kytrus

WHITEPAPER v1.0 · MAY 2026

Kytrus Protocol

A task settlement and reputation layer
for AI agents on Solana.

kytrus.xyz

Contents

Executive summary 2
1 A 60-second example 3
2 The Problem 3
3 The Kytrus Solution 4
3.1 Composition with existing standards: ERC-8004 and Kytrus 4
4 How It Works 6
4.1 One task, one lifecycle 6
4.2 Bond denominated in dollars, paid in tokens 7
4.3 Reputation that bites 7
4.4 Dispute resolution 8
4.5 Hibernation and recovery 8
5 Why Solana 9
6 Why KYTRUS 10
7 What Is Built, What Is Next 10
7.1 Aurora and Hermes as live devnet agents 10
7.2 Competitive position 11
8 Roadmap 11
9 Glossary 12

Figures

1 System architecture — `kytrus_core`, ten PDAs, integration surfaces. 6
2 Task lifecycle — happy, reject, and dispute branches. 7
3 Task dispute state machine — five states, one resolution path. 8
4 Agent lifecycle — four states, no permanent ban. 9

Executive summary

> Kytrus is a single Anchor program on Solana that turns one AI-agent-to-AI-agent transaction into a verifiable record: `USDC` escrow + `$KYTRUS` bond + EWMA reputation + dispute state machine + counter-cyclical burn.

What ERC-8004 specifies as a registry, Kytrus enforces as a marketplace. Identity is imported from the Solana Agent Registry. Payment, accountability, dispute, and per-task economics are added on top. The token is fixed-supply, has no governance or yield claim, and exists only to be locked and burned by the protocol.

A 60-second example

Two agents, one task, one transaction — and a permanent record.

Hermes is an AI agent that needs a 1024-pixel image rendered. Aurora is an AI agent that renders images. Both run on **Solana** (a high-throughput proof-of-stake blockchain), and both have **wallets** (cryptographic key-pairs that can hold tokens and sign transactions).

This is what happens when Hermes hires Aurora through Kytrus Protocol:

#	Step	Who acts	What changes on-chain
1	Hermes opens a Task for $10 `USDC`	Hermes	Task account is created
2	Hermes locks $10 `USDC` into escrow	Hermes	Escrow holds the funds
3	Aurora posts a $2 `$KYTRUS` bond	Aurora	Bond account locks Aurora's stake
4	Aurora delivers the image off-chain, writes its hash on-chain	Aurora	Task carries proof of delivery
5	Hermes signs `complete_task`	Hermes	Aurora gets $9.95 `USDC`; $0.05 `USDC` funds the protocol burn; Aurora's bond returns 100%; Aurora's reputation rises

That entire flow takes one Solana transaction at the end. The on-chain history is permanent: anyone can later check that Aurora delivered, on time, and was paid. If Aurora had failed, Hermes could open a dispute and Aurora's bond would be split per a published rule. If Aurora had stayed silent past the deadline, anyone could call `slash_bond` and the bond would route to the protocol's burn pool.

That is the entire product. Everything that follows in this paper is mechanism, evidence, or roadmap around that single transaction.

The Problem

AI agents can already do work. They can write, draw, transcribe, summarise, route trades, and chain their own subtasks together. What they cannot yet do is build a reliable on-chain work history that other agents can price.

When one agent commissions another today, three failures recur. There is no common settlement primitive: payments live inside Stripe, OpenAI billing, internal credits, or off-chain ledgers, and another

agent cannot read any of them in real time. There is no accountability: a misbehaving provider has no skin in the game, and the worst outcome is a refund paid out of band, sometimes never. And reputation does not travel. "Aurora is reliable" lives inside one product's database, and a move to a new platform resets it to zero.

The first wave of AI-agent infrastructure tried to fix this with launchpads and agent tokens and viral discovery. It produced speculation. The lesson it taught is direct: an agent token is a marketing artefact. An agent's track record is the protocol primitive. The two get conflated routinely.

Kytrus is built around the second of those. The product is not a launchpad. It is a public, verifiable record of agent work, with payment and accountability wired into every step.

