or asked how I intended to validate the architecture.
I know most of them, maybe all of em, haven’t read all 36 pages. Why? Maybe cuz I was nineteen, and I had handed them a research paper about an ambitious AI architecture. I could hardly expect them to become reviewers overnight. But, it was okay, they were amazed that, oh, this kid actually did that, and I had an aura in my class as well.

But the more uncomfortable problem was inside the paper itself. There was no experimental data. No working implementation. No **benchmark** against existing architectures. The repository was promised for later, and the website was still listed as something to be established.

A **benchmark** is a standard test or reference point used to measure and compare the performance, accuracy, or quality of a research method, algorithm, or piece of code

The paper had reached the point where the next step was obvious: implement the architecture and find out whether it worked.

I also wanted to commercialize it.

I chose application security as the first application. cuz I found it awesome, and because something sparked it.

A friend of mine had recently built an app using AI and asked me to check it for vulnerabilities. I had skills, but I said, "In return?” He said, “Oh man, you can have a burger”. “My choice?” “Ugh, okay, just do it.”
I produced roughly 3 pages of findings. They weren't polished, and my investigation was limited by what I knew at the time. Maybe only a few points in the pdf were useful, but that was enough for him to give me a treat. And that made me think, if he can pay me a burger for these rough findings, I wonder how many would pay me for real findings, and I know vibe coders are growing, and they don’t understand application security.

Chapter 3

What I built

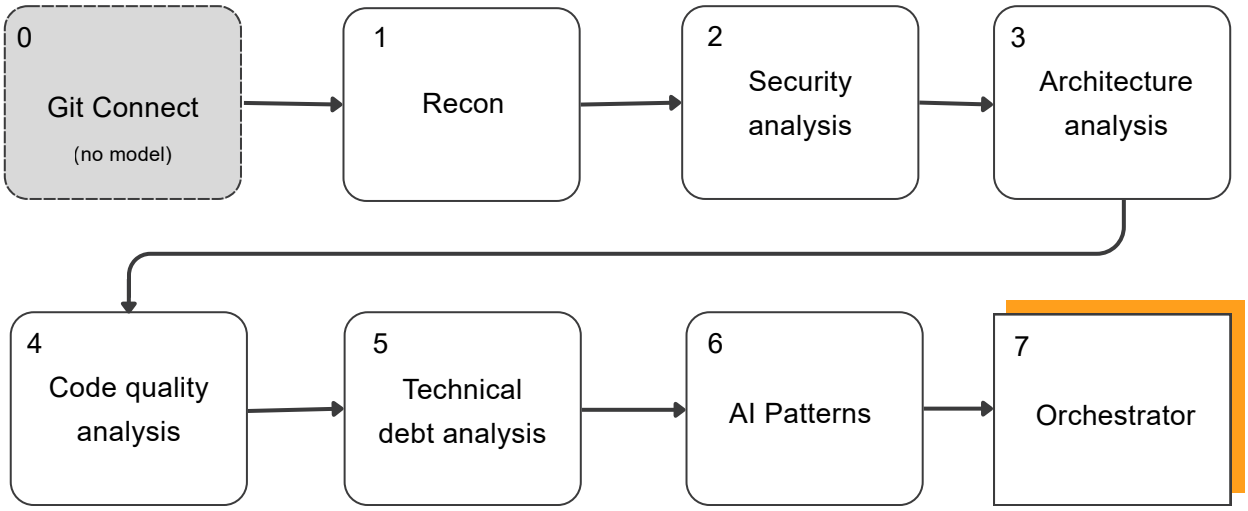
I started designing it, named it **Cortex EDR**.
The product was straightforward from a user's perspective. Sign in, open a dashboard, submit a GitHub repository URL, watch the analysis pipeline run, and receive a security report.
The implementation was less straightforward, seriously.

EDR means Endpoint Detection and Response. It is a cybersecurity technology that continuously monitors end-user devices, I designed the name's one part from my research of brain, other from cyber. (It have no affiliation with EDR, the name selection was cuz it looked cool to me)

I designed the pipeline around eight stages, numbered 0 through 7. Stage 0 was responsible for **reconnaissance**. It fetched the repository's file tree through the GitHub REST API. It did not call an LLM.

