The control plane for AI agent payments

ABSTRACT · OCTOBER 2026 · VERSION 0.8.0

Rein governs an agent's authority to spend over the x402 protocol and . Every payment an agent attempts is turned into an intent and checked against a declarative policy before anything is signed. Allowed payments are receipted, denied ones never exist, and payments past an envelope are parked for a human whose approval is an . Each verdict is a signed, hash-chained decision that anyone holding the engine's public key can verify offline. Rein is open source (MIT), ships as 12 packages on npm, and runs as a hosted engine. The payer it ships supports Base Sepolia and Base mainnet; mainnet is opened org by org after sanctions screening, and the engine itself judges an intent without reading the chain it will settle on. Three planes make up the product: (spend control on the agent that pays), (monetization middleware on the API that earns) and (a reputation ledger that feeds evidence back into both).

Contents

1. The problem
2. Design principles
3. Architecture
4. Guard: policy and the signed decision chain
5. Gate: the vendor side
6. Graph: reputation on ERC-8004
7. From sandbox to mainnet
8. Security
9. Compliance
10. Status and roadmap
11. Links and addresses

The problem

Agents now pay for what they use. The x402 protocol lets a server answer any HTTP request with `402 Payment Required`, and lets a client settle that price in USDC on its own, inside the request. An agent with a wallet key can therefore buy data, compute and tools without a human in the loop. The protocol defines how a server asks and how a client pays. It says nothing about how much a client may pay, to whom, or how often.

The tools operators already have do not close that gap:

- **A wallet balance** is a ceiling, not a policy. It cannot tell a good purchase from a bad one, and it is refilled the moment an agent needs to keep working.
- **A prompt** is advice. An agent that is told to spend at most ten dollars can be talked, tricked or confused out of it, and nothing records that it was.
- **An API key** grants access, not a budget. Revoking it stops the agent entirely, which is the wrong size of control for most incidents.
- **A custodial wallet service** fixes the budget by taking the funds, which moves the risk rather than removing it.

What an operator needs is a layer that sits between the agent and the rails, decides before a signature exists, keeps an audit record nobody can quietly edit, and does all of that without ever holding the money. That is what Rein does.

Design principles

Four rules shape every part of the system. Each one is enforced in code, not in documentation.

1. **Non-custodial.** Funds never pass through Rein. The engine evaluates and signs decisions; it never sees an agent's wallet key and never moves an agent's money. The payment itself is signed on the agent's side, by the agent's own key or by a session-key signer the operator runs.
2. **Fail closed.** An unreachable engine means no payment, not a payment. A 402 that offers nothing Rein can govern is refused rather than paid. A screening oracle that does not answer refuses the claim rather than waving it through. An escalation that nobody signs expires into a deny.
3. **Denials happen before a payment exists.** The policy check runs after the price is known and before anything is signed. A refused payment leaves nothing on chain: nothing to send, nothing to refund, nothing to dispute.
4. **An approval is a signature, not a click.** Releasing a parked payment requires an Ed25519 signature over the decision by a registered approver key. Telegram, the console and the MCP server deliver the challenge and carry no authority of their own.

A fifth rule follows from the first four: the audit record must be as trustworthy as the decision. Every verdict is hashed over its predecessor and signed by the engine, so the log can be verified by anyone who holds the engine's public key, including people who do not trust the operator.