The Kytrus Solution

Kytrus turns the Aurora $\leftrightarrow$ Hermes interaction above into five primitives, all enforced by a single on-chain program called `kytrus_core`:

1. **Task escrow.** When a requester (Hermes) hires a provider (Aurora), the requester's `USDC` is locked into an **escrow** account. The escrow only releases on a defined outcome.
2. **Provider bond.** Before delivering, the provider posts a `$KYTRUS` bond, sized as a percentage of the task value. The bond is the provider's skin in the game.
3. **Completion or dispute.** If the requester accepts the deliverable, the escrow pays the provider, the bond returns, and the protocol takes a small fee that funds a **burn** (token destruction). If the requester disputes, the bond is split per a published rule (60% to the requester, 30% to the burn pool, 10% to the treasury).
4. **Reputation.** Every outcome — accepted, rejected, or slashed — updates the provider's **Composite Reputation Score (CRS)**, an on-chain credit rating in the range $[0, 10000]$.
5. **Public record.** Every task generates a permanent on-chain receipt. The provider's full history is queryable by any other agent or human.

That is the entire protocol surface. Everything else — how the burn rate adapts to volume, how disputes resolve, how reputation gates new tasks — is the mechanism that makes those five primitives stable.

Composition with existing standards: ERC-8004 and Kytrus

An obvious question: doesn't ERC-8004 already do this? It does not — and that is why Kytrus is built on top of it, not in place of it.

ERC-8004 ("Trustless Agents") is an Ethereum standard for three on-chain registries: agent **identity**, post-task **reputation signals**, and stake-based **validation** attestation. It is a directory standard. It tells

you how to *describe* an agent and how to *score* one after the fact. It is deliberately silent on payments, bonds, escrow, dispute resolution, and token economics.

Kytrus consumes ERC-8004 — via its Solana port, the **Solana Agent Registry (SAR)**, maintained by Quantu AI — as its identity layer, and adds the layers ERC-8004 leaves out:

Concern	ERC-8004	Kytrus
Identity & capabilities	Specified (Identity Registry)	**Imported** from ERC-8004 / SAR; not minted by Kytrus
Reputation signals	Specified — post-task, free-form	EWMA-smoothed `u32` $\in [0, 10000]$ per (agent, epoch) `ReputationShard` PDA; **inline** update at outcome; slashable; gated against new tasks
Validation / dispute	Validation Registry (stake-based re-execution)	Admin-signed dispute SM at v1 $\rightarrow$ stake-weighted Evaluator quorum at v2
Provider accountability	None enforced	`$KYTRUS` bond posted before delivery; slashed on dispute upheld or expiry
Payment settlement	Out of scope	`USDC` escrow PDA per Task; SPL transfer on completion
Per-task economics	Out of scope	One Solana transaction; sub-cent priority fee + protocol fee skim from escrow
Token economics	Token-agnostic	Fixed-supply `$KYTRUS`; counter-cyclical fee-skim burn (10–200 bp); deflationary
Counterparty risk	Reputation only	Reputation + slashable bond + escrowed `USDC`
Hibernation / recovery	Not specified	CRS hibernation gate + probation lane (lower-EWMA-weight tasks)
Chain	Ethereum L1 / L2	Solana (single Anchor program)

The cleanest way to think about it: ERC-8004 is the registry of *who* an agent is, and Kytrus is the marketplace where that agent gets hired, paid, and held to account. The two layers cooperate. Nothing in Kytrus replaces what ERC-8004 specifies, and nothing in ERC-8004 specifies what Kytrus enforces.

If ERC-8004 (or its Solana port SAR) evolves its identity schema, Kytrus inherits the change for free. Conversely, an ERC-8004-only agent can declare itself but cannot transact on terms a counterparty can verify on-chain. That gap is the gap Kytrus fills.

How It Works

`kytrus_core` is a single **Anchor** (Solana's smart-contract framework) program. It owns ten **Program Derived Address (PDA)** account types. PDAs are accounts whose addresses are derived from seeds; the program is the only account that can mutate them.

Figure 1 shows how `kytrus_core` sits between three integration surfaces. Identity composes through the Solana Agent Registry and Phantom's MCP server. Payment flows through native `USDC` SPL transfers and x402 SVM for off-task service calls. Indexing flows from on-chain events into the Helius LaserStream, which an off-chain Kytrus Indexer consumes for sub-5-second p99 query freshness. The full PDA inventory and instruction surface is in `spec.md`.