Reconnaissance is the intelligence-gathering phase where the agent or an attacker systematically maps out an application's internal architecture, capabilities, and vulnerabilities before launching an attack

The remaining seven stages were handled by specialized agents:



The final agent collected the outputs of the preceding stages and generated an executive report, including an overall score between 0 and 100.

The architecture followed the original research idea of mine.. specialized components performed individual tasks, while a central component assembled their findings.

There was one important difference between the conceptual architecture and the actual implementation.
The agents never communicated directly with one another. Which, in the paper I mentioned.

Each agent read information left by its predecessors in the database, performed its assigned task, and stored its results. The orchestrator then read the accumulated information and produced the final report.

The infrastructure constraints

Here, I want to mention what went wrong while developing and deploying the application.
This is intended for engineers; if you are non-technical, you may skip it till the end of chapter 4.

I was deploying on hobby-tier infrastructure. There was no Git binary available in the environment, and I couldn't rely on the disk storage.
That ruled out the obvious approach of cloning repositories and analyzing them locally.
Instead, I built the repository inspection process around GitHub's REST API.
The system maintained a manifest of the repository's file tree. Individual agents requested files through the API, using pattern-based filtering to select relevant source files. Each stage retrieved roughly a hundred files at most.
Even then, the model did not receive an entire file.
Each model call was limited to approximately 4,000 to 6,000 characters, generally covering the first hundred lines. Code beyond that limit was invisible to the model unless another retrieval operation brought it into context.

This was a significant limitation. A vulnerability could depend on code located much further down the file, or on behavior distributed across several files.
I had to make decisions about which files to retrieve and how much context to send.

Also, the prompts imposed strict output requirements.
Agents were expected to return JSON. A security finding needed an exact line number and a code snippet. If the model could not support a finding with that information, it was expected to return no finding.
I also limited the number of findings per file. Without that restriction, some models would generate long lists of speculative problems cuz the prompt asked them to investigate vulnerabilities.
The system used a five-provider fallback chain:

- OpenAI
- OpenRouter
- Gemini
- Groq
- DeepSeek

The intention was to keep the pipeline operational when an individual provider became unavailable or failed to return a usable response. (don't tell anyone, the reason was to cut my costs)

The reports were often useful. They were also noisy.
False positives were a recurring problem. A report could identify a legitimate concern, but it could also describe a vulnerability without evidence to support the conclusion.

I initially treated this as a problem that better prompts and more specialized agents would solve.
That assumption did not hold up as well as I expected.

The first production version

CortexEDR launched on 5th March 2026, the day I turned twenty. Happy Birthday to me.
By the end of active development, the codebase contained approximately 29,000 lines of only TypeScript in a single Next.js monolith, supported by nine database migrations.
There had been 197 production deployments.
That number is worth mentioning because the system was not stable. Several deployments existed to fix problems introduced by earlier attempts to fix other problems.

I learned much more from those deployments than I had from the original architecture diagram.

Chapter 4

The Ledger of Bad Nights

CortexEDR was a monolith running a multi-agent pipeline, retrieving repositories through GitHub's API, streaming execution events to the frontend, and tracking model usage.
It worked under controlled conditions.
Production introduced different conditions.
The following incidents are drawn from the actual implementation history. They are worth documenting because most of them had nothing to do with the sophistication of the AI architecture.

1. The pipeline that died with its response

A user submitted a repository for scanning.
The interface received a scan ID immediately. The dashboard showed the scan, but nothing happened afterward.
The database row remained in pending.
Vercel showed no useful stack trace.
The problem was in POST /api/scan/start.
The route inserted a database row, deducted one unit from the user's quota, called runPipeline() without awaiting it, and returned an HTTP response.

Locally, this worked. Node continued executing the asynchronous function after the response had been sent.
The serverless environment behaved differently. Once the response completed, the execution environment could be suspended. The unawaited promise had no guarantee of finishing.
The scan had been started, but the infrastructure had no reason to keep the process alive.
I eventually moved the pipeline to a container that could remain active for days. That solved the immediate execution problem.
It did not solve everything.
The architecture still had no reliable retry mechanism if the process died halfway through a scan. A long-running container is not a job queue.

