Vextaris Documentation

Offline copy of **docs.vextaris.io**, generated 2026-11-16.

The website is the live version; this file is a snapshot of the same pages.

Contents

1. Getting Started with Vextaris
2. Architecture Overview
3. IntentVM (U.S. Patent App. 19/571,463)
4. VIP -- Vextaris Intent Programs
5. HyperSync Consensus
6. Dragonfly Stream -- Mempoolless Transaction Pipeline
7. ElasticState -- Intelligent 3-Tier Storage Engine
8. VIDA -- Sovereign Digital Identity
9. VSA v2 -- Quantum-Hardened Smart Accounts
10. Atomence -- Cross-Chain Escrow Swaps
11. VSC Standards -- Native Building Blocks
12. VSC-8: Protocol-Native Stablecoin Standard
13. VSC-20: Programmable Staking Pools
14. Vextaris Token Metadata Standard
15. VSC-88: Protocol Governance
16. VSC-88: Multi-Sig Wallets
17. VSC-99: DAO Program
18. VexBridge -- Cross-Chain Verification
19. Vextaris Validator Setup Guide
20. Validator Server Hardening Guide
21. Validator Reputation System
22. VexVisor -- Upgrade Governance Module
23. Vextaris SDK & Wallet Developer Guide
24. RPC Reference
25. CLI Reference
26. IntentVM Developer Guide
27. Building with VIPs (Vextaris Intent Programs)
28. VSC-71 Synthetic Asset Integration Guide
29. The Vextaris Ecosystem
30. Vextaris Testnet Incentive Program
31. VexFi — Financial Infrastructure
32. Vexalus — Supply Chain Intelligence
33. VNS — Vextaris Name Service
34. Bushido Passport -- Universal Gamer Identity
35. Vextaris Testnet Reference
36. Vextaris Load Test Results
37. Vextaris Keypair and Recovery Reference
38. Network Defense Model
39. Security Audit Report
40. Glossary
41. Vextaris Blockchain Technical Whitepaper

Getting Started with Vextaris

Vextaris is a next-generation Layer 1 blockchain built from the ground up to solve real problems: quantum vulnerability, cross-chain fragmentation, developer complexity, and bridge security. At its core it is a **Proof-of-Stake chain with Byzantine-Fault-Tolerant (BFT) consensus** — blocks reach **instant finality** (settled the moment they're produced, with no reorgs and no "wait for N confirmations"), validators stake and can be **delegated** to, and rewards are **quality-weighted** (paid for reliable, useful work — not stake alone). Post-quantum readiness is one property of that system, not the whole story. It runs a live testnet across 7 validators on 4 continents, relaunched on BFT consensus (fresh genesis, July 2026). The Vextaris ecosystem extends beyond the core chain into financial infrastructure (VexFi), supply chain intelligence (Vexalus), sovereign digital identity (VIDA), community governance (DaimyoDAO), and human-readable identity (VNS).

The Eight Pillars

Vextaris is built on **eight core pillars** that work together as a unified system, plus an expanding ecosystem of applications:

1. **IntentVM** (U.S. Patent App. 19/571,463) -- 1 intent, 1 signature, N operations. Declarative execution.
2. **HyperSync Consensus + Vexcel** -- BFT Proof-of-Stake with **instant finality** (no reorgs). Bonded + delegated staking; **quality-weighted** rewards and **performance-weighted** leader selection (uptime/reliability/tenure, not stake alone); transactions routed **directly to the block producer** (no mempool limbo); plus jailing, slashing evidence, and a 7-factor reputation score. Full validator lifecycle with epoch-based activation/exit.
3. **VexBridge** (U.S. Patent App. 63/987,929) -- Cross-chain verification through an attester quorum. Adopted mainnet principle: *nothing comes to Vextaris that can't leave* -- no unbacked bridged tokens (planned for mainnet). Inbound deposits are paused on testnet while attestation moves to the quorum.
4. **VSA v2** -- Smart accounts with Ed25519 + Dilithium3 dual-signature quantum hardening. Includes VNS (`.vex` name service).
5. **VexForge** -- Ten token standards (VSC-7, VSC-8, VSC-20, VSC-21, VSC-55, VSC-88, VSC-99, VSC-LAUNCH, VSC-REP, VSC-SAFE) with protocol-level anti-rug protection, programmable staking pools, and on-chain DAOs.
6. **VexiDEX** -- Native on-chain AMM with constant-product pools and LP token management.

7. **ElasticState** -- Intelligent 3-tier storage engine (Hot/Warm/Cold) with automatic data lifecycle management.
8. **Atomence** (U.S. Patent App. 63/998,160) -- Cross-chain escrow swaps. Registered market makers pay on the destination chain, settlement is proven by an attester quorum, and the user can always reclaim if the market maker does not settle within the 1-hour window.

Ecosystem layers built on the pillars:

- **VIDA** -- Sovereign digital identity: 4-tier issuer registry, soulbound credentials, W3C Verifiable Credentials export, did:vex: DID method. 18 credential types, birth-to-legacy lifecycle.
- **VIP** -- Vextaris Intent Programs: declarative on-chain programs that replace smart contracts for common patterns. No compiler, no bytecode. 30 composable operations.
- **VexFi** -- Protocol-level financial infrastructure: payments, trading, staking, custody vaults, and natural-language portfolio management.
- **Vexalus** -- Supply chain intelligence: VSC-21 NFT shipments, VSC-88 custody handoffs, and BLAKE3 document attestation — ~$0.013 per complete lifecycle.
- **VNS** -- Vextaris Name Service: permanent `.vex` names, NFT-backed, no renewal fees.
- **DaimyoDAO** -- Native DAO launcher for VSC-99 governance. Create, browse, vote, and manage DAOs on Vextaris.
- **Bushido Passport** -- Universal gamer identity: soulbound profiles, cross-game achievements, and item marketplace.

Current Status

- **Network:** Live testnet across 7 validators on 4 continents, 3 seed nodes, relaunched on BFT consensus (fresh genesis, July 2026), sub-second to ~1.5s blocks
- **Performance:** Instant, deterministic BFT finality (a committed block is final — no confirmations) and a high-throughput native execution engine. A sustained-TPS benchmark on the current consensus is in progress; we do not publish throughput numbers we have not re-verified. On-chain binary upgrades delivered via VexVisor.
- **Codebase:** 15+ Rust crates, 124 operation discriminants (disc 0-123), 110+ RPC endpoints, 354 passing tests
- **Security:** Signatures required on every transaction (unsigned transactions refused network-wide since October 2026; canonical signing), pre-flight balance validation (GBI), per-sender spam throttling, tombstone mechanism, correlation penalty, three-tier key separation, transfer guards, post-quantum signatures

- **Frontends:** 16+ deployed applications — wallet, explorer, token studio, DEX, faucet, AI assistant, Atomence, VexFi, Vexalus, VNS, IntentVM demo, DaimyoDAO, VexVisor validator portal, docs, and more

Key Links

Resource	URL
Wallet	wallet.vexspark.com
Explorer	vexscan.io
Token Studio	vexforge.xyz
DEX	vexidex.com
Faucet	vexswap.xyz
Atomence	atomence.com
VexFi	vexfi.com
Vexalus	vexalus.com
VNS	vns.name
DaimyoDAO	daimyodao.com
VexVisor	vexvisor.com
IntentVM Demo	intentvm.xyz
VexAI	vexai.xyz
GitHub	github.com/vexidus-labs

Quick Navigation

- **New to Vextaris?** Start with the Architecture Overview to understand how the system works.
- **Want to run a validator?** Follow the Validator Setup Guide.

- **Building an app?** Check out the SDK Guide and RPC Reference.
- **Governance and multi-sig?** See Protocol Governance and Multi-Sig Wallets.
- **Creating a DAO?** See VSC-99 DAO Program or launch one at daimyodao.com.
- **Running a validator?** Manage your node at vexvisor.com.
- **Using VexFi?** See VexFi — Financial Infrastructure for payments, portfolio management, and custody vaults.
- **Human-readable names?** See VNS — Vextaris Name Service to register your `.vex` name.
- **Testing on testnet?** See the Testnet Reference for chain details, system accounts, and RPC examples.
- **Read the full technical design?** The complete Whitepaper covers all eight pillars in depth.
- **Looking up a term?** The Glossary defines all Vextaris-specific terms, standards, and concepts.

Important: Vextaris is NOT Ethereum

Vextaris is its own L1 blockchain that is Ethereum-*compatible*. Key differences:

- **Native addresses use the `Vx` prefix**, not `0x`. Full 32-byte encoding with type-specific prefixes: `Vx0` (wallet), `Vx1` (token), `Vx2` (NFT), `Vx5` (multi-sig), `Vx7` (DAO), `Vx8` (pool), and more. See the Glossary for the complete list.
- `0x` only exists for EVM compatibility (MetaMask, `eth_*` RPC methods) and internal system addresses.
- **Signature scheme:** Ed25519 (not secp256k1/ECDSA)
- **Decimals:** VXS has 9 decimals (not 18 like ETH)
- **Transaction model:** Atomic bundles (multiple operations per signature), not single operations. Every bundle is signed client-side and submitted with `vex_submitBundle` -- validators refuse unsigned transactions.

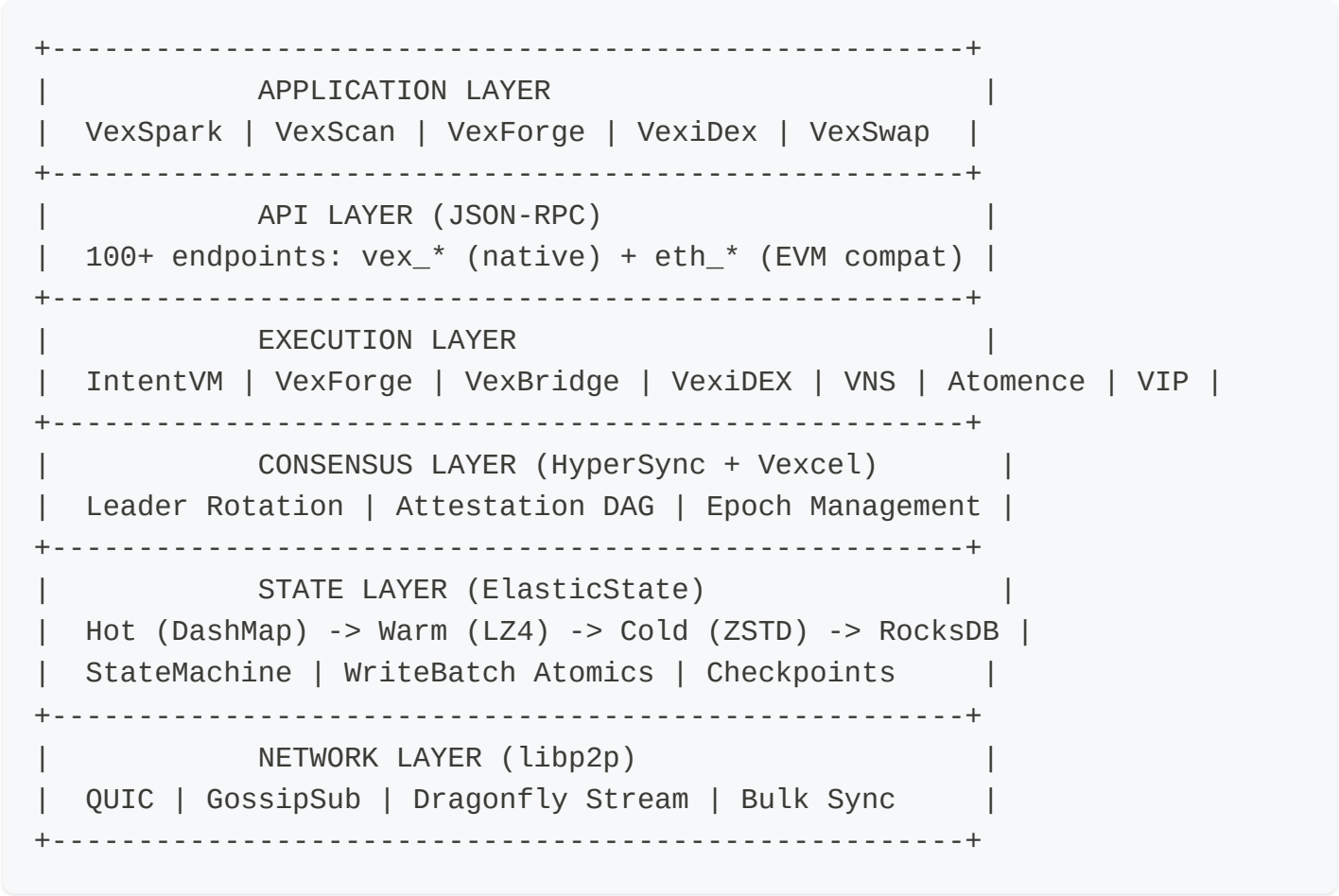
Architecture Overview

Vextaris is a Layer 1 blockchain built on eight integrated pillars. This page provides a high-level view of the system architecture, transaction lifecycle, and how the pillars compose.