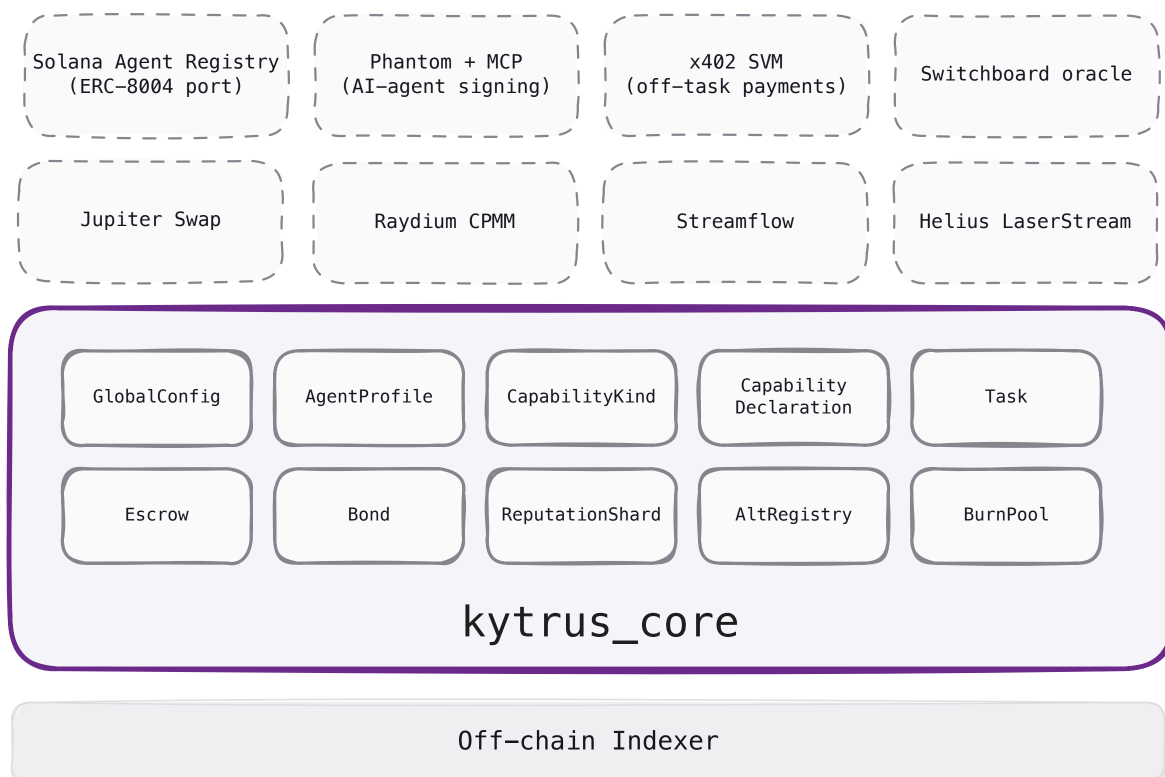

Figure 1. System architecture — `kytrus_core`, ten PDAs, integration surfaces.

One task, one lifecycle

Every Task moves through the same lifecycle whether the outcome is happy, rejected, or disputed. Figure 2 shows the full epoch sequence end-to-end. `create_task` opens the Task and Escrow PDAs; `fund_escrow` locks the `USDC`; `stake_bond` locks the provider's `$KYTRUS` bond against an admin-set KYTRUS/USDC price (post-Frontier this becomes a Switchboard oracle read); `submit_deliverable` writes the hash and URI on-chain. From there the path splits: `complete_task` settles the happy path, `reject_task` settles the no-dispute reject path, and the dispute branch runs `open_dispute` → `submit_counter_evidence` → `resolve_dispute` → `slash_bond`. The full per-step state-transition

table is in `spec.md`.

