

Flavio Espinoza

v23

Senior Full-Stack Engineer, AI-Native

flavio.espinoza@gmail.com • [Schedule a call](#) • [flavioespinoza.com](#) • [GitHub](#) • [LinkedIn](#) • (385) 216-8969 • Salt Lake City, UT

AI is a force multiplier, not a replacement. Most people still have not figured that out. I have.

Senior full-stack engineer, AI-native since 2024, and a frontier-model trainer, paid to evaluate coding agents and author the benchmarks that grade them. Top 3% ChatGPT power user — I know exactly where these tools are great and where they break, because I both build with them and sharpen them. TypeScript end to end, and since September 2026 Rust for the parts that have to stay up and stay honest: sentinels, memory engines, and the gate in front of a retrieval service.

Skills: TypeScript, React, Next.js, Node.js, Python, Rust, C# / ASP.NET, Angular, MongoDB, Elasticsearch, Google Cloud

Experience

Sentinels

Jul 2026 – Present

A family of independent, read-only watchdogs over production systems, each written in the language that fits its host. The rule is the same for every one of them: a sentinel cannot act on the system it watches, holds no key, and pages my phone the moment something goes wrong.

- **Sol Bot Sentinel** • C# + ASP.NET -- a watchdog that sits beside the live Sol Bot trading engine and never touches it. Built on ASP.NET Core (.NET 10) as a small minimal API with three background watchers. Deployed on production on Cloud Run in Oct 2026, and public on GitHub in the sentinels repo. The watchers read the engine's feed every 15 seconds over a private network (VPC). One checks that every trend flip was acted on. One checks the bots' conduct: position caps, leverage caps, and a heartbeat that has gone stale. One checks risk: the price stop, the loan-to-value (LTV) guard, and the carry floor. Each alert fires once, sits behind a bearer token, and pages my phone through Pushover. A /health endpoint goes red the moment a watcher stops polling. Read-only by design: it cannot place, change, or close a trade, and it holds no key. On its first run it caught a

restart replaying months-old trend flips into new bots; the engine now ignores any flip older than the bot itself. 38 `xUnit` tests, a multi-stage `Dockerfile`, and `GitHub Actions` CI on every push. Built and deployed on production in one sitting.

- **Sentry Engine** · `C++` -- a real-time audio engine, built to the filter stage and running in tests only, not deployed. It is a graph of audio nodes. Settings change in two smoothed steps so the sound never clicks. The core is a `BiQuadFilter`, a standard two-pole audio filter, in the Transposed Direct Form II layout, and it allocates no memory while audio is flowing. It smooths its math on every sample and refuses any frequency the sample rate cannot carry. No raw `new` anywhere, `RAII` and `Pimpl` throughout. 13 tests of the audio math (impulse response, a single-bin `DFT`, stereo parity, bypass identity) and a benchmark the compiler cannot optimize away, all under `CMake`. The graph scheduler is still a stub.
- **Candle Watch** · `Rust` -- specified for the Sol Bot trading system and in build, local first, then preview, then production on `Google Cloud`. A small daemon on `tokio` + `mongodb` + `axum`. It reads the newest candles for each market and keeps one simple state per market: ok, stale, flat, or unreadable. When a state changes it pages me through `Pushover`, and it pages once more when things recover. It never repairs anything and never trades. It survives a VM reset under `systemd`, built from a distroless `Cloud Build` image.
- **Obligation Reader** · `Rust` -- specified for the Sol Bot trading system and in build. It uses `solana-sdk` to decode Kamino obligation accounts straight from the chain. It publishes each reading next to the value the `TypeScript` decoder got, with the difference between them. It is a live parity check: it pages me when the two disagree, and it is never used to make a trading decision.
- **Corpus Sentinel** · `Rust` -- designed, with the first beams building now. An `axum` + `rusqlite` + `tokio` service that stands in front of a `TypeScript` retrieval engine. It handles the bearer-token auth, the rate limit, the health check, and the private firewall boundary, and it passes requests through to the engine on `localhost`. The engine keeps its matrix loaded while the surface the outside world can reach stays small.
- **Tribal Knowledge Watcher** · `TypeScript` -- `watcher-tribal-knowledge`. It reads `Slack` screenshots with `Gemini Vision` (OCR), turns them into structured `Markdown`, and feeds that to a retrieval engine.
- **Directive Sync Watcher** · `TypeScript` -- `watcher-directive-sync`. It watches a specs folder and syncs any directive change into the global ruleset.
- **Veritas Sentinel** · `JavaScript` -- a black-box recorder. It reads `JSONL` logs and turns them into a live `Markdown` HUD, a heads-up display.
- **Mercy** · `JavaScript` -- an older `Node` watcher that lives inside eScanner.
- **FX Watch** · `Shell` -- `fx-watch`. A `launchd` watcher that renames and files every screenshot into an asset store the moment it lands.