The Eight Pillars

1. **IntentVM** (U.S. Patent App. 19/571,463) -- 1 intent, 1 signature, N operations. Declarative execution.
2. **HyperSync Consensus + Vexcel** -- Leader rotation with weighted selection, adaptive attestation DAG for global latency fairness, jailing, and reputation scoring.
3. **VexBridge** (U.S. Patent App. 63/987,929) -- Cross-chain verification: proves events on other chains through a quorum of registered attesters. Inbound deposits are paused on testnet; at mainnet VexBridge verifies, it does not mint copies of foreign tokens.
4. **VSA v2** -- Smart accounts with Ed25519 + Dilithium3 dual-signature quantum hardening. Includes VNS (`.vex` name service) as an identity extension.
5. **VexForge** -- Ten token standards (VSC-7, VSC-8, VSC-20, VSC-21, VSC-55, VSC-88, VSC-99, VSC-LAUNCH, VSC-REP, VSC-SAFE) with protocol-level anti-rug protection, programmable staking pools, and on-chain DAOs.
6. **VexiDEX** -- Native on-chain AMM with constant-product pools and LP token management.
7. **ElasticState** -- Intelligent 3-tier storage engine (Hot/Warm/Cold) with automatic data lifecycle management.
8. **Atomence** (U.S. Patent App. 63/998,160) -- Cross-chain escrow swaps. A bonded market maker pays the user on the destination chain; an attester quorum confirms the payment; otherwise the user reclaims the escrow plus a share of the market maker's bond.

System Layers



Implementation

- **Language:** Rust -- 15 crates, memory-safe, zero-cost abstractions
- **Database:** RocksDB with atomic WriteBatch commits (crash-safe, no partial state)
- **Storage:** ElasticState 3-tier engine -- Hot (DashMap, <1us), Warm (RocksDB LZ4), Cold (RocksDB ZSTD). Zero-copy checkpoints every 1K blocks.
- **Networking:** libp2p with QUIC transport (TLS 1.3, multiplexing over UDP)
- **State Root:** LtHash homomorphic accumulator -- O(1) per write, Blake3 finalization once per block. Same approach as Solana SIMD-0215.
- **Transaction Model:** Account-based with bundle semantics (atomic multi-operation)

Transaction Lifecycle

Dragonfly Stream (direct delivery):

```
1. User submits bundle (signed Ed25519)
      |
2. RPC validates signature + format
      |
3. Validator creates PQ seal (Dilithium3 signature over blake3
commitment)
      |
4. Sealed bundle added to local LeaderBuffer + forwarded to seed peers
      |
5. Seeds relay to all connected validators
      |
6. Elected leader drains LeaderBuffer, proposes block
      |
7. Bundles executed and committed atomically as a single batch
      |
8. State root recomputed and persisted
      |
9. Block gossiped to peers -> verified -> finalized
      |
10. Explorer indexes updated (RocksDB "explorer" CF)
```

There is no gossip mempool. All transactions flow through the Dragonfly Stream sealed pipeline. PQ sealing is always on. See Dragonfly Stream for details.

Wallet Compatibility

Wallet	Status	RPC Layer	Signature
VexSpark (native)	Live	`vex_*`	Ed25519
MetaMask	Live	`eth_*`	secp256k1/ECDSA
Phantom	Planned	`svm_*`	Ed25519 (same curve)

One Ed25519 keypair works on both Vextaris and Solana. The same key generates three address formats (Vx0, 0x, base58) that all resolve to the same internal account.

How the Pillars Work Together

The eight pillars are not isolated features -- they compose:

Example: "Swap 1,000 VXS for USDC and send it to `chris.vex`"

1. **IntentVM** parses the request into a composite intent (swap + transfer)
2. **VSA v2** verifies the single signature over the entire bundle, signed client-side and submitted with `vex_submitBundle` (VNS resolves `chris.vex` to the recipient's address)
3. **VexiDEX** routes the VXS to USDC swap through the best pool
4. **VexForge** enforces the token standards' rules on the transfer (e.g. VSC-8 blacklist and pause checks)
5. **HyperSync** includes the bundle in the next block (weighted leader selection)
6. **ElasticState** serves the accounts involved and persists the result

All of this happens atomically -- one intent, one signature, one block slot. If any step fails, the entire operation rolls back.

Modules

Module	Purpose
Core Types	Primitives, 124 transaction operation types (disc 0-123), addresses, signatures
State Machine	RocksDB-backed accounts, token registry, reputation, pools, validators
VXS Economics	Token constants, supply schedule, block rewards
HyperSync Consensus	Block production, finalization, Vexcel attestation DAG, voting, epoch management
RPC Server	JSON-RPC -- 100+ endpoints (vex_* + eth_* compatibility)
Node	Binary entry point -- genesis, P2P networking, block production
P2P Network	libp2p + QUIC transport, GossipSub, peer discovery
SDK	Wallet, validator, DEX client, intent parser, bundle builder
CLI	18 command groups, 100+ subcommands
Quantum Crypto	Ed25519 + Dilithium3 (post-quantum) keypairs

Module	Purpose
VexBridge	Cross-chain verification -- attester quorum, bridge registry, pause, VSC-SAFE safety registry
VexForge	Token validation (VSC-7, VSC-21, VSC-55 standards)
IntentVM	Intent-based execution engine (U.S. Patent App. 19/571,463)
ElasticState	3-tier hot/warm/cold state management
VIP	Vextaris Intent Programs -- declarative on-chain programs (no bytecode, no VM)
Atomence	Cross-chain escrow swaps -- registered market makers, settle deadline, attester-verified settlement

Network Parameters

Parameter	Value
Chain ID	1618032 (`0x18b070`)
Block time	~2s adaptive (500ms-12s range)
Max transactions/block	10,000 (configurable via `--max-txs-per-block`)
Transaction delivery	Dragonfly Stream (direct, PQ-sealed)
PQ seal algorithm	Dilithium3 (NIST Level 3) with Ed25519 classical fallback
Bundle TTL	120 seconds
P2P transport	QUIC (TLS 1.3 + multiplexing, UDP)
Gas price	10 nanoVXS/gas (configurable)
Auto-checkpoint interval	Every 1,000 blocks
Checkpoints retained	Last 5

IntentVM (U.S. Patent App. 19/571,463)

The Problem: Traditional blockchains require users to specify exact execution steps. A simple "swap then stake" requires 2 transactions, 2 signatures, and 2 consensus rounds.

The Solution: IntentVM accepts a single intent -- expressed as structured JSON or natural language -- and decomposes it into the optimal sequence of native operations, executed atomically in one block slot.

```
User: "swap 100 VXS for USDC and stake the rest"

IntentVM decomposes:
  1. Swap (VXS -> USDC via VexiDEX)
  2. Stake (remaining VXS to validator)

Result: 1 intent, 1 signature, 2 operations, 1 block slot
```

Key Advantage -- 1 Intent, 1 Signature, N Operations

On Solana or Ethereum, "swap then stake" = 2 transactions, 2 signatures, 2 consensus rounds. On Vextaris, it is 1 bundle, 1 signature, 1 round. This gives Vextaris 3-5x effective throughput per user action compared to per-transaction chains.

Multi-Input Atomic Swaps: IntentVM can process multi-input operations -- for example, "swap 10 USDC and 10 USDT for VXS" -- in a single atomic execution. On other chains this requires 2 separate transactions with slippage risk between them. On Vextaris, both inputs are converted in one block slot with unified slippage protection.

Provenance

IntentVM was first built and proven as an on-chain program on Solana, with real transaction proofs from a working prototype. It was then rebuilt as a core protocol feature on Vextaris -- not a

theoretical design, but a port of working code.

Supported Goals

Core Operations

- **Swap** -- Token exchange via VexiDEX pools with auto-routing
- **Stake** -- Delegate VXS to validators
- **Provide Liquidity** -- Add to AMM pools
- **Transfer** -- Send tokens to an address (Vx, 0x, or `.vex` name). VNS names like `cjf.vex` are resolved automatically at execution time.
- **Bridge** -- Cross-chain with tier-aware proof verification
- **Composite** -- Atomic multi-step. Supports N-input token consolidation: `"swap 10 USDC, 20 USDT, and 30 DAI for VXS"` creates N atomic swaps in one intent.
- **Program** -- Execute a registered VIP (Vextaris Intent Program). Developers register reusable, parameterized operation sequences that users invoke by name. No smart contracts needed.
- **Custom** -- VNS registration (`"register myname.vex"`), future LLM-parsed intents

Outcome-Based Intents

Users specify **what they want**, not the steps to get there. The planner reverse-engineers the optimal execution path.

- **Acquire** -- "Get me 10 SOL, max 200 USDC". Specify target asset + amount + optional cost constraint. The planner reverse-calculates the required swap input from pool reserves using AMM math, validates against your max cost, and executes atomically.
- **Liquidate** -- "Liquidate to 500 USDC, keep VXS". Specify target asset + amount needed. The planner queries your portfolio holdings, finds the optimal sell path (best swap rates first), and greedily sells until the target is covered. Exclude tokens you want to keep.
- **Rebalance** -- "Rebalance to 50% VXS, 30% USDC, 20% ETH". Reads current portfolio, computes delta trades from current allocation to target percentages, and executes the optimal swap sequence atomically.

Portfolio Intelligence

- **Allocate** -- "Put 100 VXS split 60/40 between staking and liquidity". Divides an amount across multiple operations by percentage.
- **Recurring (DCA)** -- "Buy 10 VXS with USDC every day for 7 days". Scheduled execution with configurable interval and count.
- **Conditional (Limit Orders)** -- "Swap 100 USDC for VXS if price below 0.15". Price-triggered execution using on-chain spot prices.

Synthetic Assets (VSC-71 -- Licensee Interface)

- **MintSynthetic** -- Tokenized stocks, commodities, indices (e.g. "buy 10 AAPL with USDC")
- **OpenPosition** -- Leveraged perpetual contracts (e.g. "open 5x long ETH perp with 100 USDC")
- **ClosePosition** -- Close an existing leveraged position

VSC-71 defines the IntentVM interface for synthetic assets. Oracle pricing, margin engines, and liquidation logic are supplied by licensees, who wire their own oracle partners and risk management systems. See the VSC-71 Integration Guide for implementation details.

Routing & Price Intelligence

Auto-Routing

When no direct liquidity pool exists between two tokens, IntentVM automatically discovers intermediate routes. For example, swapping DAI → SOL may route through DAI → VXS → SOL if no direct DAI/SOL pool exists.