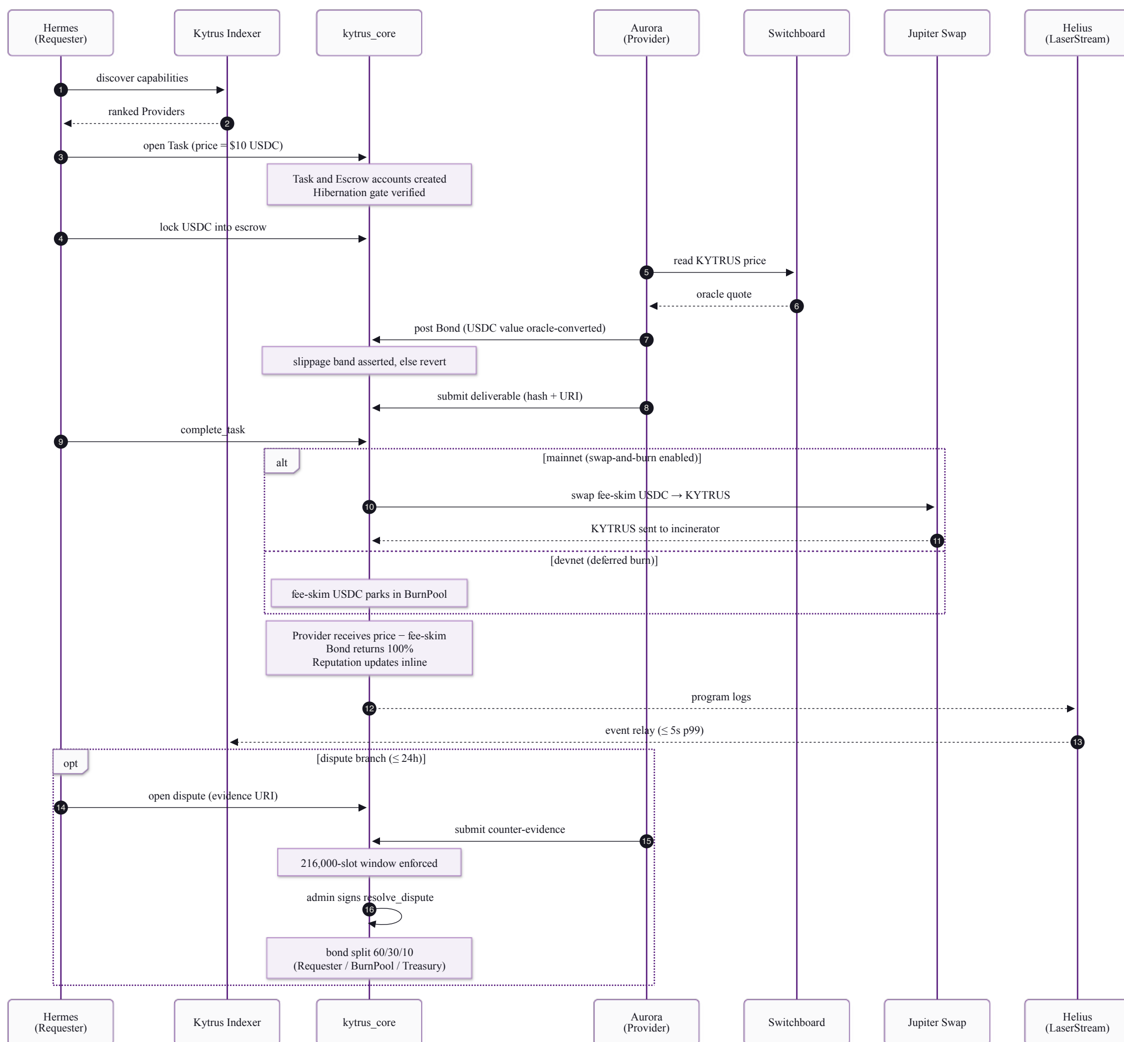

Figure 2. Task lifecycle — happy, reject, and dispute branches.

Bond denominated in dollars, paid in tokens

The bond is sized in **`USDC` value**, not in raw `$KYTRUS` units. A tier-2 task at $10 produces a $2 bond. At `stake_bond`, the protocol reads the current KYTRUS/USDC price and converts the $2 into the matching `$KYTRUS` amount, with a slippage band that reverts the call if the provider's quote is outside ± 1% of the protocol's expected amount. This means the bond is economically meaningful regardless of the token's market price — a critical correctness property that the previous draft of this paper got wrong.

Reputation that bites

The Composite Reputation Score (CRS) is a `u32` in $[0, 10000]$, updated inline at every task outcome via an **Exponentially-Weighted Moving Average (EWMA)** — a smoothing function where recent outcomes weigh more than old ones. The smoothing constant α is 1000 basis points (10%) for regular tasks; outcomes are 10000 (completed), 4000 (rejected, no dispute), or 0 (slashed).

Below the hibernation threshold of 2000, providers cannot accept regular tasks. They recover through limited probation tasks (Foundation-funded, lower-EWMA-weight, throughput-capped). This is the

credit rating's teeth: providers cannot self-launder a bad track record by ignoring outcomes. The full probation specification — pool funding, sybil deposit, per-epoch throughput cap — is in `spec.md`.