Sol Bot

May 2026 – Present

A FULL PRODUCTION Solana DeFi trading system I designed, built, and deployed end to end for a private client, solo, with AI as a force multiplier: a sharper blade in a hand that still has to know where to swing. Live on Google Cloud today: autonomous bots watch the market trend and, on a flip, open and close leveraged Kamino JLP/USDC Multiply positions on chain, unattended and 24/7 -- every private key sealed inside a Turnkey signing vault the bots can ask but never hold. 1,064 automated tests green, plus a Playwright end-to-end suite that drives the live dashboard.

- **A. Wield AI the way most developers still have not learned to** – as a force multiplier under tight direction, scoping and reviewing like an eng lead over a team of ICs, to ship production systems solo that would normally take a team; every architecture and correctness call my own.
- **B. Own the whole stack in production** – a headless Node/ESM TypeScript engine on a hardened Google Cloud VM (systemd-supervised, secrets injected from Secret Manager at process start, test-gated stamped deploys) and a Next.js / React / Tailwind operator dashboard on Cloud Run, coupled only through MongoDB Atlas so one person can hold the whole system in his head.
- **C. Custody by construction** – every bot mints its own wallet inside a **Turnkey signing vault**; a policy-gated non-root signer can sign a Kamino trade and nothing else (proven live: it signed a real Multiply, then refused a rogue transfer from the same code path), and withdrawals can only land on the client's hard-coded wallet.
- **D. Reverse-engineered Kamino's 16-instruction, ~48-account Multiply transaction** from real transactions with no SDK -- Anchor discriminators verified 10/10, accounts re-derived from first principles -- and proved the full signed write-path (open, close, swap) on mainnet: build, simulate as a hard gate, sign, submit, confirm. Even the leverage limits are proven against the chain, not the docs: 1x and 6x rejected by Kamino's own program, 2x-5x shipped.
- **E. Everything on-chain, no SDKs** – Kamino, Jupiter, Pyth, and Helius decoded from raw account bytes (Kamino's own SDK returns malformed constants and is banned), the engine building its own JLP/USDC candles directly from on-chain reads every minute, and borrow APR, JLP yield, and LTV computed from numbers it tracks itself, validated against live chain state.
- **F. Engineered the trade path for correctness under races** – an in-process signal bus (never the database) as the only trigger, an atomic compare-and-set that makes "never double-open" real, monotonic idempotency by candle, boot-time crash recovery, and a three-axis risk watcher (price / LTV / borrow-carry) proven live: the stop fired at a 0.1% trigger on a real position.
- **G. Built the trend engine as one shared math core** feeding both the historical backtester and the live Binance.US signal feed, proven byte-identical over the full

SOL/USDC lifetime (7,625 candles, 474 flips, zero drift) -- then backtested 84 leveraged positions with real soft-liquidation simulation before real money moved.

- **H. Own the client relationship end to end** – scope, build, deploy, handover: the client operates the live system today through his own Google login, funds bots from his own Ledger, and pays invoices in USDC on Solana. Money was the last thing wired, gated behind green backtests and a funding gate.

AI Tooling

Jun 2026 – Present

Every crate follows a coding standard I wrote first (one crate per module, rustfmt with house tabs, `clippy -D warnings` clean, `Result` shapes fixed, `tokio` only where it pays, no regex where a plain string call works, no literals in source), so the second Rust component looked like the first.

- **A. Shipped `session-filter`, a Rust memory engine with an AI judge in the loop** – reads a Claude Code session transcript (JSONL, 4 MB in 0.2 seconds), strips the harness noise on an explicit allowlist, drops re-injected rules and IDE context, scores every passage with a deterministic heuristic, then hands the candidates to a Claude judge invoked through the CLI login (`claude -p`, no API key) that returns typed JSON verdicts: score, category, tagline, one-line note. Output is a dated note under 80 lines, grouped Decisions, Rules, Open Threads, Facts, Wins, Losses, every item pointing back to its event in the full transcript. First real run: 207 events in, 20 kept. `serde`, `serde_json`, `chrono`, `clap`; 14 unit tests; fmt and clippy clean.
- **B. Wrote the standard before the second crate** – the Rust coding directive every crate above follows, plus the rule that a Rust component talks to a model through the Claude Code login rather than an API key, so nothing it runs can spend outside the plan.

