Stellar Oxide Gateway

Product Requirements Document

Version 1.0 | April 9, 2026 | Status: Implemented (Core) / In Progress (Ecosystem)

Category: Infrastructure / Paid Agent Services & APIs | Hackathon Alignment: Stellar x402

Executive Summary

Stellar Oxide Gateway is agent-native API monetization infrastructure built on Stellar. It transforms any HTTP endpoint into a pay-per-request service using x402-style payment challenges, Stellar-based settlement, and machine-readable responses designed for autonomous agents and programmatic consumers.

Unlike traditional API monetization approaches that rely on API keys, subscriptions, or centralized billing systems, Stellar Oxide Gateway enables direct machine-to-machine payments at the protocol level. Callers pay only when they use a service, with payments verified and settled on-chain in real-time.

The platform is designed as embeddable infrastructure rather than a standalone application. Developers can either deploy Stellar Oxide Gateway as a standalone gateway or integrate it directly into their existing Express applications as middleware.

Product Evolution:

- ✅ Current State (Implemented): Paid agent services / APIs — Provider SDK, payment gateway, route protection, intent flows, storage backends, client SDK
- 🚧 Next Phase (In Progress): Infrastructure / ecosystem tooling — Service publishing, discovery, registry compatibility, consumer resolution, trust foundations
- 🔮 Future Vision (Planned): Full ecosystem layer — Bazaar marketplace, reputation systems, multi-chain support, privacy pools, DeFi integrations

Key Value Propositions:

- Zero setup friction — No API key provisioning, no subscription management, no billing infrastructure
- Agent-native — Machine-readable payment requirements, structured receipts, automatic retry logic
- Embeddable — Mounts inside existing Express apps or runs standalone
- Blockchain-verified — Every payment is cryptographically verified and settled on Stellar
- Ecosystem-ready — Versioned manifests, discovery surfaces, registry export, consumer resolution
- Production-shaped — Multiple storage backends, health checks, graceful shutdown, comprehensive test coverage

Problem Statement

Current State

Developers building APIs for agents and automated systems face significant friction in monetization:

- API Keys require out-of-band provisioning, manual distribution, and centralized key management
- Subscriptions don't align with per-request agent workflows where usage is unpredictable
- Payment processors add latency, fees, and compliance overhead
- No standard protocol exists for machine-to-machine HTTP payments
- No discovery layer — Agents cannot programmatically find and evaluate paid services
- No ecosystem infrastructure — Each provider builds custom billing, discovery, and integration

Pain Points

For API Providers

- Building billing infrastructure is expensive and time-consuming
- Managing API keys and rate limits requires custom tooling
- Payment reconciliation across multiple consumers is manual
- No visibility into real-time revenue per endpoint
- No standard way to publish services for agent discovery
- No ecosystem tooling for service listing or reputation

For API Consumers (Agents)

- Acquiring API access requires human intervention
- Subscription costs are wasted when usage is sporadic
- No programmatic way to discover and pay for services
- Trust requires centralized intermediaries
- No standard contract for service metadata, pricing, or schemas
- Each provider has custom integration requirements

For Ecosystem Builders (Registries, Marketplaces, Platforms)

- No standard format for ingesting service metadata
- No compatibility layer for indexing providers
- No trust or reputation primitives
- No versioned contracts for ecosystem tooling

Market Opportunity

The Stellar x402 Hackathon categories explicitly call out three key areas:

- ✅ Paid agent services / APIs (Current Strength): Pay-per-token AI inference, pay-per-query search, financial market data, trading signals, web scraping, real-time news feeds, security vulnerability scanning, blockchain indexing
- 🚧 Infrastructure / ecosystem tooling (Next Phase): Bazaar-style discoverability, Bazaar-enabled Stellar facilitator, mainnet-ready facilitator infrastructure, service listing and discovery
- 🔮 Agent wallets, coordination, and commerce (Future): Agent wallet integrations, agent-to-agent communication and payments, rating/reputation/trust systems

Solution Overview

Core Concept

Stellar Oxide Gateway implements the x402 payment protocol on Stellar with a two-layer architecture.