Looking back, I should have designed the workflow around a durable job system from the beginning. The API route should have created a scan record and enqueued a job. A worker would process it, update its state, and retry failed operations from a known checkpoint.
Inngest was already in the repository, handling billing emails.
I had a job runner available and had used it for the wrong job.

2. Eight idle nodes

The frontend displayed eight agents.
All of them were dark.

The model calls were succeeding, and the backend was processing the scan. The customer, however, was looking at a frozen interface.
Both the logs and the customer were reporting something real.
The issue was a database constraint.

The application did not properly handle the returned error. As a result, current_agent never advanced, and the frontend never received the status it needed to light up the agent nodes.
The frontend was waiting for processing. The database would only accept running.

Two commits on April 26, thirteen minutes apart, fixed the issue.
The first logged the Supabase error. The second changed the status to running and updated the frontend hook to accept the correct state.

The database had been enforcing its constraint correctly. My application had been ignoring the response.
A second problem made the incident worse. The frontend polled the entire agent_events table every 500 milliseconds. That was unnecessary traffic, and it eventually became a performance concern of its own.

The lesson was a successful model call does not mean a successful application operation. Every boundary between the application and its dependencies needs proper error handling.

3. The clean report that meant nothing

One of the more concerning failures involved the security stage.
A scan completed with zero findings on a repository that contained obvious problems.
Another scan returned a finding with a CWE identifier that the model had never actually generated.
There was no exception. The agent had been marked as completed.
The prompts demanded JSON, and the providers were already configured to use JSON output modes where available. That did not guarantee that every response could be parsed.
Some models prepended explanatory text. Others returned reasoning inside a <think> block.
My parser attempted to recover the JSON by removing code fences and extracting content between the first [and the last].
When that failed, it returned an empty array.
That was the problem.
An empty array meant two completely different things:

- The model had completed its analysis and found nothing.
- The model had returned an invalid response that the application could not interpret.

The system treated both outcomes as success.
On June 24, I introduced a confidence gate and limited findings to five. That reduced some excessive output, but it did not solve the underlying parsing problem.
The correct approach would have been schema validation. If the model returned an invalid payload, the agent should fail explicitly, and the raw response should remain available for inspection.
A clean report with no findings. In my implementation, it could just as easily mean that the parser had given up.

4. Two hosts, one callback

Authentication worked on one part of the application and failed on another.
The marketing site showed a successful sign-in. The application subdomain redirected the user back to login.
In other cases, the user could authenticate and run a scan, only to find that their history was empty and their quota had already been deducted.
Authentication was split across two hosts:

- cortex-edr.com
- app.cortex-edr.com

On June 24, at 17:18, I hardcoded the OAuth callback to the application host because the marketing host returned a 404 on /api/auth/callback.
At 17:43, I reverted that change.
Google's token exchange required the callback URL to match NEXTAUTH_URL. The new callback configuration had introduced another authentication problem.
The two commits were only 25 minutes apart.
There was a second issue in the sign-in callback. It reported success even if the user record failed to insert into the database.
The application also used different identifiers in different parts of the workflow. The profile was loaded by email, while the scan was inserted using session.user.id.
Those values were not guaranteed to refer to the same database identity.
The solution required a single, consistent authentication origin and a single user identifier across the entire workflow. Profile creation and related operations also needed proper transaction handling.
The application had treated authentication as successful before confirming that the corresponding database state actually existed.

5. The disk filled

After a series of failed scans on the long-running container, new scans began failing with: ENOSPC
The operating system was reporting that there was no space left on the device.
Each scan created a workspace: /tmp/cortexedr-<scanId>
Successful scans deleted their temporary files.
Failed scans did not.
The cleanup logic was incomplete. The finally block flushed telemetry but did not remove the workspace.
The move from serverless execution to a persistent container had made the earlier temporary-file problem visible. On Vercel, the execution environment disappeared, taking its temporary storage with it. On the container, failed workspaces accumulated.
There was another inefficiency. Every model call rewrote an entire transcript file, including the prompts, rather than appending only the new information.
The event loop was doing unnecessary file operations throughout the scan.
The fix should have included unconditional cleanup in finally, incremental transcript writes, and a scheduled process to remove abandoned workspaces.
A cleanup job is useful as a backup. It should not replace proper resource management in the main execution path.