AI Voice Interface

May 2026 – Present

Model-agnostic AI voice interface: one audio pipe where the AI at the seam is swappable. I use Gemini for research, Claude for coding, ChatGPT for brainstorming -- none of them had hands-free voice I could actually use, so I built one platform that gives me all three.

- **A. Built a model-agnostic AI voice platform** – one audio pipeline (Silero VAD, Deepgram STT, streaming TTS) with the AI at the seam deliberately swappable; v1

runs the **Claude Agent SDK** (full CLI toolset), Phase 2 plugs in Gemini for research and ChatGPT for brainstorming behind the same pipe.

- **B. Killed my own first design.** v1 was Python/FastAPI middleware that quietly neutered Claude -- three tools, a system prompt it ignored, hallucinated timestamps with no shell. Deleted the middleware and ran the **Claude Agent SDK directly**, archiving the Python backend as a tombstone. Control → variant → ship the winner.
- **C. Owned the full stack solo** – React/Vite/TypeScript front end (AudioWorklet 16kHz PCM capture, streamed reply bubbles, live karaoke highlight) and a Node/TypeScript WebSocket backend; same React/TS/Node + Anthropic Claude API stack production teams ship on.
- **D. Built a real-time voice loop** – server-side Silero VAD (energy fallback + pre-roll so the first word is never clipped), Deepgram nova-3 streaming STT with keyterm prompting, and a **streaming Aura-2 TTS pipeline** that synthesizes per sentence, strips code from the spoken path, paces paragraphs, and resumes from interruptions.
- **E. Instrumented its own spend** – live context-token indicator and a model-fallback cost alert that fires the moment the SDK returns a cheaper model than requested; dual-format (JSON + Markdown) chat persistence with full-text search.

Mercor

Nov 2025 – Present (Contract)

Across six contracts (one active, five completed) I A/B evaluate frontier coding agents on real production codebases and write the rubrics that grade them -- the human-preference signal that tunes the next generation of models.

- **A. A/B evaluate coding agents head-to-head** on real-world open-source repositories – I run the same engineering task through two agents, then produce a structured preference judgment across task success, instruction following, code quality, and thoroughness.
- **B. Author the tasks themselves** – the debugging and feature-development scenarios used in these evaluations, including the acceptance criteria and the specific failure modes each task is designed to surface.
- **C. Document agent behavior against a defined taxonomy** – every finding supported by verbatim evidence pulled from the session record, never a paraphrase.
- **D. Analyze full agent session transcripts** to reconstruct what an agent did, in what order, and why -- including behavior that never appears in the agent's own summary of its work.
- **E. Deliver written evaluations against a platform-reviewed quality bar**, under confidentiality and originality requirements.

- **Skills:** Large Language Models (LLM), LLM Evaluation, Prompt Engineering, Code Review, Debugging, Software Testing, Rubric Design, Data Annotation, Artificial Intelligence (AI), Technical Writing, TypeScript, Python

Snorkel AI

Dec 2025 – Apr 2026 (Contract)

Frontier-agent evaluation -- the sharpening side of the craft. I author verifiable coding tasks for the Terminus benchmark: each a self-contained challenge with a deterministic oracle solution, a Docker environment, and behavior-based tests, calibrated against several frontier coding agents, so one task teaches and measures more than one model family at once.

- **A. Design adversarial evaluation tasks calibrated to a target difficulty band** – scenarios engineered so frontier models fail two to three of five attempts: hard enough to expose a real capability gap, never unsolvable.
- **B. Run head-to-head evaluations of competing frontier models** on self-authored scenarios, producing structured comparative judgments rather than pass/fail scoring.
- **C. Author the rubrics and the five-test suite** backing each scenario, so every evaluation is reproducible and defensible against a fixed standard.
- **D. Passed an LLM-as-a-judge quality gate on every submitted evaluation** – two returned as exemplary.
- **E. Worked under confidentiality and originality requirements** – every scenario and rubric written from scratch.
- **Skills:** Large Language Models (LLM), Prompt Engineering, Red Teaming, Adversarial Machine Learning, LLM Evaluation, Rubric Design, Benchmarking, Reinforcement Learning from Human Feedback (RLHF), Artificial Intelligence (AI), Technical Writing, Quality Assurance

Bless Network

May 2024 – Mar 2025

Bless Network (formerly Blockless) -- seed-funded (~\$8M: a \$3M pre-seed and a \$5M seed), ~12-person decentralized-compute startup. The world's first shared computer, where everyday devices contribute idle browser compute for token rewards. Growth was the product; I own the user-facing growth surface end to end, with no spec, just conversations with our founders and CTO.