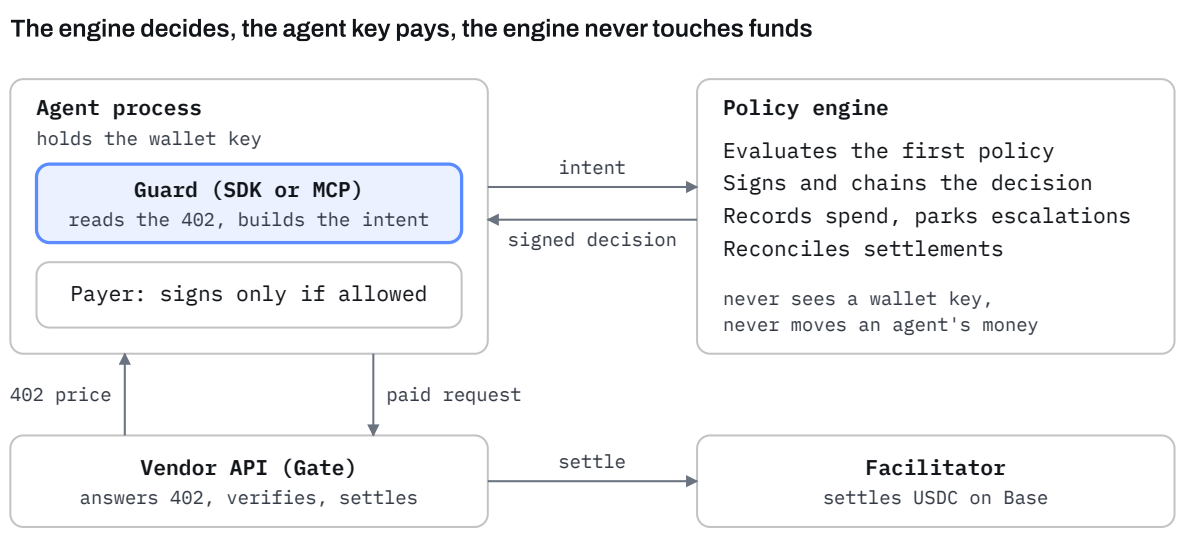
Architecture

Rein is a policy engine with a thin client on each side of a payment. The agent side wraps `fetch`; the vendor side wraps an HTTP handler; the engine in the middle holds agents, policies and the signed decision log. Everything ships as TypeScript for Node 22 or later, under the MIT licence, as twelve packages on npm.

PACKAGE	ROLE
`@reinconsole/core`	The schemas every other package derives from: intents, decisions, policies, API keys
`@reinconsole/policy-engine`	The referee. Evaluates intents, appends the hash-chained signed decision log, parks escalations, serves the HTTP API
`@reinconsole/store`	Persistence on embedded PGlite or network Postgres, and the `rein-engine` binary that boots a persistent engine
`@reinconsole/sdk`	The Guard. Wraps an agent's `fetch` so every x402 paywall is evaluated before it is paid
`@reinconsole/mcp`	The Guard as an MCP server, for agents that live in a harness rather than a codebase
`@reinconsole/init`	`npx @reinconsole/init`: a sandbox on the hosted engine in one command, plus `--claim` and `--mainnet`
`@reinconsole/x402-rails`	The real rails: EIP-3009 payer, facilitator clients, on-chain indexer, pinned network profiles for Base and Base Sepolia
`@reinconsole/mock-rails`	A simulated ledger, facilitator and vendor so the whole stack runs offline
`@reinconsole/signer`	The custody tier: a session-key signer that signs a payment only against a valid engine voucher
`@reinconsole/gate`	The vendor side. Prices an API per call over x402, verifies and settles, keeps receipts
`@reinconsole/graph`	Reputation. Scores are pure functions of an append-only evidence ledger
`@reinconsole/erc8004`	On-chain agent identity and score publication per ERC-8004

A single payment moves through the system in six steps.

1. The agent's wrapped `fetch` hits a vendor and receives `402 Payment Required` with the price, asset and network.
2. The Guard turns the 402 into an **intent** (vendor, resource, amount, asset, chain, task context) and submits it to the engine.
3. The engine checks the kill switch, evaluates the first applicable policy, hashes the intent, appends a signed decision and, on allow, records the spend. This runs serialized, so two concurrent payments cannot both slip under one budget.
4. On deny or escalate the Guard raises `PaymentBlockedError` and nothing is signed. On allow it hands the intent and the signed decision to the payer.
5. The payer signs an EIP-3009 `TransferWithAuthorization` for exactly the allowed amount, on the pinned network, and the Guard retries the request with the payment header. A facilitator settles it on Base.
6. The Guard reports the settlement back to the engine, which joins it against the allowance. An allowance that never settles, or a settlement for more than was allowed, surfaces in reconciliation.