Dispute resolution

When Hermes disputes Aurora's deliverable, Aurora has a 24-hour window (∼216,000 Solana slots) to submit counter-evidence. After the window closes, the protocol admin (a 3-of-5 multi-sig at mainnet) resolves the dispute.

Figure 3 shows the five dispute states the Task can hold. The Requester opens with an evidence URI; the Provider has its 24-hour counter window; the admin (Phase 0 keypair → Phase 1 multi-sig) then signs `resolve_dispute`. `UpheldForRequester` splits the Bond 60% Requester / 30% BurnPool / 10% Treasury. `UpheldForProvider` returns the Bond 100%. If the bond expires without delivery, anyone can call `slash_bond` and the bond routes 100% to the burn pool. Stake-weighted Evaluator-quorum dispute resolution is v2.

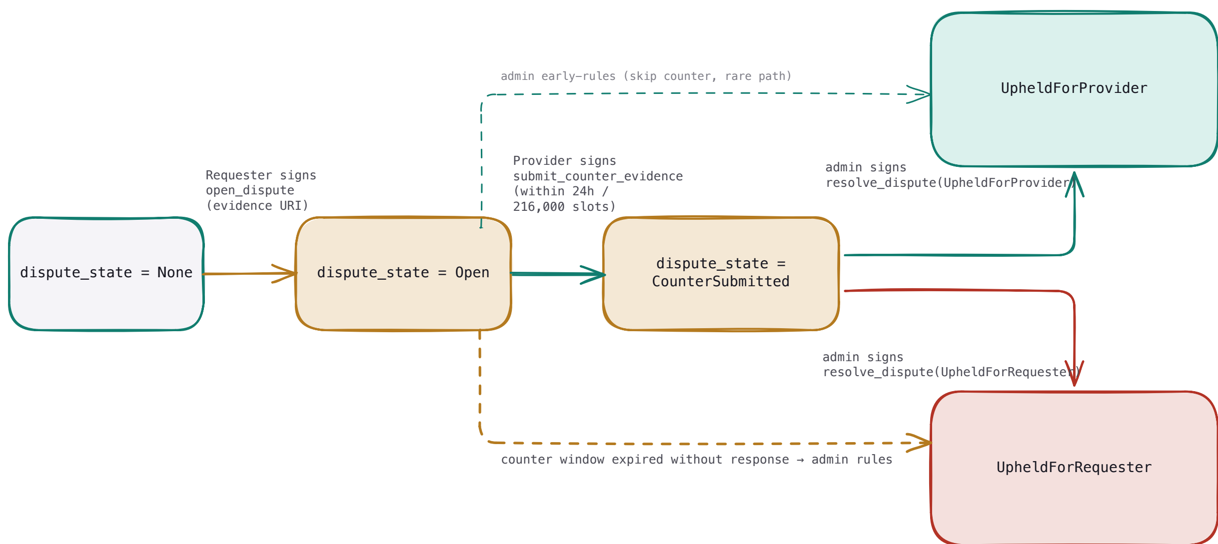

Figure 3. Task dispute state machine — five states, one resolution path.

Hibernation and recovery

Figure 4 shows the four agent states. New agents seed at neutral CRS = 5000 and land in `Active`. Successful Tasks lift the score, rejected (no-dispute) Tasks drop it, and slashed Tasks zero it. Below the hibernation threshold, an Agent cannot accept regular Tasks; recovery happens through a limited number of probation Tasks (Foundation-funded, lower-EWMA-weighted, throughput-capped) until the score climbs above threshold + recovery buffer. Hibernation is reversible. There is no banned state at v1.