- **A. My Chief Technology Officer directed me to use AI to start coding** – my first time. I used **ChatGPT** to figure everything out (May 2024), and it is where I went from never having used AI to running it as daily core infrastructure.

- **B. Replaced the old email/password/verify signup with a one-click Web3Auth single sign-on (OIDC)** – Google, GitHub, and passwordless email, each deriving a self-custodial wallet and authenticating by challenge-response signature instead of a password. The marketing lead, tracking MailChimp signups against the prior six months, reported **completed signups up ~70%** and **abandonment down ~45%** (team-reported, not instrumented).
- **C. Compounded an underperforming referral program** (a 10% bonus already in place) by building stacked, OAuth-verified social and quiz boosts on top -- X +5%, Discord +5%, quiz +5% -- summed into one live "Total Boost" to drive virality. The founders reported **engagement up ~45%** (team-reported, not instrumented).
- **D. Fought the X/Twitter OAuth integration to completion** against an API mid-rename from twitter.com to x.com with badly out-of-date docs – a moving-target integration with no clean answer, shipped by iterating until it worked.

Akash Network

Feb 2022 – Dec 2022

Overclock Labs (Akash Network) – a seed- and token-funded decentralized cloud marketplace on Cosmos, ~20 people. On a three-person, fully-remote team I proposed the stack and owned the front end, the Node (Fastify mutual-TLS) proxy, the in-browser wallet and certificate crypto, and the four-stage CI/CD pipeline behind the Akash Console -- turning a nine-step command-line deploy into a web app a developer could actually finish.

- **A. Turned a nine-step blockchain deploy protocol into a guided, finishable flow** – from template to running workload -- hiding wallet signing, on-chain certificates, bid selection, and lease creation behind a usable UI.
- **B. Proposed the stack and built the multi-step (5-step) SDL deployment wizard** (React + create-react-app, Formik validation) with an 18-plus template marketplace of pre-validated node configs pulled via the GitHub API.
- **C. Designed the Fastify mutual-TLS proxy and WebSocket real-time monitoring** – streaming logs, Kubernetes events, lease status, and an xterm shell back from 100-plus providers, with throttled rendering tuned for continuous streams.
- **D. Built the four-stage GitHub Actions CI/CD pipeline** – auto-versioning, multi-arch Docker builds, staging, and approval-gated production.

WebShield

Apr 2019 – Feb 2022

WebShield -- a seed-stage healthcare-identity startup (~7 people; seed led by New Enterprise Associates, with a milestone-based convertible note up to \$10M) building a

person-centric healthcare authorization and consent network. As the sole front-end engineer, I owned the authentication layer and the Node API server behind Exemplar, the app that demonstrated the product to customers and investors.

- **A. Built a production single sign-on (OpenID Connect) layer from scratch, implemented directly from the OAuth 2.0 / OIDC spec on a Koa/Node backend** – the authorization-code flow, JWT signature verification, and state/nonce replay/CSRF protection by hand -- because no off-the-shelf module existed for our stack.
- **B. Led the team's pivot off a stalled enterprise-federation effort (PingFederate) to Okta**, clearing the configuration bottleneck and freeing the CTO to focus on the core network.
- **C. Designed a contract-first "mock network"** – a GraphQL `resolved_person` + `trust_summary` projection served as fixtures -- that let the entire product ship and demo **months ahead of the Go backend**; going live to production was a single endpoint-URL swap.
- **D. Owned both directions of the network API: publishing new people and organizations into it (ingest) and querying resolved identities back out (discovery + trust query)**, all behind that one swappable contract.

Street Fighter

Jun 2017 – Apr 2019

A professional crypto trading terminal where the chart IS the interface -- chart-as-order-entry, Fibonacci batch ordering, and a from-scratch indicator library. Same client as eScanner.