Layer 1: Paid Agent Services / APIs (✅ Implemented)

The foundation layer that enables pay-per-request APIs:

- Agent calls protected endpoint
- Gateway returns 402 Payment Required with machine-readable requirements
- Agent signs Stellar payment transaction
- Agent retries request with x-payment header
- Gateway verifies payment locally (no third-party round-trip)
- Gateway settles payment on Stellar blockchain
- Gateway returns protected resource + structured receipt

Layer 2: Infrastructure / Ecosystem Tooling (🚧 In Progress)

- Provider publishes service metadata
- Service becomes discoverable via manifest + registry export
- Agents/platforms resolve service metadata programmatically
- Consumers select compatible services by category/network/asset
- Ecosystem tools (registries, bazaars, marketplaces) index services
- Trust and reputation signals emerge over time

Product Evolution Path

- ✅ Phase 1: Core Infrastructure (Complete) — x402 payment flow, Provider SDK, intent-based execution, multiple storage backends, service discovery surfaces, client SDK

- 🚧 Phase 2: Ecosystem Tooling (Current Focus) — Versioned provider manifest, registry-friendly export, consumer-side service resolution, provider/service identity metadata, trust foundations
- 🔮 Phase 3: Advanced Features (Planned) — Bazaar-style marketplace UI, registry crawler, provider reputation system, consumer wallet-connect flow, mainnet facilitator, privacy pools, multi-chain support, DeFi integrations

Roles and Actors

Implementer (Provider)

The implementer is the provider of the paid API. Examples include: startups exposing search, data providers exposing market data, AI products exposing inference, and platforms exposing paid internal tools.

Responsibilities: Integrate Stellar Oxide Gateway, configure protected routes with metadata, set pricing, provide merchant wallet, define backend handler or upstream proxy, publish provider and service identity.

Consumer

The consumer is the caller of the API. Examples include: agent frameworks, backend services, platform servers, and frontends with wallet-based payment.

Responsibilities: Discover compatible services, handle 402 challenges, sign and submit payments, parse receipts and responses.

Ecosystem Builder

Registries, bazaars, marketplaces, and platforms.

Responsibilities: Index services from manifest + registry export, filter and rank by compatibility, display provider metadata, enable trust and reputation signals (future).

Integration Modes

Mode 1: Standalone Gateway

Deploy **Stellar Oxide** as its own service that sits in front of backend APIs.

Best for: Hosted gateway deployments, reverse-proxy style monetization, simpler standalone operator flow, adding payment gating to existing services without code changes.

Mode 2: Embedded Provider

Mount Stellar Oxide Gateway routes directly inside an Express application using registerStellar Oxide GatewayRoutes(app, options).

Best for: Existing APIs, app teams, product integrations, tighter integration and shared application context.

Payment Models

Service-Paid Flow (Backend Wallet)

Best for: Agents, server-to-server usage, platform-sponsored usage.

Flow: Backend calls endpoint → receives 402 → signs payment with service wallet → retries with x-payment → response returned.

Security: Backend/service wallet secrets belong in env vars or secret manager. Never in frontend browser code.

User-Paid Flow (Wallet-Connect)

Best for: Consumer-facing apps, direct end-user payment flows, wallet-native agent products.

Flow: Frontend calls endpoint → receives 402 → user wallet signs payment → frontend retries with x-payment → response returned.

Security: Use wallet-connect style integrations. Do not put raw private keys in frontend/browser env.

Functional Requirements

FR-1: Payment Challenge Generation

Description: When a caller requests a protected endpoint without payment, the gateway must return a 402 status with machine-readable payment requirements.

Status: ✅ Implemented | Location: server/provider.js

Acceptance Criteria:

- Response status is 402 Payment Required
- Response includes accepts array with payment requirements
- Requirements include: payTo, maxAmountRequired, asset, resource, scheme
- Requirements include endpoint metadata: description, category, billingUnit, audience, tags, useCases
- Optional inputSchema and outputSchema for agent integration

FR-2: Payment Verification

Description: The gateway must verify signed Stellar payment payloads locally without requiring a third-party verification service.