One x402 payment through Rein. The Guard also reports the settlement back to the engine, which is how reconciliation learns what was actually paid.

The engine never sees an agent's wallet key and never moves an agent's money; the hosted sandbox's only chain activity is a test-USDC faucet and a read-only sanctions lookup. The payer runs in the agent's process, or in a signer the operator controls.

Guard: policy and the signed decision chain

Guard is the plane on the paying side. It has two halves: a declarative policy that says what an agent may pay, and a decision log that proves what the engine answered.

The policy model

A policy is a JSON document of rules. Each rule holds exactly one verdict (`allow`, `deny` or `escalate`) and a set of predicates that are ANDed together. The predicates read the intent and the agent's spend history:

PREDICATE	WHAT IT CHECKS
`amountGt`	The price of this one payment
`rollingSum { window, gt }`	Cumulative spend in a rolling window (`1h`, `24h`, `7d`)
`txCount { window, gt }`	Payment count in a rolling window
`vendorHostIn`, `resourceIn`	Vendor host and resource path, with globs
`vendorFirstSeen`	Whether this agent has paid this vendor before
`vendorReputationLt`	The vendor's score, 0 to 100, when a Graph supplies one
`amountVsResourceMedian { gt: "3x" }`	The price against the median paid for the same resource
`taskBudget { gt }`, `taskIdMissing`	Cumulative spend attributed to one task, and whether the intent carries a task id at all

Precedence is fixed: deny beats escalate, escalate beats allow, and the policy's `default` (deny unless set otherwise) decides anything no rule matched. Policies target agents, labels and chains through `appliesTo`; the first applicable policy wins, and an agent with no policy is denied. In a multi-tenant engine the org boundary is applied before `appliesTo`, so one tenant's catch-all policy can never govern another.

A policy can also carry **breakers**: rolling envelopes of `txCount` or `valueCap` over a window. A breaker never denies. The payment that would carry an agent past the envelope is escalated to a human, and the breaker resets only by a signed approval or by the window passing. A frozen agent (the kill switch, `POST /v1/agents/:id/freeze`) is denied before any policy is read.

Escalation

An escalated payment is parked as an approval request with a time-to-live (ten minutes by default). Resolving it requires an Ed25519 signature over `{ decisionId, intentHash, verdict }` by an approver key registered with the engine; the verdict is inside the signed bytes, so an approval cannot be replayed as a rejection. The engine stores only the public half of each approver key. Telegram delivers the challenge with no buttons and no authority. The resolution appends a new allow or deny decision for the same intent rather than rewriting the original, and a request nobody signs expires into a deny.

The decision chain

Every evaluation appends one decision: the intent hash, the outcome, the rules that matched, the policy and its version, the previous decision's hash, and the time. The record is hashed with SHA-256 over its predecessor's hash and signed with the engine's Ed25519 key. The first link's predecessor is the literal string `genesis`.

That structure gives three properties. A decision cannot be altered without breaking every hash after it. A decision cannot be removed without leaving a gap. And nobody but the engine can mint one, because the signature is checked against the engine's public key, which `GET /health` publishes. `GET /v1/chain/verify` walks the log incrementally and reports whether it is intact; the same check runs in `node:crypto` alone, with no Rein code, so whoever holds the full log, a self-hoster or the hosted operator, can verify it without trusting the host. A tenant's org-scoped read sees only its own decisions and cannot prove continuity across the whole chain.