The router evaluates all available intermediate tokens and selects the route with the deepest liquidity (highest minimum reserve across both legs). Slippage constraints are applied per leg to protect against multi-hop price impact.

Price Oracle (TWAP Foundation)

Every swap on VexiDEX records a cumulative price observation. This enables:

- **Spot pricing** -- Current pool ratio for any token pair
- **TWAP pricing** -- Time-weighted average price over configurable windows
- **Auto-routed pricing** -- Price quotes through intermediate pools when no direct pair exists

Price Impact Protection

Before execution, IntentVM checks that no single swap would drain more than 25% of any pool's reserves. This is tighter than the 50% circuit breaker in the swap handler, protecting users from excessive slippage on large orders. Simulation endpoints return price impact in basis points with human-readable warnings.

Constraint Enforcement

All constraints can be set programmatically (JSON) or extracted automatically from natural language:

Constraint	NL Example	Effect
Slippage	"with 3% slippage"	Rejects if output deviates beyond threshold
Deadline	"within 5 minutes"	Rejects if intent not executed before deadline
Max Fee	"max fee 50000"	Reverts if gas cost exceeds cap (in nanoVXS)
Min Output	"minimum 95 USDC"	Rejects if output below floor
Route Preference	Automatic / PreferDex / Custom	Controls DEX selection

Execution Previews

IntentVM provides three levels of pre-execution analysis:

Method	Purpose
`vex_getExecutionPlan`	Execution steps + gas estimate
`vex_simulateIntent`	Full dry-run: per-pool quotes, price impact (bps), balance checks, warnings
`vex_previewIntent`	Human-readable wallet confirmation: summary text, fee in VXS, slippage %, route type

Wallet developers should call `vex_previewIntent` before `vex_submitIntent` to show users a confirmation screen with estimated costs and warnings.

Multi-Language Support

The NL parser accepts intents in 7 languages. Non-English input is normalized to English before parsing. All intent types -- swaps, transfers, staking, outcome-based intents, and constraints -- work in every language.

Written number support: The parser converts written-out numbers to digits before processing: "swap five hundred VXS for ETH" becomes "swap 500 VXS for ETH". Supports compounds ("two thousand"), decimals ("one point five"), and multipliers ("a hundred").

Language	Swap Example	Buy Example (→ Acquire)
English	"swap 100 USDC for VXS"	"buy 10 VXS with USDC"
Spanish	"intercambiar 100 USDC por VXS"	"comprar 10 VXS con USDC"
Portuguese	"trocar 100 USDC por VXS"	"comprar 10 VXS com USDC"
French	"échanger 50 ETH pour VXS"	"acheter 10 VXS avec USDC"
German	"tauschen 100 USDC für VXS"	"kaufen 10 VXS mit USDC"
Italian	"scambiare 100 USDC per VXS"	"comprare 10 VXS con USDC"
Japanese	"100 USDCをVXSに交換"	--

Note: "buy" verbs produce **Acquire** intents (target output amount), while "swap" verbs produce **Swap** intents (input amount). See the Developer Guide for full verb and connector mappings.

Architecture

```
User Input (NL or JSON, any supported language)
    |
    v
Multi-Language Normalizer
    |  maps non-English verbs to English equivalents
    v
Constraint Extractor
    |  strips deadline, max_fee, slippage from text
    v
NL Parser / JSON Deserializer
    |  pattern matching → Goal enum
    v
IntentBuilder
    |  fluent API, builds Goal + Constraints
    v
RPC: vex_previewIntent (optional)
    |  wallet confirmation screen
    v
RPC: vex_submitIntent
    |  builds intent bundle, submits to leader
    v
Execution Engine
    |  decomposes Goal → native Operations
    |  auto-routing, price impact check
    |  constraint enforcement (deadline, fees, slippage)
    v
Block Execution (atomic)
```

Future: LLM Integration

The natural-language parser handles common intent phrasings out of the box. For more complex or free-form intents, integrate an LLM:

```
User: "Buy VXS with half my USDC and stake it with the highest-APY
validator"
  -> LLM extracts: Composite([
       Swap { from: USDC, to: VXS, amount: 50% of balance },
       Stake { token: VXS, amount: all, validator: highest_apy }
     ])
```

Integration path:

1. Client sends NL to LLM API (Grok, Claude, etc.)
2. LLM returns structured Goal JSON
3. Client submits structured JSON via `vex_submitIntent`

Developer Guide

For code examples, SDK usage, and RPC integration, see the IntentVM Developer Guide.

For VSC-71 synthetic asset integration (tokenized stocks, perpetuals), see the VSC-71 Integration Guide.

VIP -- Vextaris Intent Programs

The Problem: Traditional blockchains require developers to write, compile, and deploy smart contracts (Solidity on Ethereum, Rust on Solana) to create custom on-chain logic. This introduces bytecode verification headaches, compiler bugs, reentrancy exploits, and a steep learning curve.

The Solution: VIPs are **declarative composition templates** that sequence native Vextaris operations into reusable, parameterized programs. No bytecode. No compiler. No VM overhead. The program definition IS the source code -- written in JSON, readable by anyone, and statically analyzed for safety at registration time.

```
Developer writes:
  {
    "namespace": "defi",
    "name": "claim_and_stake",
    "steps": [
      { "op": "MintMultiToken", "args": { "recipient": "${sender}" } },
      { "op": "StakeToPool",    "args": { "amount": "${params.amount}" }
}
    ]
  }

User invokes:
  "run defi/claim_and_stake with amount=1000"

Chain executes:
  1. MintMultiToken (reward tokens to user)
  2. StakeToPool (auto-stake rewards)
  Result: 1 signature, 2 operations, atomic execution
```

Why VIPs Instead of Smart Contracts

Vextaris has **104+ native operations** -- token creation, NFT minting, AMM swaps, multi-sig custody, governance, staking, bridging, and more. These are not contract calls -- they are protocol-level primitives, already audited, gas-optimized, and battle-tested.

Smart contracts were invented because early blockchains had ~3 operations (send, receive, multisig). Developers needed a way to express complex logic. Vextaris already has the complex

logic built in. What developers need is a way to **compose existing operations** into reusable sequences.

Dimension	EVM Smart Contracts	Solana Programs	Vextaris VIPs
Format	Compiled bytecode	Compiled BPF/SBF	Declarative JSON
Compilation	Solidity -> solc -> bytecode	Rust -> cargo-build-sbf	**None required**
Verification	Decompile bytecode, match source	Verify .so against source	**Read the JSON** -- it IS the source
Safety	Reentrancy, overflow, delegatecall	CPI confusion, PDA collision	Static analysis of known operation patterns
Gas model	Opcode-level metering	Compute unit budgets	Sum of constituent operation costs (predictable)
Attack surface	Massive (entire VM + compiler)	Large (loader + CPI)	Minimal (template expansion only)

How VIPs Work

1. Registration

A developer writes a JSON program definition and submits it via `vex_registerProgram`. The chain:

1. **Validates** the JSON structure (params, steps, returns)
2. **Runs static safety analysis** (9 rules -- see Safety Engine below)
3. **Assigns a safety score** (0-100) and level (Verified/Standard/Warning/Blocked)
4. **Stores on-chain** with version number and namespace ownership
5. **Charges a registration fee** (50 VXS)

If any critical safety rule fails, the registration is **rejected** -- the program never reaches the chain.

2. Invocation

A user invokes a VIP via IntentVM by naming the program and supplying its parameters -- for example, running `defi/claim_and_stake` with an `amount` of `1000000000`. The invocation references the program by namespace and name and passes the required parameter values.

The chain:

1. **Loads** the program definition
2. **Checks** safety level (Blocked programs cannot execute)
3. **Validates** required parameters are provided
4. **Expands** the template into native operations
5. **Executes** all operations atomically (all-or-nothing)

3. Simulation

Before signing, users can dry-run any VIP via `vex_simulateProgram` to see exactly what operations will execute, estimated gas costs, and any warnings. This is the DYOR tool built into the protocol.

Safety Engine

Every VIP is analyzed at registration time. The safety engine enforces these rules:

Rule	What It Checks	Result If Failed
No Hardcoded Recipients	Value-moving operations must use `${sender}` or `${params.*}` as recipient	**REJECT**
No Hidden Approvals	Approve operations must have spender as `${params.*}` (user-provided)	**REJECT**
Allowed Operations Only	Only safe operations allowed in VIPs (no bridge admin, no key management)	**REJECT**
Expansion Limits	Max 50 expanded operations per invocation	**REJECT**
Gas Ceiling	Estimated gas must be below 5,000,000	**Warning**
Template Validation	All `${...}` expressions must reference declared params or prior steps	**REJECT**

Rule	What It Checks	Result If Failed
URI Sanitization	All URIs pass XSS/injection blocking (same as VSC-SAFE)	**REJECT**
Step ID Uniqueness	No duplicate step IDs, no forward references	**REJECT**
Namespace Rules	3-32 chars, alphanumeric + hyphens, reserved names require governance	**REJECT**

Safety Levels

Level	Score	Meaning
Verified	80-100	Passes all checks; author has strong reputation or governance audit
Standard	50-79	Passes all checks; normal author
Warning	25-49	Community-flagged or has suspicious patterns
Blocked	0-24	Failed analysis or governance-blocked; **cannot be invoked**

Community Flagging

Anyone can flag a VIP via `vex_flagProgram`. Each flag reduces the safety score. At 5+ flags, the score drops significantly. Governance can reclassify programs via proposals (same pattern as VSC-SAFE contract safety).

The safety registry is **governance-managed** -- new scam patterns, blocked addresses, and verified authors can be added over time via governance proposals. No binary upgrade required.

Template Expressions

VIPs use a restricted substitution syntax (NOT a programming language):

Expression	Meaning
`${sender}`	Address of the user invoking the VIP

Expression	Meaning
`${params.name}`	User-supplied parameter value
`${steps.id.result.field}`	Output of a previous step
`${item}`	Current element in a loop iteration
`${block.height}`	Current block height
`${block.timestamp}`	Current block timestamp

What is NOT supported (by design): arithmetic, conditionals, string manipulation, external calls, variable mutation, nested expressions. This restriction IS the safety guarantee.

Universal Data Anchoring

Alongside VIPs, Vextaris introduces the **Anchor** operation -- a lightweight way to write a hash to the chain without token semantics. An anchor records, under a chosen namespace, a key identifying the data, a content hash of the data, and an optional metadata URI (for example, an IPFS link).

- **5,000 gas** (10x cheaper than token mints)
- **Namespace-isolated** (partners cannot collide)
- **Queryable** via `vex_getAnchors` and `vex_verifyAnchor`
- **Universal** -- any application can use it for data attestation

Integration with IntentVM

VIPs are a first-class IntentVM goal type. This means VIPs can be invoked via natural language:

```
"run vexalus/create_shipment with contents=Electronics origin=Shanghai
destination=Rotterdam"
```

The IntentVM parser recognizes the `run <namespace>/<name>` pattern, extracts parameters, and creates a program-invocation intent which the chain expands into native operations.

RPC Methods

Method	Purpose
`vex_registerProgram`	Register a new VIP
`vex_getProgram`	Get VIP definition (params, steps, safety score)
`vex_listPrograms`	Browse VIPs by namespace, author, or safety level
`vex_simulateProgram`	Dry-run expansion (see operations before signing)
`vex_getProgramSafety`	Full safety analysis with flag details
`vex_flagProgram`	Flag a VIP for community review
`vex_anchor`	Anchor a data hash on-chain
`vex_getAnchors`	Query anchored data
`vex_verifyAnchor`	Verify a hash matches an anchor

CLI Commands

```
vexidus vip register --program '{"namespace":"myapp",...}' --from
<address>
vexidus vip show <namespace> <name>
vexidus vip list --namespace myapp
vexidus vip simulate <namespace> <name> --params '[["key","value"]]'
vexidus vip safety <namespace> <name>
vexidus vip flag --namespace myapp --name risky-prog --reason "suspicious
transfers" --from <address>
vexidus vip anchor --namespace myapp --key <hex> --hash <Vxh...> --from
<address>
vexidus vip get-anchors <namespace> <key>
vexidus vip verify <namespace> <key> <expected_hash>
```

Safety Philosophy

Vextaris implements multi-layered safety features for VIPs: static analysis at registration, runtime validation at execution, author reputation scoring, and community-driven governance oversight. These features are designed to detect and deter common attack patterns.