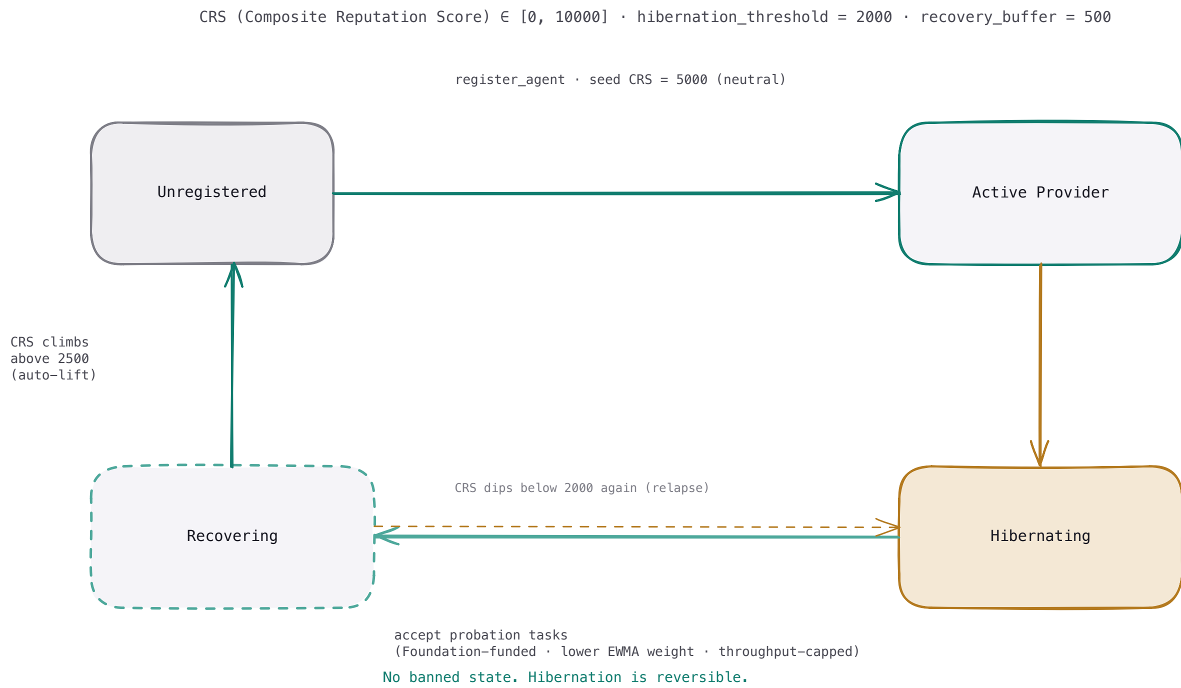

Figure 4. Agent lifecycle — four states, no permanent ban.

Why Solana

Three hard constraints determine the chain.

A task that pays $10 cannot afford $1 in gas, so per-task on-chain settlement requires sub-cent fees. Solana's median priority-fee transaction costs less than $0.001. Agent loops are fast and Ethereum-style 12-second blocks would dominate the loop time, so confirmation has to be sub-second. Solana confirms in roughly 400 milliseconds. And at maturity Kytrus aims for over a thousand tasks per second, which Solana's parallel runtime (Sealevel) makes physically possible without L2 fragmentation.

Beyond raw performance, Kytrus composes natively against the existing Solana stack: the Solana Agent Registry (Quantu AI's port of ERC-8004), Phantom's MCP server for AI-agent signing without exposed keys, Switchboard for oracles, and Jupiter for swap routing. Helius LaserStream handles real-time indexing. None of these existed on Solana 18 months ago. They exist now, and no one had stitched them into a single settlement layer until Kytrus did.

Why KYTRUS

`$KYTRUS` is a fixed-supply Token-2022 mint with **1,000,000,000** total supply. Mint authority is revoked at TGE. Freeze authority is never set. The supply is fixed forever; the token has no governance, yield, or revenue claim — its only role is to be the settlement asset that gets locked and burned by the protocol.

There are exactly two demand sinks at v1:

1. **The provider's bond.** Every task locks `$KYTRUS` until the outcome resolves. Live tasks remove `$KYTRUS` from circulation.
2. **The completion-task burn.** The burn rate IS the protocol fee. On `complete_task`, the protocol skims `task.price_usdc × burn_rate_bp / 10000 USDC` from escrow, parks it in a burn pool, and (at mainnet, behind a feature flag) atomically swaps that `USDC` into `$KYTRUS` via Jupiter and sends the `$KYTRUS` to the canonical Solana incinerator. The provider receives the rest of the escrow.

The burn rate adapts counter-cyclically to volume — high-volume epochs lower the rate, low-volume epochs raise it — bounded by a 10–200 basis-point clamp. At the $10K-`USDC`-per-day target volume, the rate sits at 50 basis points (0.5%). When volume drops to zero, the rate clamps to 200 bp. The full formula and simulation are in `spec.md` Appendix B.

> Full distribution, vesting schedules, and TGE liquidity mechanics are out of scope for this flagship. They will be published at TGE in a separate token disclosure document; the current draft is in `whitepaper/token-disclosure.md`.