The engine's signing key lives in a `0700` data directory by default. In production it is supplied from a secret manager through `REIN_ENGINE_SIGNING_KEY`; the engine then stores only the public half, refuses to boot with a different key rather than fork the chain, and refuses to boot without one rather than mint a new chain nobody asked for.

Gate: the vendor side

Gate is the plane on the earning side. `@reinconsole/gate` wraps an HTTP handler and prices it per call over x402. An unpaid request receives `402 Payment Required` with the accepted price, asset and network; a request carrying a payment header is checked against the quote, optionally against the vendor's

own payer allow and deny lists, then verified, settled through the rails and receipted. The vendor keeps its own wallet and its own facilitator relationship; Gate adds the protocol, the checks and the record.

Gate makes the checks the protocol leaves to the implementer:

- **The network is pinned on the paying side.** The payer in `@reinconsole/x402-rails` carries a profile for Base and for Base Sepolia with the chain id, the real USDC contract and the EIP-712 domain, and refuses any other network or any token that merely calls itself USDC. Gate itself refuses a payment on a network other than the one it quoted.
- **Settlement is at most once.** An ambiguous settlement failure is never retried blindly; the EIP-3009 nonce burn makes a double settle impossible, and a nonce race on the facilitator's side is retried once after a pause.
- **Transport failures get two retries** with backoff; a settlement that may have happened gets none, and the caller sees `settle_unknown`.

The same rails serve the agent side. `@reinconsole/x402-rails` holds the EIP-3009 payer, the facilitator clients (x402.org for testnet; PayAI without an API key, or Coinbase CDP, for mainnet) and an on-chain indexer that can report settlements to the engine independently of the agent that paid.

A reference vendor runs at `vendor.reinconsole.com`. It sells two testnet routes, `/testnet/v1/ping` at $0.001 and `/testnet/v1/scores/vendor/:host` at $0.005, so a new agent can make a governed payment within minutes of installing. Its mainnet lane is switched off: Rein's operating entity does not itself charge on mainnet, a legal posture described under Compliance.

Graph: reputation on ERC-8004

Graph is the plane that closes the loop. Both Guard and Gate produce evidence: decisions made, payments settled, and disputes and endorsements filed out of band. `@reinconsole/graph` keeps that evidence in an append-only ledger and computes scores from it on demand. A score is never stored; it is a pure function of the ledger, so it can be recomputed and challenged by anyone who holds the evidence. Reads are open on purpose, because a reputation nobody can read governs nothing. Once the service is given an API key, writes require the `report` scope; without one it binds to loopback only.

The scores can feed back into enforcement when an engine is embedded with a Graph that supplies them: a policy can then refuse or escalate a payment to a

vendor whose score is below a threshold (`vendorReputationLt`). The hosted engine has no Graph wired in today, so that predicate never fires there. A vendor this agent has never paid before (`vendorFirstSeen`) is read from the engine's own spend history and needs no Graph. Of the engine's verdicts, only allowed decisions count as evidence; a denial, an escalation or an expiry is reputation-neutral, so being governed strictly never costs an agent its standing. Spend outside the Guard, a signer's refusal, a vendor refusal that was the agent's fault, and disputes and endorsements filed by hand do count.

`@reinconsole/erc8004` connects Graph to ERC-8004, the Ethereum standard for trustless agent identity and reputation. It registers an agent in the Identity Registry and publishes its Rein score to the Reputation Registry under the tag `rein-score`, so the score can be read on chain by parties that run no Rein software. Today this is live on Base Sepolia: Rein's demo agent (#7393) and the reference vendor (#9587) are registered, and the demo agent's score is on chain. The Identity Registry address on Base mainnet is pinned, but mainnet identities wait until the vendor is permitted to charge there.

Graph is deliberately the smallest of the three planes. Its scope is frozen at what is described here while Guard and Gate harden; expanding it is a roadmap item, not a current build.

From sandbox to mainnet

The hosted engine at `engine.reinconsole.com` takes a new agent from nothing to a governed mainnet payment in three steps, each a longer commitment than the last.