No automated system can guarantee the safety of every program. Users should always review the execution plan via `vex_simulateProgram` before signing, verify the program author's reputation, and exercise their own judgment. Always DYOR.

Safety features, not safety promises. We build the tools. We encourage DYOR. We make it harder for bad actors. We do not claim to make it impossible.

HyperSync Consensus

HyperSync is the consensus protocol powering Vextaris. It is a **Byzantine-Fault-Tolerant (BFT) Proof-of-Stake** protocol: validators bond VXS (and can be **delegated** to by any holder), take turns proposing blocks under weighted leader rotation, and vote to commit. Reaching a **2/3+ supermajority makes a block final immediately** — Vextaris has **instant finality**, so there are no reorganizations and no need to wait for confirmations. This is a categorical difference from Proof-of-Work and longest-chain Proof-of-Stake chains, where settlement is only probabilistic.

How It Works

1. **Linear leader slots:** Each leader produces 5 consecutive blocks before rotation. Selection is deterministic and verifiable, derived from the slot window with a stake-weighted random pick. This reduces handoff overhead across geographically distributed validators and stabilizes direct-to-leader transaction routing.
2. Weight = pure stake-weighted (deterministic, no performance modifiers)
3. Leader proposes block containing up to 50,000 transactions (100,000 in benchmark mode)
4. Block is gossiped to all peers via GossipSub `"blocks"` topic (QUIC transport, TLS 1.3) using Borsh binary encoding with automatic JSON fallback
5. Validators verify proposer signature (Ed25519) and execute transactions via optimistic parallel execution
6. Block finalized after gossip consensus

Block Production

- Adaptive block intervals (~2s nominal, 100ms-12s range, configurable via `--block-time`)
- Pressure-aware timing: 2s normal, 250ms high load, 100ms extreme load
- Up to 50,000 transactions per block (100,000 with `--benchmark-mode`)
- **Vexcel Attestation DAG:** When the leader is slow, non-leader validators produce lightweight attestation blocks (zero transactions) that prove liveness without entering the canonical chain. When the leader is fast, the chain is purely linear with zero overhead. This adaptive mechanism ensures global latency fairness -- high-latency validators are never disadvantaged by geography.

Leader Skip & Failover

When a leader misses its block slot (network latency, temporary downtime), the protocol recovers automatically:

- **Staggered failover:** Backup validators take over with position-based timeouts (5s base + 3s per position). Only one backup fails over at a time, structurally preventing multi-way forks. Failover works within leader slots -- if a leader goes offline mid-slot, the backup produces the remaining blocks in the slot.
- **Leader skip rotation:** Validators that miss 3 consecutive blocks are temporarily excluded from the leader rotation (in-memory only). They must produce 2 consecutive blocks to rejoin. This prevents a high-latency validator from repeatedly stalling the chain.
- **Non-punitive:** Skip rotation does not affect rewards, staking, or on-chain state. It is purely a block timing optimization. Only double-signing (provable malicious behavior) triggers jailing.
- **Zero-stall guarantee:** Maximum delay from any single validator failure is 5 seconds (one failover timeout). The chain self-recovers without human intervention.

Wire Protocol

Block gossip and P2P sync use a dual-format encoding system for safe rolling upgrades:

- **Borsh binary encoding** (default when `--borsh-wire` is enabled): 5-10x smaller blocks on the wire, ~10x faster serialization than JSON. Messages prefixed with `VX` (0x56, 0x58) magic bytes.
- **Automatic format detection:** Nodes auto-detect Borsh vs JSON on every received message. New and old binary versions can coexist during upgrade transitions.

Validator Economics

Parameter	Value
Minimum stake	1,000 VXS
Unbonding period	21 days
Max active validators	100 (configurable)

Parameter	Value
Epoch duration	1 hour (1,800 blocks)
Commission range	0-50%

Block Rewards

~65M VXS emitted over 10 years (~4% of total supply):

Period	Annual Total	Foundation (15%)	Validator Pool (85%)	Per-Block @2s (1.0x density)
Year 1-2	13M VXS/yr	1.95M	11.05M	~0.700 VXS
Year 3-5	6.5M VXS/yr	0.975M	5.525M	~0.350 VXS
Year 6-10	3.9M VXS/yr	0.585M	3.315M	~0.210 VXS
Year 11+	0	0	0	0

Per-block reward is time-weighted: `reward = annual x block_interval / seconds_per_year`. Transaction density multiplier scales the validator portion: 0.5x (empty) to 2.0x (1000+ txs).

Transaction Fees

- 80% of fees go to the block proposer, 20% to Foundation Treasury (no burn)
- Gas price: 10 nanoVXS per gas unit (configurable via `--gas-price`)
- Typical transfer cost: ~~0.00021 VXS (~~$0.0002 at $1/VXS)

Leader Skip Rotation

When a validator misses 3 consecutive leader slots, it is automatically skipped in leader rotation until it produces a block again. This prevents offline validators from degrading block time -- instead of waiting 10+ seconds for failover on every missed slot, the remaining validators continue producing at full speed.

- **Trigger:** 3 consecutive missed blocks

- **Recovery:** Produce 2 consecutive blocks to rejoin rotation
- **Deterministic:** All nodes compute skip status identically from the same block data
- **Failsafe:** If all non-jailed validators are skipped, the full non-jailed set is used (chain never stops)
- **Upgrade grace:** Skip is not applied during the 100-block upgrade grace period

With 7 validators and 1 offline, block time stays low instead of degrading, as the remaining validators continue producing at full speed.

Linear Leader Slots

An earlier iteration of Vextaris used **linear leader slots**, where each leader produced 5 consecutive blocks before rotation -- the same approach used by Solana and other high-performance blockchains. This mechanism was subsequently superseded by Malachite BFT consensus; the design and its rationale are retained here for reference.

Benefits over per-block rotation:

- **Reduced handoff overhead:** Leader changes once every ~13 seconds instead of every ~2.5 seconds
- **Stable routing:** Dragonfly Stream's direct-to-leader routing target is stable for the entire slot
- **Geographic efficiency:** When EU validators lead consecutive slots, block propagation is sub-50ms. Intercontinental latency is paid once per slot transition, not once per block.
- **Better bundle batching:** Transactions have a 13-second window to arrive at the leader instead of 2.5 seconds

Failover within slots: If the leader goes offline mid-slot, the existing position-based failover kicks in after 5 seconds. The backup validator produces the remaining blocks in the slot. The next slot starts automatically with a different leader.

Scaling: With 10 validators at 5 blocks/slot, each validator leads for ~13 seconds every ~130 seconds. With 50 validators, each leads for ~13 seconds every ~650 seconds. Geographic clustering (multiple EU validators in consecutive slots) provides sustained low-latency windows.

No Slashing -- Jailing Instead

Vextaris does not slash validator stake. Misbehaving validators are jailed (removed from rotation after consecutive missed blocks). This protects delegators from losing funds while still incentivizing uptime.

- **Double-sign detection:** Validators producing conflicting blocks at the same height are jailed with escalating penalties (1h → 24h → 7d → 30d)
- **Unjail:** Submit `vex_unjail` after jail duration expires
- **Upgrade grace period:** 100 blocks (~200s) after upgrade halt -- prevents jailing validators for upgrading

Bad actors lose opportunity cost (missed rewards) and reputation, not principal.

Reputation Scoring

Validator reputation uses a 7-factor, 100-point scoring system that affects leader selection weight and future reward multipliers.

Factor	Weight	Measures
Uptime	25 pts	Block production consistency
Commission fairness	15 pts	Reasonable rates, stability
Self-stake	15 pts	Skin in the game
Governance participation	15 pts	Upgrade voting activity
Tenure	10 pts	Time active on network
Delegator trust	10 pts	Stake attracted from others
Reliability	10 pts	Historical jail count, missed blocks

Reputation Grades

Grade	Score Range
A+	95-100

Grade	Score Range
A	90-94
B	75-89
C	55-74
D	35-54
F	0-34

New validators start at score 50 (neutral, "unproven"). A validator doing everything right reaches 80+ in ~7 days and 90+ in ~30 days.

Vexcel -- Adaptive Attestation DAG

Vexcel (Vextaris Accelerated Consensus Layer) is an adaptive consensus extension that makes HyperSync latency-tolerant across global validator sets.

How it works:

- When the leader produces blocks on time, the chain is purely linear -- zero overhead, same as a traditional BFT chain.
- When the leader is slow (network latency, geographic distance), non-leader validators produce lightweight **attestation blocks** -- signed, zero-transaction blocks that prove liveness and state agreement.
- Attestation blocks are stored separately and never enter the canonical chain. The next leader references attestation hashes in its block header, forming a DAG of liveness proofs.
- If the leader fails entirely, the first attestation is promoted to canonical after a grace period -- deterministic, same result on every node.

What this solves:

- **Geographic fairness:** High-latency validators (Singapore, Mumbai) produce blocks at the same rate as low-latency validators (Europe). Before Vexcel, the closest validator to the seed produced 68% of blocks. After Vexcel, all validators produce a roughly even share regardless of geographic latency.

- **No rollbacks:** Traditional failover systems produce temporary blocks that must be rolled back when the leader's block arrives. Vexcel eliminates rollbacks entirely -- attestation blocks are never finalized, so there is nothing to undo.
- **Unified mechanism:** Attestation blocks replace 4 separate mechanisms (failover, voting, heartbeat, liveness proof) with one simple concept.

RPC endpoints:

- `vex_getAttestations(height)` -- Returns attestation blocks at a given height
- `vex_getBlockDAG(height)` -- Returns the canonical block plus all attestation blocks, showing the full DAG structure

Consensus Security

- **Byzantine Fault Tolerance:** Tolerates up to 1/3 malicious validators
- **Block Proposer Signatures:** Every block is Ed25519-signed by its proposer. Invalid signatures result in instant peer banning.
- **PeerGuard:** Per-peer scoring system with replay detection (200-hash ring buffer), rate limiting, and validator allowlisting. Score drops to 0 = permanent ban.
- **Validator Set from State:** No self-adding. Validators loaded from on-chain staking state only.
- **Attestation Validation:** Attestation blocks are validated for correct signatures, uniqueness (one per validator per height), and zero transaction content. Duplicate attestations are treated as double-sign violations.

Dragonfly Stream Integration

Dragonfly Stream is a complementary transaction delivery system that eliminates the gossip mempool. While Vexcel handles block-level consensus (attestation DAG, leader promotion), Dragonfly Stream handles transaction-level delivery (sealed forwarding, PQ proofs, MEV elimination).

Together they form a complete pipeline: transactions flow directly to the leader via Dragonfly, and if the leader fails, Vexcel promotes the next validator who already holds those transactions via the pipeline backup. Zero gap, zero loss.

See Dragonfly Stream for full details.

P2P Transport

- **QUIC:** TLS 1.3 + multiplexing over UDP. Replaced TCP+Noise+Yamux.
- **GossipSub:** `"blocks"` and `"votes"` topics only (no transaction gossip). Explicit peers for small networks, flood_publish enabled.
- **Authenticated Validator Identity:** On connection, validators prove Ed25519 signing key ownership via domain-separated BLAKE3 signature over PeerId. Mutual authentication in one round-trip. Enables allowlisting and direct-to-leader routing.
- **Dragonfly Stream:** Direct-to-leader sealed forwarding over a dedicated authenticated P2P channel. Leader PeerId resolved from the authenticated identity map at submit time, with redundancy for resilience.
- **Bulk Sync:** Custom SyncCodec with 64MB response limit, 60s timeout, batch size 10.
- **Bundle TTL:** 120s default. Expired bundles rejected on receive, skipped on drain.