Other failures

The incidents above were not the only ones. It’s what I can recall and produce.
Stage 0 initially downloaded files without a token and retried anonymously after receiving a 401 response. That meant the platform could generate up to sixty unauthenticated requests per hour.
It also swallowed failed downloads, ignored the repository tree's truncation flag, and still reported that the repository had been indexed successfully.
The conversational interface had its own context-management problem. Its token compressor always preserved the first issue, even when that issue contained a 5,000-character source file. The file retrieval was also tied to a branch hardcoded as main.
The PDF generation route initially waited around ninety seconds for a font CDN. Once that was fixed, it encountered another error.
There was also a deployment failure caused by initializing new Resend() at module import time. CI did not have an API key, so the application could not deploy.

None of these problems required a new AI model or a better orchestration algorithm.
They required more careful software engineering.
That was probably the most uncomfortable part of building Cortex. I had spent months thinking about the coordination of artificial intelligence, while basic application boundaries were causing production failures.

Chapter 5

Three Months of Silence

The first major obstacle to selling CortexEDR was not technical. It was payment infrastructure.

Stripe is an international payment gateway used for billing, It isn't available in Pakistan. **Paddle**, is a merchant of record, works for billing.

I was based in Pakistan. **Stripe** was not available to me as well as PayPal and Wise as well.
I eventually set up **Paddle** and worked through its verification process.

At the time, I had almost no users.
I was worrying about payment processors before I had established that anyone wanted to pay for the product.
Then there was the name.

Palo Alto Networks is a global cybersecurity leader known for building enterprise firewalls, cloud security solutions, and automated threat-detection platforms.

I designed the logo, chose the name Cortex, bought the domain, and only afterward discovered that **Palo Alto Networks** already had a major security product called **Cortex XDR**.

Cortex XDR is an Extended Detection and Response (XDR) software application developed by Palo Alto Networks.

That was an expensive lesson in doing basic market research before committing to a brand.

I began with people I knew. Friends and developers in my circle tried the application and gave positive feedback. Just "WOW".
Product Hunt received zero upvotes. LinkedIn generated no meaningful response. X and Instagram were equally quiet.
I had made no product videos and avoided using my voice in promotional content because I did not want to appear on camera.
I had assumed that if the product was good enough, people would come.

People didn't come.

For approximately three months, the pipeline ran for very few people.

I had built something that required users to submit their source code, trust an unfamiliar security product, understand an AI-generated report, and decide what to do with the results. I had spent most of my time on the engineering and almost none on explaining why someone should care.

Rejections

During this period, I applied to **Y Combinator** twice. Both applications were rejected, without even any feedback.
I also pitched at **NIC** Lahore and was rejected.
Another university startup competition did not go anywhere.
At the time, I thought people weren't understanding the technology.
That was partly true, but it was also an excuse.
At a later competition at Beaconhouse National University, sponsored by Google Developers Group, Cortex received third place.

Y Combinator is a premier Silicon Valley startup accelerator that helps early-stage companies grow quickly with funding, network and programs.

NIC is a government-backed startup incubator network across Pakistan. It provides with free coworking spaces, mentorship, grants, and network to help turn innovative ideas into scalable businesses.

I had made the architecture the center of the pitch instead of the problem the product was supposed to solve.

Afterward, I wrote that security should be easy enough for a non-technical person to understand and implement. That was the clearest description of what I had been trying to build, probably, and it came much later than it should have.

Cortex was kinda baby that never stopped crying.

Moving Anyways

I wanted CortexEDR to do more than produce reports.
If the system found a legitimate issue, there was a possibility of contributing an actual fix to the software being analyzed.

In May 2026, I pointed CortexEDR at InsForge, an open-source backend platform with Y Combinator backing.

The scan highlighted two areas: how the authentication flow validated redirect URLs and how production logging handled errors.

I manually reviewed both before reporting anything.
I submitted a pull request that changed the redirect behavior to deny redirects by default when no allowlist was configured. It also prevented the **Winston** logger from printing **stack traces** in production and added tests.