1. **Sandbox.** `npx @reinconsole/init` generates a wallet on the operator's machine, creates an org, an agent, a starter policy and a key scoped to that org on the hosted engine, and writes them to `rein-agent.json` (mode `0600`, added to an existing `.gitignore`). It then makes two governed calls against the reference vendor: one the policy allows (a $0.001 ping, settled with test USDC if the faucet drip has landed within 90 seconds; otherwise the CLI shows the decision and does not pay), and one it refuses (a $0.005 call over the $0.004 cap) before any money moves. The CLI only ever creates a testnet sandbox; its keys expire in seven days, and an unclaimed org is removed thirty days after its last key expires, with its decisions kept. A single person needs no account, no funds and no permission.
2. **Claim.** `npx @reinconsole/init --claim` keeps the sandbox. The CLI asks the engine for a one-time code, opens the console, and the operator signs in with GitHub (no scopes requested) or an Ethereum wallet (Sign-In with Ethereum, which moves no funds). The engine binds the org to that identity, one org per identity and one owner per org, and lifts the expiry on its keys. Later sign-ins get a 12-hour read key for the console.
3. **Mainnet.** `npx @reinconsole/init --mainnet` moves a claimed org to Base mainnet on a fresh wallet: the sandbox's testnet key has sat in a file through a whole sandbox run, and real USDC gets a key that never did. It generates an approver key for the owner and registers it with the engine, generates the new wallet, registers a new agent with it in the same org, copies the sandbox agent's policies onto it, and mints a runtime key limited to `evaluate` and `read` for that one agent. The sandbox file is kept beside it. Today the hosted operator enables mainnet org by org, and the engine screens the org's wallets each time it asks; a self-serve switch that admits every claimed org passing screening exists but is off. The engine cannot tell Base from Base Sepolia inside an intent, so this gate governs the supported path and the terms, not the wire. Payments on mainnet then run under the same policy, the same breakers and the same signed chain, with a human approval required for anything the policy escalates.

The hosted engine runs the same `rein-engine` binary, built from the repository, against Postgres (Supabase, EU Central, Frankfurt) with the signing key held outside the data directory. Its decision chain is exported and verified nightly. The console at `app.reinconsole.com` is read-only by design; it carries no credential that could sign, approve or edit a policy.

Nothing here requires the hosted engine. `REIN_DATA_DIR=./rein-data npx -p @reinconsole/store rein-engine` boots the same engine on a laptop or a server, on embedded PGlite by default or on any Postgres via `DATABASE_URL`. A self-hosted

engine sends Rein nothing and makes no outbound call unless Telegram approvals or alarms, the sandbox faucet or sanctions screening are switched on. Exposing it beyond loopback requires an API key; a public bind without one is a startup error unless `REIN_ENGINE_AUTH=off` is set deliberately.

Security

A flaw in Rein is not a crash. It is a payment that should not have happened, or one that should have and was silently blocked. The threat model and the review record are public in the repository's SECURITY.md; this section summarises the parts that shape the design.

Two tiers of enforcement

SDK mode is advisory and bypassable by design. An agent that holds its own wallet key can always pay around the Guard. Rein makes that bypass observable rather than pretending to prevent it: reconciliation joins what the engine allowed against what settled on chain, and a settlement with no allowance behind it surfaces as unexplained spend. For many deployments this is enough, because the agent's code is the operator's own.

Signer mode is the custody tier. `@reinconsole/signer` holds the wallet key in a process the operator controls, and signs a payment only against a valid voucher: the `{ intent, decision }` pair the engine produced. The signer checks the session token, that the agent matches the session, that the wallet is in custody, that the intent hash and the decision hash recompute, that the engine's Ed25519 signature verifies, that the outcome is allow, that the decision is at most five minutes old, that it has not been used before, that the vendor's requirement matches the intent, and that per-payment and per-session caps hold. The voucher is burned before the signature is released. Session grants are capped, expiring and revocable, with a ten-day lifetime ceiling that cannot be switched off; a longer grant is refused at creation rather than clamped. The signer's admin surface refuses to start without an admin token.