Block Commit Pipeline

```
Begin block
    |
Execute bundles + rewards
    |
Commit block (single atomic write batch)
    |
Recompute state root
    |
Persist to storage
```

All block writes (accounts, validators, blocks, explorer indexes, intent status) are buffered during block scope and committed atomically via RocksDB WriteBatch. Crash-safe -- no partial state.

CLI Flags

Flag	Default	Description
`--validator-key <path>`	none	Ed25519 signing key for block/vote signing
`--block-time <secs>`	12	Block production interval

Flag	Default	Description
`--max-txs-per-block <N>`	10000	Maximum transactions per block
`--gas-price <N>`	10	Base gas price in nanoVXS/gas
`--rpc-rate-limit <N>`	100	Per-IP RPC rate limit in requests/second (0 = unlimited)
~~`--dragonfly-stream`~~	deprecated	Dragonfly Stream is always active (Phase 3). Flag retained as hidden no-op.

Dragonfly Stream -- Mempoolless Transaction Pipeline

Dragonfly Stream replaces the traditional gossip-broadcast mempool with a sealed, direct-delivery pipeline. Transactions flow from client to block leader without pooling -- eliminating the primary attack surface for MEV (Maximal Extractable Value).

Why No Mempool?

Every major MEV attack requires observing pending transactions before they are included in a block. On traditional chains, the mempool is a public broadcast channel where bots watch for opportunities to front-run, sandwich, or reorder transactions.

Dragonfly Stream removes the pool entirely. Transactions are forwarded directly to the current block leader via authenticated P2P channels, sealed with post-quantum cryptographic signatures that prove temporal ordering.

```
Traditional chain:
  User -> Broadcast to ALL peers (mempool) -> Bots observe -> Leader
includes
  Risk: Front-running, sandwich attacks, censorship

Dragonfly Stream:
  User -> RPC node -> Sealed forward directly to Leader + 1 backup ->
Block
  Risk: None of the above -- no public pool to observe
```

How It Works

Transaction Flow

1. **User submits bundle** to any RPC node (signed Ed25519)

2. **RPC validates** signature and format
3. **Seal creation** -- the receiving validator creates a post-quantum seal over the transaction:
    - A cryptographic commitment binding the transaction to its height and the sealing validator
    - Signature: Dilithium3 (NIST Level 3 post-quantum) or Ed25519 (classical fallback)
4. **Leader resolution** -- the RPC node resolves the current leader's PeerId from the authenticated validator identity map
5. **Direct forwarding** -- sealed transaction sent directly to the elected leader + 1 bootnode backup via dedicated P2P command (not gossip). Falls back to all-bootnode routing if leader PeerId is unknown.
6. **Leader includes** the transaction in the next block
7. **Seal verification** -- leader verifies seals on received transactions

PQ Seals (Post-Quantum Cryptographic Seals)

Every forwarded transaction carries a cryptographic seal that proves:

- **Who** saw the transaction (validator identity)
- **When** they saw it (block height binding)
- **Authenticity** (quantum-resistant Dilithium3 signature)

Validators auto-generate Dilithium3 keypairs on startup. No manual key management required.

Seal types:

- **PQ (Dilithium3)** -- Primary. NIST-approved lattice-based post-quantum signatures. ~3.3 KB per seal.
- **Classical (Ed25519)** -- Fallback for validators without PQ keys. 64-byte signature.

Authenticated Validator Identity

On peer connection, validators cryptographically prove their identity by signing a domain-separated message that binds their Ed25519 signing key to their libp2p PeerId:

```
message = blake3("vexidus-validator-announce-v1" + peer_id)
signature = ed25519_sign(validator_key, message)
```

Both sides authenticate in one round-trip. This creates a live mapping from validator pubkey to PeerId, enabling direct-to-leader transaction routing. The mapping is self-healing -- if a validator's PeerId regenerates (e.g., data wipe), the next connection re-announces automatically.

Direct-to-Leader Routing

When a transaction is submitted, the RPC layer resolves the elected leader's PeerId from the authenticated identity map and sends the sealed transaction directly:

```
Phase 3 (before):
  Client -> Validator_A (RPC) -> Seed_1, Seed_2, Seed_3 -> All Validators
  Latency: ~3 hops, 5x bandwidth amplification

Phase 3.2 (current, linear leader slots):
  Client -> Validator_A (RPC) -> Leader (direct, stable for 5 blocks) + 1
Bootnode (backup)
  Latency: ~1 hop, 2x bandwidth, routing target stable for ~13 seconds
```

Why 1 bootnode backup? The leader may rotate at slot boundaries. The backup bootnode relays to all connected validators, ensuring the transaction reaches the new leader if rotation occurs.

Stable routing target: A stable routing target (rather than one that changes on every block) dramatically reduces stale routing and bundle loss on intercontinental links.

Graceful fallback: If the leader's PeerId is unknown (not yet authenticated, disconnected, or validator just started), the system falls back to Phase 3 all-bootnode routing automatically.

LeaderBuffer Re-Forward

All validators buffer incoming transactions in their local LeaderBuffer, but only the elected leader drains it for block inclusion. To prevent silent bundle drops on non-leader nodes:

- **Non-leader detection:** Every 2 seconds, non-leaders check if their buffer has bundles sitting for 8+ seconds
- **Re-forward:** Stale bundles are drained and forwarded to the current leader over the direct P2P channel
- **Deduplication:** The leader's LeaderBuffer deduplicates via hash tracking -- re-forwarded bundles that already exist are silently ignored

- **Anti-loop:** After draining, the staleness timer resets. No re-trigger until new bundles arrive and age

This ensures transactions submitted to any validator eventually reach the current leader, even across leader slot transitions.

Pipeline Failover

If the leader fails entirely:

1. Bootnode backup already relayed the transaction to all validators
2. Vexcel attestation promotion seamlessly transitions to the new leader
3. New leader already holds the transaction in its local LeaderBuffer
4. Non-leader re-forward ensures stale bundles reach the new leader within 10 seconds
5. No re-submission needed

This builds on the existing Vexcel adaptive attestation DAG -- Dragonfly Stream and Vexcel form a self-reinforcing system.

MEV Protection

Dragonfly Stream is one component of a multi-layer MEV elimination strategy:

Layer	Mechanism	What It Prevents
No mempool	Dragonfly direct delivery	Front-running (no pool to observe)
Deterministic ordering	Content-based transaction ordering	Proposer reordering (content-determined order)
Atomic execution	IntentVM composite intents	Sandwich attacks (no inter-step window)
PQ seals	Dilithium3 temporal proofs	Quantum tx forgery
Attestation commitments	Validators commit received tx hashes	Censorship (provably detectable if leader drops attested txs)

At full deployment, Vextaris has **zero known MEV vectors**: no mempool to observe, the proposer cannot control ordering, intents execute atomically, and censorship is cryptographically detectable.

Achieved without BLS/DKG threshold encryption -- simpler, faster, stronger.

Wire Protocol -- Borsh Binary Encoding

All P2P communication (transaction forwarding, block sync, attestation gossip) uses a dual-format encoding system:

- **Borsh binary** (active when `--borsh-wire` enabled): 5-10x smaller messages on the wire, ~10x faster serialization/deserialization compared to JSON.
- **Automatic format detection**: Messages prefixed with `VX` magic bytes (0x56, 0x58) are decoded as Borsh; all others as JSON. This enables safe rolling upgrades -- old and new binary versions coexist seamlessly during transitions.
- **ForwardBundle impact**: Under Borsh encoding, sealed transaction bundles forwarded to the leader are significantly smaller, reducing latency on intercontinental P2P links.

Phased Rollout

Dragonfly Stream deployed incrementally via VexVisor on-chain governance -- no genesis restart required at any phase.

Phase	Status	Description
Phase 1	Live	Direct forwarding of sealed transactions to the block leader over authenticated P2P, replacing gossip broadcast.
Phase 2	Live	PQ sealing. Auto-generated Dilithium3 keys. Seal creation and verification on every forwarded transaction.
Phase 3	Live	Mempool eliminated. Transaction gossip topic removed. PQ sealing always-on.
Phase 3.1	Live	Authenticated validator P2P identity + direct-to-leader routing. Transactions sent to the elected leader (1 hop) with redundancy.
Phase 3.2	Live	Stable routing target + re-forwarding of stale buffered transactions, ensuring transactions submitted to any validator reach the current leader.
Phase 4	Planned	PQ enforcement. Classical-only seals rejected. Validator registration requires Dilithium3 pubkey.

Key Properties

- **O(1) bandwidth** -- each transaction forwarded to 2 targets (leader + 1 bootnode backup), not O(N) gossip flood
- **Pipeline failover** -- bootnode backup relays to all validators; new leader already holds transactions
- **Censorship accountability** -- attestation blocks commit to received transaction hashes; leader censorship is provably detectable
- **Adaptive timing synergy** -- Dragonfly Stream + Vexcel + pressure-aware micro-blocks form a self-regulating throughput engine
- **No hard restart** -- all parameters (buffer sizes, PQ requirements, max transactions per block) adjustable via governance or CLI flags
- **Quantum-safe from day one** -- Dilithium3 seals protect transaction ordering against future quantum computers

Dragonfly Stream + Vexcel

Dragonfly Stream and Vexcel are complementary systems:

- **Vexcel** handles block-level consensus (attestation DAG, leader promotion, geographic fairness)
- **Dragonfly Stream** handles transaction-level delivery (sealed forwarding, PQ proofs, MEV elimination)

Together they form a complete pipeline: transactions flow directly to the leader via Dragonfly, and if the leader fails, Vexcel promotes the next validator who already holds those transactions via the pipeline backup. Zero gap, zero loss.

Configuration

Dragonfly Stream is always active as of Phase 3. PQ sealing activates automatically when a validator signing key is present (ephemeral Dilithium3 keys are auto-generated on startup). No flags required.

The `--dragonfly-stream` flag is deprecated (hidden no-op, retained for backward compatibility).

RPC Endpoints

Method	Description
`vex_getDragonflyStats`	Returns Dragonfly Stream statistics (forwarded count, received count, seal type, enabled status)

ElasticState -- Intelligent 3-Tier Storage Engine

The Problem

Every blockchain today treats all state equally. A validator that transacted 5 seconds ago and a wallet that has not moved in 3 years sit in the same data structure, consuming the same resources. As chains grow, state bloat becomes the dominant bottleneck -- RocksDB compaction slows down, bloom filter pressure increases, and every full node must store the entire history of every account at the same access speed.

Ethereum's state trie holds 300M+ accounts in a single Merkle Patricia Trie. Solana's account store exceeds 500 GB. Both treat a billion-dollar DeFi protocol and a dust account identically. This is why "state rent" and "state expiry" have been debated for years without resolution -- they punish users for a problem that is actually an infrastructure design failure.

The Solution

ElasticState automatically routes account data through three storage tiers based on activity, giving sub-microsecond access to hot data while compressing dormant accounts by up to 90%.

```
+-----------------------------------------------------+
|  HOT TIER -- DashMap (in-memory)                     |
|  Active accounts (last 7 days)                      |
|  Access: &lt;1us | Capacity: 100K entries / 8 GB        |
+-----------------------------------------------------+
|  WARM TIER -- RocksDB (LZ4 compression)              |
|  Recent accounts (7-90 days inactive)               |
|  Access: ~100us | Fast SSD reads                    |
+-----------------------------------------------------+
|  COLD TIER -- RocksDB (ZSTD max compression)         |
|  Dormant accounts (90+ days inactive)               |
|  Access: ~1-10ms | 90% storage reduction            |
|  Includes Merkle proofs for light client verification|
+-----------------------------------------------------+
```

How It Works

1. **Write-through:** Every state mutation writes to both the main RocksDB (source of truth for state root) AND the appropriate ElasticState tier. The main DB stays canonical; ElasticState is a read-acceleration layer.