Status: ✅ Implemented | Location: server/payments.js

Acceptance Criteria:

- Accepts payment payload in x-payment header (base64-encoded JSON)
- Verifies network, destination, amount, asset, and signature
- Verifies transaction has not been used before (nonce check)
- Verifies transaction has not expired (ledger bounds check)
- Supports both native XLM and Soroban token transfers

FR-3: Payment Settlement

Description: After verification, the gateway must submit the signed transaction to Stellar and return a structured receipt.

Status: ✅ Implemented | Location: server/payments.js

Acceptance Criteria:

- Submits native payments to Horizon, Soroban contract calls to RPC
- Returns transaction hash, ledger number and close time
- Returns payer and payee addresses and amounts in display and base units
- Returns explorer URLs for transaction and accounts
- Includes receipt version for forward compatibility

FR-4: Intent-Based Execution

Description: Support a two-phase flow where callers create a payment intent before committing payment.

Status: ✅ Implemented | Location: server/provider.js

Acceptance Criteria:

- POST /intents creates intent with endpoint and query
- Returns intent ID and payment requirements
- POST /intents/:id/execute executes behind x402 payment gate
- Intent lifecycle: pending → paid → executed (or failed)
- Execution follows stored policy (no re-evaluation)

FR-5: Dynamic Pricing and Policies

Description: Routes must support dynamic pricing, conditional paywalling, and custom payment metadata.

Status: ✅ Implemented | Location: server/provider.js

- pricing function receives { req, query, config, endpoint } and returns price USD
- shouldRequirePayment function or boolean controls paywall per request
- paymentMetadata function or object adds custom fields to receipt
- Policy evaluation happens once per request (or intent creation)

FR-6: Upstream Proxying

Description: Routes without local handlers must proxy to upstream APIs.

Status: ✅ Implemented | Location: server/handlers/shared.js

- Route config accepts upstream: { url, headers }
- Gateway forwards request body plus query, endpoint, path, intentId
- Payment verification happens before proxying

FR-7: Storage Abstraction

Description: Intent and usage data must be persistable across multiple backend types.

Status: ✅ Implemented | Location: server/intents.js, server/logger.js, server/postgres.js

- Supports memory, file, SQLite, and Postgres storage
- All backends implement healthCheck() and close()
- Postgres adapters lazy-initialize and auto-create tables
- Storage failures return 503 for dependency errors

FR-8: Service Discovery

Description: Providers must expose machine-readable metadata for ecosystem tooling, registries, and agent platforms.

Status: ✅ Implemented | Location: server/provider.js

- GET /.well-known/Stellar Oxide Gateway.json returns versioned manifest
- GET /registry/export returns flattened listing with filters
- GET /capabilities returns endpoint catalog with payment metadata
- GET /discovery/resources returns paginated resource list
- Registry export supports ?category=, ?tag=, ?audience= filters

FR-9: Client SDK

Description: Consumers need a drop-in client that handles the full 402 → sign → retry flow automatically.

Status: ✅ Implemented | Location: client/payFetch.js

- payFetch(url, options) wraps standard fetch
- Detects 402, builds and signs Stellar payment, retries with x-payment header
- Supports both native XLM and Soroban tokens
- Auto-funds testnet accounts in demo mode

FR-10: Service Resolution

Description: Consumers need programmatic discovery of Stellar Oxide Gateway services.

Status: ✅ Implemented | Location: client/lib/service-resolution.js

- resolveStellar Oxide GatewayService(baseUrl) fetches all discovery surfaces in parallel
- selectStellar Oxide GatewayRoute(service, criteria) filters routes by id, category, method, tag, audience

FR-11: Operational Readiness

Description: Providers must expose health and readiness probes for production deployment.