Winston is a popular, general-purpose logging library for Node.js applications, and **Stack Trace** is a snapshot of the active function calls in an application at the exact moment an error or crash occurs on a live system.

Automated reviewers did not identify a problem with the initial version.
A maintainer did.

The default-deny behavior would break existing users who had upgraded without configuring an allowlist. The application had intentionally allowed this configuration to reduce friction during development.

I changed the implementation to produce a warning instead.
The automated reviewers then marked the revised behavior as critical.
Another maintainer explained the design decision: an empty allowlist was permissive by design, and the dashboard already warned users about the configuration.

The redirect change was unnecessary.

I reverted it and rewrote the tests to protect the existing behavior. A third maintainer approved the logging change.

The pull request was merged on May 8.

What actually changed?

Production logs no longer expose stack traces containing internal file paths, and the redirect matcher gained a regression test suite.
The vulnerability I had initially emphasized was not considered a vulnerability by the maintainers. It was an intentional design decision.

My original description of the contribution was more generous than the final diff deserved.
That distinction matters.
A security scanner can identify behavior that looks suspicious. It cannot always determine whether that behavior is a vulnerability without understanding the application's intended design.
The source code alone does not contain every relevant product decision.

Two more contributions.

I submitted another set of changes to Apache Superset, now renamed Paradev.
The work included AES-256-GCM encryption for stored Slack and Linear tokens, along with a fix for an insecure direct object reference (IDOR).

An IDOR occurs when an application allows someone to access another user's data by changing an identifier in a request without properly checking authorization.
During review, maintainers identified a database query that referenced a column I had not selected.
They also pointed out that existing plaintext tokens could cause failures after deployment if the migration did not account for them.

I corrected both issues. A maintainer closed the pull request in July.

Chapter 6

The CLI and the attack tool

By this point, it was clear that the web application was not attracting consistent usage.

I started questioning the way people had to access it.
A developer interested in checking a repository had to visit a website, create an account, and go through the application workflow. Perhaps that was too much friction for someone who only wanted to run a quick security check.

I decided to build an open-source command-line tool.

There were three reasons for doing it. I wanted more people to try the technology, I wanted to learn Go, and I wanted a concrete engineering project that I could show independently of the web application.

Cortex CLI

Cortex CLI was written in Go. In May 2026.

The file walker and project scaffolding were added on May 4. The first security detectors arrived on May 6. All eight architecture detectors were implemented on May 8. The terminal interface, AI provider layer, and commands followed on May 10.

The CLI had three stages:

- Reconnaissance.
- Security.
- Architecture.

An important architectural decision separated detection from explanation.

The detectors were deterministic functions. They looked for specific patterns in source code. A finding had to exist before the language model was involved. The model then received the finding and explained it.

In the CLI, the model did not decide whether a vulnerability existed. It explained an observation produced by a deterministic detector.
That separation made the output easier to reason about, although it also meant that the CLI could only detect problems covered by its existing rules.

Cortex Attack

While working on the CLI, I started thinking about another limitation.
Why restrict reconnaissance and security analysis to source code?
Applications run on networks. Their exposed services, web servers, dependencies, and configurations can reveal problems that are not obvious from source-code inspection alone.
I began building Cortex Attack.
The tool orchestrated several existing security utilities:

- Nmap
- Nikto
- Semgrep
- Trivy
- WhatWeb
- Curl
- A custom crawler

It used disposable Docker environments when a required tool was not installed on the host.
The results were then passed to a language model, which attempted to connect individual findings into an attack graph and generate remediation suggestions.
The attack graph was intended to show relationships between separate observations.

The tool was restricted to passive analysis of localhost. It was not intended to scan arbitrary external systems without authorization.
Cortex Attack was published on npm and PyPI.
Installation was simple:
npm install -g cortex-attack
At its peak, the package recorded 176 downloads in one week.
That was modest compared with established security tools, but it was actual distribution outside the original web application.

A change in architectural direction

Looking at CortexEDR, Cortex CLI, and Cortex Attack together revealed a thing I had not properly considered when writing the original research paper.
The paper placed most of the intelligence in the orchestrator. Specialized agents were expected to perform their individual tasks, while the central component coordinated their work and synthesized the result.