2. **Auto-promotion on read:** A cold account accessed during transaction execution is automatically promoted to the warm tier. A warm account accessed repeatedly migrates to hot. Users never notice -- access is seamless.

3. **Background demotion:** Every 100 blocks (~20 minutes), a background task scans the hot cache and demotes accounts inactive for 7+ days to the warm tier. Accounts inactive 90+ days in warm are compressed to cold.

4. **Zero-copy checkpoints:** RocksDB's built-in `Checkpoint` API creates instant snapshots via hard-links every 1,000 blocks. No I/O overhead, no lock contention. Old checkpoints auto-pruned (keep last 5).

Comparison

Chain	State Model	Problem
Ethereum	Single Merkle Patricia Trie	300M+ accounts in one structure. State rent debated since 2017, still unresolved.
Solana	Flat account store	500+ GB. "Account rent" charges users to exist.
Cosmos	IAVL tree	State grows linearly. No tiering.
Vextaris	**ElasticState 3-tier**	**Hot accounts in memory (<1us), cold accounts compressed 90%. Main DB stays small.**

Key Design Decisions

- **Active on all validators** (since early 2026). Compiled as default feature, enabled via `--elastic-state` CLI flag.

- **State root integrity preserved.** The main RocksDB default CF feeds the LtHash accumulator, which produces the state root. ElasticState is a read cache that cannot affect consensus.
- **No state rent.** Vextaris will never charge users to keep their accounts alive. ElasticState solves the problem at the infrastructure level -- dormant accounts are compressed, not evicted.
- **Merkle proofs for cold accounts.** Cold-tier accounts include verifiable proofs, enabling future light clients to verify historical state without downloading the full chain.

Parameters

Parameter	Default	CLI Flag
Hot cache entries	100,000	`--elastic-hot-entries`
Hot cache memory	8 GB	`--elastic-hot-memory-mb`
Hot to Warm threshold	7 days inactive	--
Warm to Cold threshold	90 days inactive	--
Demotion interval	Every 100 blocks (~20 min)	--
Warm compression	LZ4 (fast)	--
Cold compression	ZSTD (max, ~90% reduction)	--
Feature flag	On (default since early 2026)	`--elastic-state`

RPC

`vex_getStorageStats` -- Returns tier counts, memory usage, storage reduction percentage.

State Root: LtHash Accumulator

The state root is computed using an **LtHash (Lattice Hash) accumulator** -- a homomorphic hash where `hash(set) = sum(hash(element))`. This allows O(1) state root updates per write instead of O(N) full database scans. Same approach as Solana SIMD-0215.

- **Accumulator**: 1024 x u16 (2048 bytes), updated on every consensus-relevant state write or delete
- **State root**: Blake3 over the accumulator -- computed once per block after all bundle execution
- **Migration**: First boot computes the accumulator from a one-time full DB scan, then persists it
- **Crash safety**: Accumulator is included in the atomic WriteBatch. Rolled back from backup on commit failure.

Per-CF Compression (Deployed)

RocksDB column families use per-CF compression tuned to access patterns:

Column Family	Compression	Rationale
Blocks, Explorer, NFTs	ZSTD	Immutable data, 40-60% better compression
Accounts, Validators, Attestations, Pools	LZ4	Fast reads, frequent updates

This achieves ~35% total disk reduction on archive nodes compared to uniform compression.

VIDA -- Sovereign Digital Identity

> "Vida" means "life" in Portuguese and Spanish -- reflecting the birth-to-legacy lifecycle this system represents.

What is VIDA?

VIDA (Vextaris Identity & Digital Authentication) is a sovereign digital identity protocol built directly into the Vextaris L1 blockchain. It provides a framework for issuing, holding, verifying, and revoking digital credentials -- from birth certificates to professional licenses to property titles -- as cryptographically signed, soulbound on-chain records.

VIDA is not a separate product or sidechain. It uses existing Vextaris primitives -- post-quantum keypairs, soulbound NFTs, on-chain attestation, and declarative intent programs -- composed into a universal identity layer.

Why On-Chain Identity?

Today (Fragmented)	VIDA (Unified)
Paper birth certificates that fade, burn, or get lost	Soulbound on-chain credential, verifiable forever
Social Security cards you memorize but never carry	Cryptographic proof you present without exposing the number
Passports that take weeks to renew and can be forged	Quantum-resistant credential issued in seconds, verified in milliseconds
Driver's licenses that differ by state and can't be verified digitally	Standardized credential with built-in revocation
Education transcripts requiring snail-mail verification	Instant cryptographic proof from the issuing institution

Core Principles

Individual Sovereignty -- The person owns their identity. Governments are issuers and verifiers, but the individual holds the keys.

Single Source of Truth -- One wallet, one identity. Every credential attached to the same cryptographic root.

Privacy by Design -- Credentials are encrypted on-chain. Only the holder and authorized verifiers can read them. Future ZK integration enables proving claims ("I am over 21") without revealing underlying data.

Quantum-Resistant -- Every VIDA wallet uses Ed25519 + Dilithium3 dual-key architecture (VSA v2). Credentials issued today remain secure against quantum computers.

Government-Compatible -- Any government entity -- from a clerk's office to a federal ministry -- can become an approved credential issuer using standardized templates.

Architecture

Issuer Framework

VIDA uses a 4-tier trust model for credential issuers:

Tier	Name	Trust Level	Example
1	Sovereign	Highest -- sovereign authority	Federal agencies, state DMVs
2	Delegated	High -- state/provincial/local government	Driver's licenses, voter registration, property
3	Institutional	High -- regulated entities	Universities, hospitals, employers
4	Verified	Medium -- verified private entities	Insurance companies, certification bodies, supply-chain inspectors

Issuers must be registered on-chain with verified credentials. Higher-tier issuers can endorse lower-tier ones, creating a web of trust rooted in sovereign authority.

Credential Types

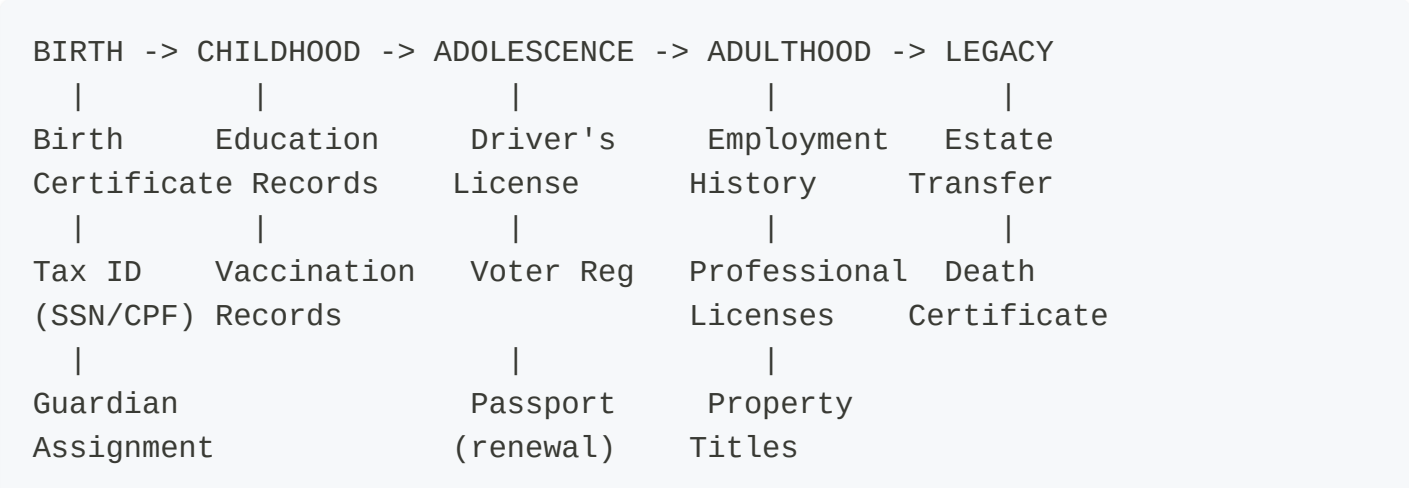
VIDA supports 18 standard credential types covering the full human lifecycle, plus a `custom:` prefix for application-specific credentials:

Category	Credentials
Civil Identity	Birth certificate, citizenship, tax ID, death certificate
Government ID	Passport, driver's license, voter registration, state ID
Education	Diploma, transcript, professional license
Health & Safety	Vaccination record, background check
Financial	Credit attestation, employment record
Travel	Visa, residency
Property	Property title
Custom	`custom:<name>` -- application-specific (e.g. `custom:supply_chain_inspector`)

All credentials are **soulbound** -- they cannot be transferred to another wallet. They can be revoked by the issuer if circumstances change (e.g., license expiration, court order, professional misconduct).

The `custom:` prefix lets any domain extend VIDA without protocol changes -- the supply-chain inspector credential used by Vexalus is one example.

The VIDA Lifecycle



A person's wallet accumulates credentials over their lifetime. At age 18, a maturity claim transfers guardian control to the individual. At end of life, estate credentials enable inheritance.

On-Chain Operations

VIDA adds 5 native L1 operations to Vextaris:

Operation	Description
RegisterIssuer	Register an organization as a credential issuer (tier 1-4)
IssueCredential	Issue a soulbound credential to a wallet (atomic: validate issuer → mint soulbound NFT → anchor doc hash)
RevokeCredential	Revoke a previously issued credential (burn NFT + permanent revocation record)
VerifyCredential	On-chain verification of credential validity (existence, expiration, issuer status, soulbound flag)
UpdateIssuerStatus	Update issuer status (active / suspended / revoked) -- governance-controlled

These are first-class blockchain operations -- not smart contract calls. They execute at consensus speed with the same finality guarantees as token transfers.

W3C Compatibility

VIDA credentials can be exported as W3C Verifiable Credentials (VCs), enabling interoperability with existing identity systems:

- **DID Method**: `did:vex:<wallet-address>` resolves to the holder's on-chain public key and credential list
- **VC Format**: Standard JSON-LD with the W3C Credentials v2.0 context
- **Verification**: Any W3C-compatible verifier can validate VIDA credentials without running a Vextaris node

RPC Endpoints

Method	Description
`vida_registerIssuer`	Register a new credential issuer

Method	Description
`vida_updateIssuerStatus`	Update issuer status (suspend / revoke / reactivate)
`vida_issueCredential`	Issue a credential to a wallet
`vida_revokeCredential`	Revoke an issued credential
`vida_verifyCredential`	Full trust chain check (existence, expiration, issuer status, soulbound)
`vida_getIssuer`	Query issuer details and trust tier
`vida_listIssuers`	List registered issuers, filter by tier or jurisdiction
`vida_getStats`	Network-wide credential and issuer statistics
`vida_exportW3C`	Export credential as W3C Verifiable Credential (JSON-LD)
`vida_resolveDID`	Resolve a `did:vex:` identifier to its DID Document

Supply-Chain Inspector Credential

The first production credential type using the `custom:` extension is `custom:supply_chain_inspector` -- a Tier 4 (Verified) credential issued to independent inspection firms.

The credential lets an inspector cryptographically sign attestations over a supply-chain dispute evidence package. A verifier then checks:

1. **Signature** — Ed25519 over the canonical attestation payload
2. **Credential validity** — `vida_verifyCredential` confirms the inspector's credential is current, not revoked, and held by the address that signed
3. **Type match** — the on-chain credential type is `custom:supply_chain_inspector`
4. **Specialization** — optional (`cold_chain`, `hazmat`, `automotive`, `pharmaceutical`, `electronics`, `general`)

Revoke an inspector's credential and any past attestations they signed cease to verify -- making collusion economically irrational. This is the Vexalus Accountability Web Layer 7.