- **A. Built the chart itself into the order-entry surface** – a custom **D3.js v4** candlestick engine (reworked from react-stockcharts) where open orders are **draggable lines** on the price chart.
- **B. Built drag-to-modify orders** – dragging an order line triggers an atomic, fully async **cancel-and-replace**: cancel, wait for escrowed funds to release, recalculate the amount while preserving total cost, then place the new limit order.
- **C. Built Fibonacci batch ordering with curve-based "paradigm" distributions** – draw a fib range and deploy 10+ weighted limit orders in one action, front-loading the deepest rungs to cover a spread.
- **D. Built the real-time Node.js + Express + Socket.IO backend on CCXT's** unified exchange layer, with live WebSocket order/balance updates and a D3.js depth chart; shipped in versions (DAX → Bittrex + HitBTC → HitBTC). This project later won me the **Sol Bot** commission.

eScanner

2018 – 2019

A real-time cryptocurrency market scanner I built solo for a private trading client -- live-streaming an entire exchange's markets, aggregating them in Elasticsearch, and surfacing the moves worth trading through fast, filterable candlestick views. Same client as Street Fighter.

- **A. Built a real-time crypto market scanner end to end (sole developer)** – live exchange data over **Socket.IO**, **Elasticsearch** time-series aggregation, and interactive **D3.js / React-Stockcharts** candlesticks.
- **B. Koa.js API on an 8-instance PM2 cluster** behind Nginx with SSL and `ip_hash` sticky-session load balancing across 5 Socket.IO upstreams.
- **C. Streamed HitBTC market data via CCXT** into OHLCV candle aggregations with configurable intervals and percent-change / volume filters, watchlists, and ignore-lists.
- **D. React / Redux SPA (50+ components)** with JWT / Passport auth, Stripe subscriptions, and SendGrid email.

Swim AI

Sep 2016 – Jun 2017

Swim AI (now Nstream) – a seed-stage edge-computing startup building a real-time streaming and IoT (Internet of Things) analytics platform for device tracking and monitoring; a tight core R&D team of fewer than 15 in San Jose during my tenure. It went on to raise a \$10M Series B (ARM, Cambridge Innovation Capital) after I left.

- **A. Designed and developed a web map application on the FLUX pattern** with unidirectional IoT data flow.
- **B. Augmented human decision-making with the most accurate, relevant real-time and contextual data** – turning live device data into operational decisions.
- **C. Delivered projects for the San Francisco Transit Authority** (live bus tracking), **Palo Alto** (traffic-light monitoring), the **City of Chicago** (transformer and meter locations and power consumption), and **Boston** (automated street-light control). Built with React, the Google Maps API, and Swim APIs; deployed on AWS.

SolarCity

May 2015 – Sep 2016

I was a Lead Full-Stack Engineer on the internal software team that built the real-time installation-analytics platform.

- **A. Led development of a real-time data analytics platform** (Python, Django, Angular) for high-volume solar-installation metrics.
- **B. Architected a Django REST API** serving real-time analytics to an Angular front end with D3.js visualizations.
- **C. Integrated Elasticsearch** for full-text search and analytics across installation records and customer data.
- **D. Optimized MS SQL Server queries**, cutting dashboard load times from **40+ seconds to under 3**.
- **E. Wrote C# / ASP.NET back-end services** behind the Angular analytics front end.

Vivint Solar

Apr 2013 – May 2015

I joined as the 6th engineer on a lean core development team that rapidly scaled to 45 engineers over 14 months.

- **A. Led a 3-person engineering team building Mercury**, Vivint Solar's real-time sales-and-installation analytics platform — a key source of the **sales-and-revenue analytics and presentations used in the company's 2014 NYSE IPO due diligence**.
- **B. Engineered an Angular pivot dashboard** to replace static HTML readouts -- letting stakeholders instantly slice-and-dice installation metrics, sun-hour distributions, and sales-rep performance in real time.
- **C. Supported macro-filtering** (state, system size, module count), account-level drill-downs, and instant **CSV export** without interface degradation.
- **D. Built the authentication layer and the Node API layer** that hooked into a new near-real-time **Elasticsearch** indexed data store — our backend team stood up that store, ingesting relational rows from **SQL** databases and **Salesforce** via the **Elasticsearch JDBC River**, replacing a legacy daily reporting job that took **over an hour**. Our APIs were what tied into it and powered the slice-and-dice.
- **E. Built C# / ASP.NET back-end services** for Mercury alongside the Angular front end.