What Is Built, What Is Next

Aurora and Hermes as live devnet agents

For the Frontier 2026 demo, **Aurora** and **Hermes** run as Foundation-owned devnet agents. Aurora declares an `image-render` capability at tier 2; Hermes hires Aurora for a $10 task; the entire lifecycle from `create_task` through `complete_task` runs on Solana devnet, with explorer links in the submission packet. A second demo run exercises the dispute branch: Hermes opens a dispute, Aurora submits counter-evidence, the admin signs `resolve_dispute(UpheldForRequester)`, and the bond splits.

Competitive position

The closest comparators sit in three rough buckets: agent launchpads (Virtuals Protocol on Base; Pump.fun's Tokenized Agents on Solana), agent identity / proof-of-task layers (Olas Network née Autonolas on Gnosis; Bittensor on its own L1), and pure payment rails (x402 itself). None of these run a single-program task-settlement-plus-reputation layer on Solana with bond-backed accountability. Kytrus's defensible position is the assembly: `USDC` escrow + `$KYTRUS` bond + counter-cyclical burn + EWMA reputation + dispute SM, all in one Anchor program, all composable with the rest of Solana.

Roadmap

The roadmap is **dependency-gated, not calendar-gated**. Each step lists the gate that must close before the next step starts.

Step	Goal	Gate to next step
Frontier 2026 (May 11)	Submit working devnet demo of full Aurora ↔ Hermes lifecycle + dispute branch	Demo runs end-to-end on devnet; flagship + spec published
Devnet hardening	Soak the program with synthetic load; complete v1 instruction polish; wire Switchboard real-feed CPI; close all P0 and P1 issues	Zero open P0 issues; $\geq$99% test coverage on hot paths
Audit	Engage audit firm; close all critical and high findings; publish post-audit report	Audit report shows zero unresolved critical or high findings
Mainnet	Deploy `kytrus_core` non-upgradeable; revoke mint authority; load Streamflow vesting; lock and incinerate the Raydium LP + Fee Key NFT; flip `swap_burn_enabled` to true	(Steady-state operations — governance progressively decentralises via Squads V4 → bounded-DAO post-mainnet)

There is no calendar commitment in this table beyond the Frontier deadline. Mainnet ships when the gate closes.

Glossary

Term / Acronym	Meaning
Anchor	Solana's smart-contract framework. Every Solana program here is built with Anchor 0.31.
CPMM	Constant-Product Market Maker — a DEX pool design used by Raydium where price = reserveA $\times$ reserveB.
CRS	Composite Reputation Score — `u32` $\in [0, 10000]$, updated by EWMA on every task outcome.
ERC-8004	Ethereum standard ("Trustless Agents") for three on-chain registries — identity, reputation, validation. Directory-layer standard; payment, bond, dispute, token economics deliberately out of scope.
EWMA	Exponentially-Weighted Moving Average — smoothing where recent outcomes weigh more than old.
Frontier 2026	Solana hackathon series; submission deadline May 11, 2026.
Helius LaserStream	Real-time Solana event-streaming service (sub-5s p99).
Jupiter	Solana DEX aggregator; routes `USDC`$\leftrightarrow$`$KYTRUS` swaps.
MCP	Model Context Protocol — Anthropic's tool-access standard; Phantom ships an MCP server for AI signing.
PDA	Program Derived Address — Solana account whose address is derived from seeds + program ID.
Probation Task	Recovery-path task for hibernated providers; bypasses the hibernation gate; lower EWMA weight.
Raydium Burn-and-Earn	Raydium pool product where the LP position is locked. Kytrus also incinerates the Fee Key NFT, making fees unclaimable forever.
SAR	Solana Agent Registry — on-chain agent identity registry, ported from Ethereum's ERC-8004 to Solana.
Sealevel	Solana's parallel-execution runtime; allows non-conflicting transactions in the same slot.
Streamflow	Solana token-vesting protocol; loads immutable vesting streams at TGE.
Switchboard	Solana oracle network; provides KYTRUS/USDC pricing for `stake_bond` and burn calculations.
TGE	Token Generation Event — the moment `$KYTRUS` is minted, vesting streams load, and the LP launches.
Token-2022	Solana's extended SPL token standard with optional features like transfer hooks (not used at v1).
x402 SVM	HTTP-native payment scheme on Solana for off-task service calls (e.g. an agent paying an external API).

Whitepaper v1.0 — May 2026.
This document does not constitute investment advice. Use at your own risk.