The CLI and Attack tools took a different approach.
They relied heavily on existing deterministic security tools. The model was used to interpret and explain the results after the tools had produced them.

In practice, this was a more constrained responsibility for the language model.
I still believed specialized agents could provide value in more complex investigations. But I had not demonstrated that the original seven-agent architecture consistently outperformed a simpler implementation.

Chapter 7

Lights Off

In October 2026, I stopped active development of Cortex.

No more feature releases. No more launch attempts. No more spending time keeping the live product operational.

The decision was not based on a single technical failure.
The project had consumed a substantial amount of time, and the available usage did not justify the cost of maintaining the complete ecosystem. The commercial model had not worked, and I had other engineering priorities that needed attention.

I could have continued adding features. There was always something else to build.
But another deployment would not have solved the underlying problem.

I decided to preserve the research, code, and documentation instead of continuing to operate a product without a convincing reason to do so.

What remains is a 36-page research paper, approximately thousands of lines of code, two open-source security tools distributed through package registries, one merged contribution to an external project, and a collection of production incidents I can still remember without opening the issue tracker.

As a business, Cortex did not work.
As an engineering experiment, it produced something more specific than the original research paper had promised.

Looking back at the production failures, a pattern becomes apparent. Most of the serious incidents occurred at the boundaries between components.
The agents themselves were rarely responsible for these failures.
The software around them was.

This is not an argument against multi-agent architectures. It is an observation about where my implementation spent most of its time failing. I had concentrated on designing the intelligence layer while underestimating the engineering required to keep the rest of the system consistent.

There is another question I still cannot answer.
Did seven specialized agents actually outperform one well-written prompt on the same repository?
I believed they would. The research paper argued for specialization, and the product was built around that assumption.
But I never conducted the controlled comparison.

I never ran the same repository through both architectures, used the same models and evaluation criteria, and measured whether the additional coordination produced better results.
The paper called for experimental validation. I moved into product development before completing that part of the research.

If I were starting again, I would validate the architecture before building the product around it. I would also speak to potential customers much earlier. Fifty people willing to use a product would have taught me more about its commercial direction than several months of building features in isolation.
I would probably look for a cofounder with experience selling to businesses, too. I spent most of my time thinking about the engineering and treated distribution as something that could be addressed afterward.

It couldn't.
I still think the original architectural question is interesting. I just have less certainty about the answer than I did at nineteen.

I also no longer think the choice of framework, language, or model determines whether an engineering project succeeds. Those decisions matter, but they cannot compensate for an incorrectly understood problem or an architecture that has never been tested against a simpler alternative.
The fan is still on my ceiling.
I still have trouble sleeping when a thought refuses to finish. I haven't stopped having those thoughts. I've become more careful about what I do with them.

Before I start building now, I try to find out who actually needs the thing.
And whether I've spoken to them.

Appendix: Project Records

Attribute	Details
Founder & Engineer	Hamza Hafeez
Thesis	Brain’s prefrontal-cortex-inspired executive coordination for multi-agent AI systems.
Products	Cortex EDR, Cortex CLI and Cortex Attack
Model Providers	OpenAI, DeepSeek, Gemini, OpenRouter
Codebase Languages	39,528 lines of TypeScript, Go, and SQL (PostgreSQL). YAML for CI, Dockerfile.
Major Technologies	Next.js, Postgres (JSONB as shared agent memory), Supabase Realtime, polling, NextAuth.js, Paddle, Resend, Inngest, Vercel, Railway, Docker, GitHub Actions, GitHub REST API
Production Deployments	197
Scan Time	aprox. 2-2.5 mins
Registered Users	528
Monthly Active Users	8-16
Downloads	928 (Cortex Attack)
Peak Weekly Downloads	176 (Cortex Attack)
Github Stars	31
Followers	97
Total Revenue Generated	$100
Total Development Cost	$60 (Cursor & Claude Code)
Total Operational Costs	$115 overall, ~$15/month (server, domain & services)

Thank you so much for reading this case study

linkedin.com/in/hmza-hb github.com/hmza-hb hmza.cc

PROJECT CORTEX



An engineering case study of multi agent orchestration, AI driven security analysis, research, experimentation, and the lessons learned building Cortex from September 2025 to October 2026.