Consumer Surface (VexSpark Wallet)

VexSpark v1.2 (multichain: Vextaris L1 + 7 EVM chains + Solana) ships a complete VIDA credential experience for individuals:

Surface	What it does
Credentials list	View every VIDA credential issued to your wallet -- tier badge, issuer name, status, type-specific icon
Credential detail	On-chain `vida_verifyCredential` check, view metadata, see issuer's tier and jurisdiction
Present (QR)	Generate a QR carrying the credential reference + Ed25519 signature over a verifier-supplied challenge. The verifier scans to cryptographically confirm you hold the credential -- your secret key never leaves the device
W3C export	Download the credential as a W3C Verifiable Credential (JSON-LD) for use in non-Vextaris identity systems
Kiosk Mode	Large-format display + camera scan-to-verify for institutional reception desks, employer check-in, event verification

The Quantum Shield panel in VexSpark derives an ML-DSA-65 post-quantum keypair alongside the existing Ed25519 key -- the wallet address does not change. The network does not accept ML-DSA-65 keys yet: that upgrade is built but not activated, so until it is, the panel prepares the key without registering it on-chain. (Accounts can already hold a CRYSTALS-Dilithium3 key, which the chain verifies today.)

What VIDA Does NOT Do

- It does not replace governments -- it gives them better tools
- It does not require surrendering privacy -- it enables proving claims without revealing data
- It does not create a surveillance system -- the individual owns their data, not the state
- It does not depend on constant internet -- credentials are locally cached and verifiable offline

Current Status

VIDA is live on the Vextaris testnet with the full issuer registry, credential issuance / revocation / verification, W3C VC export, and `did:vex:` DID resolver, with credentials issued on-chain.

Capability	Status
Issuer registry (4 tiers)	Live
18 standard credential types + `custom:` prefix	Live
Soulbound credential issuance	Live
On-chain revocation with permanent record	Live
Full trust-chain verification	Live
W3C Verifiable Credentials export	Live
`did:vex:` DID method resolver	Live
Supply-chain inspector credential type	Live (Vexalus pilot)
VexSpark consumer surface (list / present / W3C export / Kiosk Mode)	Live (v1.2)
MaturityClaim VIP template (age 18 sovereignty transfer)	Planned
Multi-approval issuance flow (2-of-3 for critical credentials)	Planned
ZK selective disclosure	Planned

See the CLI reference for VIDA command usage (`vexidus vida` command group).

VSA v2 -- Quantum-Hardened Smart Accounts

Dual-Signature Architecture

Every Vextaris account uses Ed25519 (classical, Solana-compatible) alongside Dilithium3 (NIST-approved post-quantum lattice-based signatures). This provides quantum resistance TODAY while maintaining compatibility with existing Ed25519 wallets.

Algorithm	Purpose	Security Level
Ed25519	Transaction signing, block proposing	Classical (Solana-compatible)
Dilithium3	Post-quantum signatures (dormant, ready for activation)	NIST Level 3 (quantum-safe)
Blake3	Hashing (state root, tx hash, address derivation)	256-bit (quantum-resistant)

Address Derivation (Same Key, Three Views)

```
Ed25519 Keypair -> pubkey (32 bytes)
  +---> Vx0:    SHA256(pubkey)[0..20] -> base58 -> "Vx0abc..."   (native)
  +---> 0x:     last 20 bytes -> "0x71C7..."
(EVM/MetaMask)
  +---> base58: raw pubkey -> "7xKXtg..."
(Solana/Phantom)
       All resolve to the same internal 32-byte account address.
```

Address Formats

Format	Example	Used By
Vx0 (native)	`Vx0abc...xyz`	VexSpark, native SDK
0x (EVM)	`0x71C7...`	MetaMask, eth_* RPC

Format	Example	Used By
System (32-byte hex)	`0x000...0001`	Genesis accounts, internal

Key Management

VSA v2 provides protocol-level key management without requiring smart contracts:

- **AddKey** -- Add Ed25519 or Dilithium3 key to your account
- **RemoveKey** -- Remove a key by hash (must keep at least one)
- **RotateKey** -- Atomic swap of one key for another
- **Guardian recovery** -- 3-of-5 trusted guardians can recover access (configurable threshold, timelock delay, cancelable)
- No seed phrase required for recovery (protocol-level, not a service)

All key management operations are defined natively in the protocol and live on testnet.

Pubkey Revelation

Accounts created by receiving transfers get a placeholder public key. On first send, the wallet includes the real Ed25519 public key alongside the signed bundle. The protocol verifies the key hashes to the account address, registers it, and proceeds -- seamless to the user.

Flow:

1. The wallet includes the sender's Ed25519 public key with the signed bundle
2. The network confirms the public key hashes to the sender's address
3. The network verifies the Ed25519 signature against the provided public key
4. On success, the public key is registered for the account -- future sends do not need revelation

The native wallet VexSpark handles pubkey revelation automatically.

Guardian Recovery

Setup

An account owner configures recovery with a `SetRecovery` operation that specifies:

- **Guardians** -- up to 10 trusted Vx0 addresses
- **Threshold** -- the M-of-N guardian approvals required to recover
- **Timelock** -- a delay before recovery can finalize (recommended: 48-72 hours)

Recovery Flow

1. **User loses all keys**
2. **Guardians initiate**: M-of-N guardians submit `InitiateRecovery` with a new public key
3. **Timelock starts**: The pending recovery is recorded on the account
4. **Owner can cancel**: If they regain access during timelock, submit `CancelRecovery`
5. **After timelock**: Anyone submits `FinalizeRecovery` -- replaces all account keys with the recovery key

Operations

Operation	Who Signs	What Happens
`SetRecovery`	Account owner	Configures guardians, threshold, timelock
`InitiateRecovery`	Guardian	Starts recovery with new pubkey, records guardian approval
`CancelRecovery`	Account owner	Cancels pending recovery
`FinalizeRecovery`	Anyone	After timelock, replaces keys. Requires threshold guardian approvals.

VNS -- Vextaris Name Service

VNS provides human-readable `.vex` names as an extension of the VSA identity system. Each `.vex` name IS a VSC-21 NFT -- ownership of the NFT IS name resolution. Transfer the NFT, transfer the name. No separate registry update needed.

- Register `chris.vex` -> mints an NFT in the global VNS collection
- Resolve `chris.vex` -> find the NFT -> the owner's address IS the resolved address
- Transfer uses the existing `TransferNft` operation -- no new operation type needed

Pricing

Name Length	Fee
3-4 characters	1,000 VXS
5-7 characters	100 VXS
8+ characters	10 VXS

Fees are credited to the Foundation (`0x...01`).

Validation

- 3-64 characters, alphanumeric + hyphen
- No leading/trailing hyphen
- Normalized to lowercase, `.vex` suffix stripped before storage

RPC Endpoints

Method	Description
`vex_registerName`	Register a `.vex` name
`vex_resolveName`	Resolve name to address
`vex_reverseResolve`	Address to name lookup
`vex_getNameInfo`	Full name details

IntentVM Composition

VNS names compose atomically with IntentVM -- "send 10 VXS to chris.vex and stake the rest" is one intent, one signature.

Transfer Guard

Transfer Guard is a self-imposed account protection mechanism that lets users set rules on their own account to prevent unauthorized or accidental transfers.

Operations

Operation	Description
`EnableTransferGuard`	Activate transfer guard with configurable rules
`DisableTransferGuard`	Remove transfer guard (requires current guard key)
`UpdateTransferGuard`	Modify guard rules (whitelist, daily limits, cooldowns)
`TransferGuardApprove`	Pre-approve a specific transfer that would otherwise be blocked

Guard Rules

- **Whitelist mode** -- only allow transfers to pre-approved addresses
- **Daily transfer limit** -- cap total outgoing VXS per 24-hour period
- **Cooldown period** -- require a delay between enabling guard changes and them taking effect (prevents instant removal by an attacker)
- **Guardian key** -- optionally require a separate key to modify guard settings

Transfer Guard is entirely opt-in. Accounts without it enabled behave exactly as before.

Quantum Migration Path

- **Phase 1 (Now)**: Ed25519 only. Wallet SDK compatible (Phantom, MetaMask via bridge).
- **Phase 2 (~2028)**: Users add Dilithium3 key alongside Ed25519 via `AddKey`. Either key signs. Address unchanged.
- **Phase 3 (~2032+)**: High-value ops require Dilithium3. V0-only accounts get warnings.

The infrastructure is ready -- post-quantum key generation works and signature verification already handles both Ed25519 and Dilithium3 signatures. A post-quantum block-signature field is already reserved on every block for future dual-sig enforcement.

Why Not ML-KEM?

Dilithium3 is a signature scheme; ML-KEM is a key encapsulation scheme. Blockchains need signatures. Vextaris uses Ed25519 for speed and compatibility, Dilithium3 for quantum hardening.

Atomence -- Cross-Chain Escrow Swaps

U.S. Patent Application 63/998,160 (Filed March 6, 2026)

Atomence is Vextaris's cross-chain settlement layer. A user locks an asset in an on-chain escrow on Vextaris; an independent, bonded **market maker** pays the user on another chain from its own inventory; once a quorum of registered **attesters** confirms that payment is final, the escrow is released to the market maker. If the market maker does not deliver in time, the user takes the escrow back -- plus compensation from the market maker's bond.

The escrow rules are enforced by the protocol itself. No company, admin key or relayer can move, freeze or redirect an escrowed order.

> ⚠ **TESTNET STATUS (NOVEMBER 2026)**
>
> Bridge deposits are **paused** on testnet, and a pause also stops new Atomence orders and matches -- so testnet Atomence is **not open for new swaps** right now. The CLI and SDKs do not yet expose the destination-asset and minimum-amount fields that new orders need; they will before Atomence reopens. Existing orders can still be cancelled or reclaimed.

> ⓘ **NEW RULES FROM BLOCK 30,655,525**
>
> This page describes how Atomence works from block **30,655,525** onward. From that block:
>
> - every order states the **destination asset** and the **minimum amount** the user must receive;
> - only **registered market makers** can match orders;
> - a matched order must be settled within **1 hour**, or the user can reclaim the escrow **plus 10% of the market maker's bond**;
> - settlement requires signatures from a **quorum of registered attesters**, and only for **final** payments;
> - one destination-chain payment can settle only one order;
> - the fee is **0.30%**, split between the market maker and the protocol treasury -- **nothing is burned**;

- a bridge pause stops new orders and matches but never locks funds that are already in flight.

Orders created before activation do not state a destination asset or minimum amount, so they **cannot be matched afterwards**. If such an order is still pending, cancel it; if it was matched before activation and never settled, reclaim it once its settle deadline has passed (see Reclaim).

Why Swaps Instead of Wrapped Tokens

Many bridges move value by locking an asset on one chain and minting a "wrapped" copy on another. The copy is only worth something while the bridge, its vault and its signers keep working -- when a bridge fails, holders of the copy can be left with a token nobody has to redeem.

Atomence avoids that: the user never holds an IOU for a foreign asset. They swap a Vextaris asset for a real asset on the destination chain, paid by a market maker that has posted a bond, and the escrow guarantees that the user either gets paid or gets their funds back.