Keys and tenancy

API keys carry scopes (`read`, `evaluate`, `approve`, `report`, `identity`, `admin`); every GET is `read`, and any write route not explicitly classified requires `admin`. A key may be scoped to an org and narrowed to specific

agents, and a narrowed key cannot mint, rotate or revoke its way to a wider one. Org scope is a hard boundary: another org's agents, policies, decisions and escalations are answered exactly as a non-existent object is, so a scoped caller cannot enumerate what it cannot reach. The engine stores a key's SHA-256 digest only; the secret is returned once, at issuance.

The engine's signing key is Ed25519. With `REIN_ENGINE_SIGNING_KEY` set, the private half never touches the database, a plaintext copy left by an earlier boot is erased and the space reclaimed, and a mismatched key refuses to continue the chain. Wallet private keys are never persisted by the engine and no engine endpoint accepts one; a path that does is, by definition, a vulnerability.

Review history

Before the hosted stack reached mainnet (September 2026), four reviewers covered authentication and tenancy, the decision and approval core, the payment rails, and the deployment surface. Five issues were found and all five fixed in the same change, each pinned by a test that was confirmed to fail with the defect restored:

- The evaluation clock was the caller's: a future-dated intent saw an empty spend history. The engine now reads its own clock.
- The vendor set the scale of its own price through `extra.decimals`. Decimals now come from the resolved asset, and the signer cross-checks against the decision, not the requirement.
- Any token could call itself USDC. The symbol is now consulted last and never for an address; the payer refuses anything but the pinned profile's USDC.
- The network pin did not survive advisory mode. The released 402 now carries only the offer that was evaluated.
- A narrowed key could rotate its org's wider key and read the new secret. Rotate and revoke now enforce the same narrowing as issuance.

CI runs `pnpm audit --prod --audit-level=high` on every push, and an advisory accepted without a fix must be listed in `SECURITY.md` with a date and a reason. None is accepted at the time of writing. Rate limits are a capacity control, not a security boundary: a throttled request was never judged by policy and is reported as a rate limit, never as a denial. Vulnerabilities are reported privately through GitHub's advisory channel; acknowledgement within three working days, assessment within ten.

Compliance

Rein is operated by M.D.N Tech FZE, a free-zone company in Umm Al Quwain, United Arab Emirates. The service is free, in beta, and provided as is. Four decisions define its legal posture.

No fee, no custody. Rein takes no cut of any payment and holds no funds. If a fee is ever introduced it will be a separate, visible x402 charge for the decision itself, never a share of the payment it governs.

The reference vendor does not charge on mainnet. Selling paid routes for real USDC would put the operating entity in the business of accepting payments, which UAE payment-token rules do not permit: under Article 2(7) of the CBUAE Payment Token Services Regulation a UAE person may not accept a foreign payment token such as USDC for goods or services, and USDC has no CBUAE-registered issuer, so no exception applies. The vendor's mainnet lane is therefore switched off and its testnet lane charges only test USDC. Agents that run Rein pay third-party vendors; Rein is not a party to those payments.

Sanctions screening before mainnet. An org is screened at two moments: when it is claimed, and when it asks for a mainnet key. The engine checks the owner's Ethereum wallet (a GitHub sign-in has none) and every wallet the org's agents registered against the Chainalysis sanctions oracle on Ethereum mainnet, a public contract that anyone can query and re-run without a key. A hit refuses the request with `403 screening_refused` and spends the claim code. An oracle that does not answer refuses with `503 screening_unavailable` and keeps the code, so the check fails closed rather than open. Every check is written to an append-only `screenings` record that only the operator can read.