Status: ✅ Implemented | Location: server/provider.js

- GET /health returns liveness status
- GET /ready returns readiness status with storage health checks, 503 if unhealthy
- Provider exposes getReadinessReport() and close() for graceful shutdown

FR-12: Usage Analytics

Description: Providers must log requests and expose aggregated revenue stats.

Status: ✅ Implemented | Location: server/logger.js, server/provider.js

- Every request is logged with endpoint, query, timestamp, payment, intentId, flow
- GET /stats returns total requests and total revenue
- Stats computed from configured usage store across all backends

Technical Specifications

Supported Networks

Network ID	Stellar Network	Status

stellar-testnet	Testnet	✅ Supported
stellar	Mainnet	✅ Supported
base-sepolia	Base Sepolia	🔮 Future
base	Base Mainnet	🔮 Future
solana-devnet	Solana Devnet	🔮 Future

Supported Assets

Asset Type	Example	Verification Method
Native XLM	"native"	Horizon payment operation
Soroban Token	"C..." (contract address)	RPC contract call simulation
USDC (testnet)	Built-in registry	Soroban transfer

Storage Backends

Type	Use Case	Initialization	Health Check
memory	Testing, ephemeral	Immediate	Always healthy
file	Single-process, simple	Immediate	File access check
sqlite	Local persistent	Auto-create tables	SELECT 1 query
postgres	Production	Lazy init + auto-create	SELECT 1 query

Configuration

Required Environment Variables:

```
X402_NETWORK=stellar-testnet          # or stellar
WALLET_ADDRESS=G...                   # Merchant public key
```

Key Optional Environment Variables:

```
PORT=3000
GATEWAY_URL=http://localhost:3000
X402_ASSET=USDC                       # or native, or C... contract address
STELLAR_SECRET_KEY=S...               # Payer secret key
AUTO_FUND_TESTNET_ACCOUNTS=true       # Auto-fund in demo mode
```

Use Cases

Use Case 1: Paid Search API for Agents

Actor: Research agent | Goal: Pay per query for web search results

Flow: Agent calls POST /search → receives 402 with $0.05 requirement → signs payment → retries → gateway verifies, settles, returns results + receipt → agent logs receipt for accounting.

Template: examples/paid-search-provider.js

Use Case 2: Pay-Per-Token AI Inference

Actor: AI agent platform | Goal: Charge per inference request with dynamic pricing based on model size

Flow: Platform mounts Stellar Oxide Gateway routes for /inference → uses pricing function based on model parameter → caller creates intent → pays against intent → inference executes and returns result.

Template: examples/paid-inference-provider.js

Use Case 3: Freemium API with Conditional Paywall

Actor: SaaS platform | Goal: Free tier for pro users, paid for free users

Flow: Platform checks x-plan header in shouldRequirePayment → pro users bypass → free users receive 402 and must pay → payment metadata includes tenant ID for accounting.

Use Case 4: Market Data Provider

Actor: Financial data provider | Goal: Monetize real-time market data per query

Flow: Provider exposes /market-data → dynamic pricing based on data freshness and asset type → agents pay per query → provider returns structured market data + receipt.

Template: examples/paid-market-data-provider.js

Use Case 5: Web Scraping Service

Actor: Data extraction service | Goal: Charge per scraping job

Flow: Service exposes /scrape → pricing based on target URL complexity → agents submit scraping requests with payment → service returns extracted data + receipt.

Template: examples/paid-scraper-provider.js

Non-Functional Requirements

NFR-1: Performance

- Payment verification must complete in <100ms for native payments
- Payment verification must complete in <500ms for Soroban payments
- Storage operations must not block request handling
- Postgres adapters must use connection pooling

Status: ✅ Implemented (connection pooling via pg package)

NFR-2: Security

- Private keys must never be logged or exposed in responses
- Nonce reuse must be prevented (in-memory set)
- Transaction expiry must be enforced (ledger bounds check)
- Payment amounts must be verified in base units (no floating point)
- SQL injection must be prevented (parameterized queries)

Status: ✅ Implemented

NFR-3: Reliability

- Storage failures must return 503 (not 500)
- Intent execution must be idempotent (status check before execution)
- Concurrent intent execution must be prevented (in-memory lock)

Status: ✅ Implemented

NFR-4: Observability

- All errors must include actionable messages
- Payment verification failures must include specific reason codes
- Readiness checks must report per-backend status

Status: ✅ Implemented

Production Readiness

Current state: Development/staging quality — strong local validation and unit/integration tests. Not yet sufficient for production deployment with high confidence.