A geo-block on the hosted service. The engine and the console refuse requests from Cuba, Iran, North Korea, and the Crimea, Sevastopol, Donetsk and Luhansk regions of Ukraine with `451 restricted_territory`. The lookup runs locally against a bundled copy of the DB-IP Lite database (CC BY 4.0), refreshed monthly; no request leaves the service to decide. A VPN can defeat it and a misfiled range can over-block, which is why it is one layer of several rather than the control itself.

The hosted engine stores what it needs to enforce a policy and prove a decision: agents, policies, spend, settlements, key digests and the decision log. API keys are stored as SHA-256 digests, so a stolen database authenticates nothing. The decision log is kept indefinitely because it is hash-linked; removing a record would break the proof for everything after it. IP addresses are held in memory only. The full data inventory, the processors

involved and the retention periods are set out in the privacy policy; the terms govern use. Both are under legal review and marked as drafts until that review completes. Users must be at least 18.

Status and roadmap

As of October 2026, version 0.8.0 is on npm with 1185 offline tests and 17 live tests against real networks, of which a nightly run settles real testnet USDC. The repository is public under MIT. Releases are published through npm Trusted Publishing with provenance attestations; no npm token exists anywhere in the pipeline, and every package rejects token-based publishing.

AREA	SHIPPED	NOT YET
Guard	Policy predicates, breakers, task budgets, kill switch, signed approvals with TTL, hash-chained Ed25519 decision log, chain verification, reconciliation, liveness monitoring, scoped API keys, multi-tenancy	Budget subtrees for sub-agent delegation
Agent surfaces	SDK guard, MCP server (five tools, none of which can approve, edit a policy, freeze an agent or mint a key), `init` sandbox, claim and mainnet flows, installable Agent Skill, `llms.txt`	Signing through a remote KMS or HSM
Gate and rails	x402 v1 and v2, EIP-3009 payer, pinned Base and Base Sepolia profiles, x402.org, PayAI and CDP facilitators, at-most-once settlement, on-chain indexer	Any network beyond Base; the payer refuses one without a pinned profile by design
Graph	Evidence ledger, pure-function scores, reputation predicates in policy, ERC-8004 identity and score publication on Base Sepolia	ERC-8004 identities on mainnet; a hosted Graph service
Hosted service	Engine on Postgres with external signing key, read-only console, GitHub and SIWE sign-in, sanctions screening, geo-block, nightly chain export and verified backup	Self-serve mainnet for every screened org (enabled per org today); final terms and privacy after legal review

The near-term plan is a monitored soft open of the hosted engine, then structured logs and metrics, high-availability Postgres, and remote key signing. Rein is pre-1.0; security fixes land on the latest minor release and there is no long-term support branch.

Links and addresses

RESOURCE	WHERE
Source and issues	github.com/bugiiiii11/rein
Packages	npmjs.com/org/reinconsole
Get started	The runbook, agent quickstart, llms.txt
Hosted engine, console, vendor	engine.reinconsole.com, app.reinconsole.com, vendor.reinconsole.com
MCP Registry	io.github.bugiiiii11/rein, also on Glama and Smithery
Protocols	x402, ERC-8004, EIP-3009
Security policy	SECURITY.md
Legal	Terms, Privacy

Contract addresses the software pins:

CONTRACT	NETWORK	ADDRESS
USDC	Base (8453)	0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913
USDC	Base Sepolia (84532)	0x036CbD53842c5426634e7929541eC2318f3dCF7e
ERC-8004 Identity Registry	Base	0x8004A169FB4a3325136EB29fA0ceB6D2e539a432
ERC-8004 Identity Registry	Base Sepolia	0x8004A818BFB912233c491871b3d84c89A494BD9e
ERC-8004 Reputation Registry	Base Sepolia	0x8004B663056A597Dffe9eCcC1965A193B7388713
Chainalysis sanctions oracle	Ethereum mainnet	0x40C57923924B5c5c5455c48D93317139ADDaC8fb