Priority Work for Production

- Priority 1: End-to-End Integration — HTTP integration tests, real 402 → pay → retry → verify → settle → response flow, cover all public endpoints
- Priority 2: Database and Storage Confidence — Real Postgres integration tests, schema creation verification, degraded-dependency tests
- Priority 3: Payment and Blockchain Confidence — Staging tests with real Stellar testnet payments, validate settlement receipts, test native XLM and Soroban paths
- Priority 4: Security Hardening — Review payment verification for tampering, add abuse tests (replay, duplicate execution, malformed headers), add request/body size limits
- Priority 5: Concurrency and Race Conditions — Test simultaneous execution, concurrent paid-route request tests, shutdown-during-request tests

Appendix A: Default Endpoints

Endpoint	Method	Path	Category	Base Price	Description
ai	GET	/ai	ai-inference	$0.02	AI-related queries
data	GET	/data	data-api	$0.01	Dataset queries
compute	GET	/compute	compute	$0.03	Processing tasks

All default endpoints add a $0.01 surcharge for queries over 20 characters.

Appendix B: Package Exports

Main package exports include:

- registerStellar Oxide GatewayRoutes, createStellar Oxide GatewayApp, createStellar Oxide GatewayProvider, validateProviderOptions
- NETWORK_IDS, SUPPORTED_NETWORK_IDS, isSupportedNetworkId, CONTRACT_VERSIONS
- payFetch, resolveStellar Oxide GatewayService, selectStellar Oxide GatewayRoute

Server utilities: createPaymentContext, loadGatewayConfig, validateGatewayConfig, requirePayment, requirePaymentWith, createIntentStore, createUsageStore

Pricing utilities: getPriceUsd

Client utilities: payFetch, fetchStellar Oxide GatewayManifest, fetchStellar Oxide GatewayCapabilities, fetchStellar Oxide GatewayRegistryExport, fetchStellar Oxide GatewayDiscovery

Appendix C: Paid Agent API Templates

1. Paid Search Provider

File: examples/paid-search-provider.js

Use Case: Pay-per-query web search for research agents and workflow enrichment

Features: Dynamic pricing by query complexity, structured search results, agent-friendly schemas

2. Paid Market Data Provider

File: examples/paid-market-data-provider.js

Use Case: Real-time financial market data, trading signals, news feeds

Features: Per-query billing, asset-specific pricing, structured market data responses

3. Paid Scraper Provider

File: examples/paid-scraper-provider.js

Use Case: Web scraping, data extraction, content collection

Features: URL-based pricing, extraction result schemas, agent-consumable output

4. Paid Inference Provider

File: examples/paid-inference-provider.js

Use Case: AI inference, summarization, generation

Features: Model-based pricing, token counting, structured AI responses

Appendix D: Deployment Checklist

Pre-deployment:

- Set X402_NETWORK to target network
- Set WALLET_ADDRESS to funded merchant wallet
- Configure X402_ASSET (native or USDC)
- Set up Postgres database (if using)
- Configure DATABASE_URL connection string
- Test merchant wallet can receive payments
- Run yarn test:production on testnet

Deployment:

- Deploy with PORT and GATEWAY_URL configured

- Verify GET /health and GET /ready return 200
- Test 402 challenge generation
- Test payment verification with test wallet
- Monitor settlement success rate

Post-deployment:

- Publish manifest at /.well-known/Stellar Oxide Gateway.json
- Submit registry export to ecosystem directories
- Monitor GET /stats for usage
- Set up alerts for storage health failures
- Document service in provider metadata