Problem with wrapped / reverse-bridge exits	Atomence
A vault or custodian must hold the backing	No vault -- market makers pay from their own inventory
Holders depend on the bridge staying alive	Every order ends either settled or refunded; nothing is left pending forever
Liquidity fragmented across wrapped versions	The user receives the real asset on its own chain

---

How a Swap Works

```
User swaps a Vextaris asset for SOL on Solana:

1. USER    →  Creates an order: locks the asset in escrow and states
              destination chain, destination address, destination asset
("native" = SOL)
              and the MINIMUM amount of SOL they must receive
2. MM      →  A registered market maker matches the order
              (the 1-hour settle window starts)
3. MM      →  Pays at least the minimum amount of SOL to the user's
Solana address
4. ATTESTERS → Each independently checks the payment on Solana and, once
it is
              final, signs a statement naming this order, the recipient,
              the asset and the amount paid
5. MM      →  Submits the settlement with a quorum of attester signatures
6. CHAIN   →  Verifies the signatures and that amount paid ≥ the user's
minimum,
              then releases the escrow to the MM (minus the protocol fee)

If step 5 has not happened within 1 hour of matching:
   USER    →  Reclaims the full escrow + 10% of the MM's bond
```

1. Creating an order

The user locks the asset they are selling (VXS or another Vextaris token) and states:

Field	Meaning
Destination chain	The chain the user wants to be paid on (e.g. Ethereum, Solana, Bitcoin)
Destination address	The user's own address on that chain
Destination asset	`native` for the chain's own coin (ETH, SOL, BTC), or the token contract / mint address on the destination chain
Minimum received	The smallest amount of the destination asset the user accepts, in that asset's base units
Max spread	The largest spread (in basis points) the user accepts
Expiry	How long the order waits for a match (default 15 minutes, up to 1 hour)

The minimum amount is the user's price protection: a settlement that pays less than it is refused by the chain, regardless of what anyone signs.

While an order is **pending** (not yet matched), the user can cancel it at any time and the escrow is returned. An order that expires without being matched is refunded.

2. Matching -- registered market makers only

Only an **active, registered market maker** can match an order. Matching commits the market maker to deliver and starts a **1-hour settle window**. A matched order can no longer be cancelled by the user -- it ends either in settlement or in a reclaim after the deadline.

3. Payment and attestation

The market maker pays the user directly on the destination chain, at least the minimum amount, in the stated asset, to the stated address. It then collects signatures from the attesters and submits the settlement on Vextaris.

The market maker should pay early enough for the payment to become **final** (see Finality) and be attested **within the 1-hour window**.

If the market maker does not deliver: reclaim

If a matched order is not settled by its deadline, the user reclaims it (the same cancel operation used for pending orders, which is accepted once the deadline has passed). The user receives:

- the **full escrowed amount**, and
- **10% of the market maker's VXS bond**, taken from that market maker as a penalty.

A market maker whose bond falls below the 1,000 VXS minimum is deactivated and cannot match further orders until its bond is topped back up. **A market maker that walks away can never lock a user's funds.** Only the order's creator can cancel or reclaim it.

Market Makers

Market makers register by posting a **VXS bond**. The bond determines their tier and is what pays users when a market maker fails to settle in time.

Tier	VXS Bond	Max Spread	Description
Tier 1	100,000 VXS	0.50% (50 bps)	Institutional -- tightest spreads
Tier 2	10,000 VXS	1.50% (150 bps)	Professional desks
Tier 3	1,000 VXS (minimum)	5.00% (500 bps)	Individual market makers

- Registration is open to anyone who posts at least **1,000 VXS**. The tier follows the cumulative bond -- topping up moves a market maker to a higher tier without unstaking.
- The tier's max spread is the cap a market maker may quote; competition between market makers pushes actual spreads lower.
- Each failed settlement costs **10% of the bond**, paid to the affected user. Size your bond and your open orders accordingly.

What a market maker needs to run: inventory on the destination chains it serves, a VXS bond on Vextaris, and the ability to pay promptly and submit settlements with attester signatures before each deadline.

Proof of Payment: Registered Attesters

A settlement is accepted only with signatures from a **quorum of registered attesters**. Each attester runs on separate infrastructure with its own key and its own connection to the destination chains. On testnet the five attesters are operated by the Vexidus team on five separate machines; independent outside operators are planned before mainnet.

For each settlement, every attester independently:

1. reads the order from Vextaris and checks it is matched and states a destination asset and minimum amount;
2. looks up the payment on the destination chain;
3. checks that it is **final**, paid the order's destination address, in the order's destination asset, and at least the order's minimum amount;
4. only then signs a statement that names **this order, the recipient, the asset and the amount paid**.

Because the signed statement names the specific order, a signature cannot be reused for another order or another payment. The chain then rebuilds the statement from the order's own fields, checks the signatures against the registered attester set, counts the quorum, and re-checks that the amount paid is at least the user's minimum.

Quorum

Destination chain	Signatures required
Ethereum and other EVM chains, Solana	At least **half** of the registered attesters
Bitcoin	At least **70%** of the registered attesters
Any chain	**Never fewer than 3**

With five registered attesters this is **3-of-5** (Bitcoin: **4-of-5**). Each attester counts once, however many signatures it submits.

Finality

Attesters sign only payments that are final on the destination chain:

Chain	Final when
Ethereum and EVM chains	The payment's block is at or below the chain's **finalized** checkpoint, and the transaction succeeded
Solana	The transaction is confirmed at **finalized** commitment and succeeded
Bitcoin	The payment has at least **6 confirmations**

If anything is uncertain -- not yet final, missing data, an RPC error -- an attester refuses to sign.

Who the attesters are

The attester set is changed **only by on-chain governance**. On testnet, five attesters run today on five separate machines and watch the public test networks. **Independent outside operators are planned before mainnet.**

One payment, one order

A destination-chain transaction can settle **at most one order**. Once used, the same payment cannot be presented again for any other order.

Fee Model

A flat **0.30% (30 bps)** fee is taken from the escrowed amount when an order settles:

Recipient	Share
Market maker	0.15% (15 bps)
Protocol (Foundation) treasury	0.15% (15 bps)

Nothing is burned. On settlement the market maker receives the escrowed amount minus the 0.15% treasury share.

```
User escrows 1,000 VXS:

  Total fee (0.30%):      3 VXS
  To the protocol treasury: 1.5 VXS
  To the market maker:    998.5 VXS  (escrow minus the treasury share)
```

The market maker also earns the spread it quotes, bounded by its tier's maximum. Cancelled, expired and reclaimed orders pay no Atomence fee.

Pause Behaviour

When the bridge is paused:

- Atomence accepts **no new orders** and **no new matches**;
- orders already in flight **can still settle**, pending orders **can still be cancelled**, and matched orders **can still be reclaimed** after their deadline.

A pause never locks funds.

Supported Chains

Chain	Destination assets	Attester finality rule
Ethereum and EVM chains	Native coin, or a token by contract address	Finalized checkpoint
Solana	SOL, or an SPL token by mint address	Finalized commitment
Bitcoin	BTC	6 confirmations

On testnet, attesters observe the public test networks of these chains.

Atomence vs Traditional Bridge Exits

Dimension	Atomence	Traditional Bridge Exit
What the user receives	The real asset on its own chain	Often a wrapped copy, or a claim on a vault
Counterparty protection	Escrow + 1-hour deadline + bonded market maker	Trust in the vault and its signers
Price protection	User-set minimum amount, enforced on-chain	Varies
Liquidity source	Market maker inventory	Locked vault capital
Proof verification	Quorum of registered attesters (separate infrastructure; independent operators planned), final payments only	Oracle/guardian set or light client
Failure outcome	User reclaims escrow + 10% of the MM bond	Depends on the bridge

---

Relationship to VexBridge

Atomence moves value; VexBridge proves facts about other chains. The attester quorum that confirms Atomence payments is the verification layer VexBridge is becoming.

- **Today (testnet):** Atomence handles outbound swaps (Vextaris asset → asset on another chain). Inbound VexBridge deposits are **paused** on testnet.
- **Planned for mainnet:** Atomence becomes the way value moves between Vextaris and other chains **in both directions**. For inbound swaps, the market maker locks the Vextaris-side asset in escrow **first**; the user pays on the foreign chain only after seeing it locked, and is released the escrow once attesters confirm the payment. Major foreign assets (BTC, ETH, SOL, foreign USDC) are swapped into native Vextaris assets rather than held as wrapped tokens.

See VexBridge for the full mainnet bridging model.

Cross-Zone Settlement (VIMS)

The Vextaris Interplanetary Monetary System (VIMS) plans to use the same order lifecycle -- escrow, match, settle, refund -- for **cross-zone currency exchange** (e.g. VGD on Earth to VLD on the Lunar zone).

Querying Atomence State

Four read-only RPC methods expose the Atomence query surface:

Method	Returns
`vex_getAtomenceOrder(order_id)`	A single order's full lifecycle record: status, amounts, market maker, timestamps, and -- for orders created under the new rules -- `dest_asset`, `min_dest_amount` and `settle_deadline`
`vex_listAtomenceSettlements(token?, since_block?, limit?)`	Paginated list of settled orders, optionally filtered by token mint or starting block
`vex_getMarketMakers(tier?)`	Active market makers, optionally filtered to one tier
`vex_atomenceStats(token?)`	Aggregate statistics: settled count, total volume, volume-weighted average spread, last settlement block, active market maker count

The CLI exposes the same queries plus market-maker registration and cancel / reclaim:

```
# Read
vexidus atomence list-settlements --token VXS --limit 50
vexidus atomence get-order <order-id>
vexidus atomence list-mms --tier 1
vexidus atomence stats

# Write (require --key for local signing)
vexidus atomence register-mm --from <addr> --stake 1000000000000 --key
keys/mm.key   # 1,000 VXS
vexidus atomence cancel --from <addr> --order-id <id> --key keys/user.key
# cancel pending / reclaim after deadline
```

All Atomence writes are signed transactions -- sign client-side and submit with `vex_submitBundle`.

VSC Standards -- Native Building Blocks

VSC (Vextaris Standard Contract) standards are native operations built directly into the Vextaris L1 chain. Unlike Ethereum where every token, staking pool, or governance system requires deploying custom smart contract bytecode, Vextaris bakes these capabilities into the protocol itself.

VSCs define what the chain can do. VIPs define what you build with it.

- **VSC standards** are the building blocks -- audited once, used by everyone, no deployment needed
- **VIP programs** are the compositions -- custom JSON templates that chain VSC operations into application-specific workflows

Think of VSCs as LEGO bricks and VIPs as instruction manuals. A brick is a token. Another brick is a staking pool. A VIP says "connect these bricks in this order to build a castle."

Standard Categories

Asset Standards

Define how digital assets work on Vextaris. Create them with a single signed transaction.

Standard	Type	Ethereum Equivalent	Status
VSC-7	Fungible tokens	ERC-20	**Live**
VSC-8	Stablecoins (regulated, issuer-controlled)	FiatTokenV2 + protocol security	**Live**
VSC-21	Non-fungible tokens (NFTs)	ERC-721	**Live**
VSC-55	Multi-token (fungible + NFT in one collection)	ERC-1155	**Live**
VSC-71	Synthetic assets (stocks, perps)	--	**Planned**

Protocol Standards

Define how applications and infrastructure work on Vextaris. Also native -- no contract deployment.

Standard	Type	Ethereum Equivalent	Status
VSC-20	Programmable staking pools	StakingRewards / Convex	**Live**
VSC-88	Protocol governance + multi-sig	Governor + Gnosis Safe	**Live**
VSC-99	DAO program (token-weighted governance)	Compound Governor	**Live**
VSC-LAUNCH	Presale with anti-rug escrow	--	**Live**
VSC-REP	Token reputation scoring	--	**Live**
VSC-SAFE	Bridge + NFT safety scoring	--	**Live**

Why Both Categories Are Native (Not VIPs)

On Ethereum, you deploy a separate smart contract for every staking pool, every DAO, every governance system. Each one needs its own audit, introduces its own bugs, and has its own gas costs.

On Vextaris, these are **protocol primitives**. Every staking pool works the same way because VSC-20 is baked into the chain. Every DAO uses the same battle-tested VSC-99 code. Every multi-sig uses the same VSC-88 implementation.

This is the same approach as Solana (SPL Token is a native program) and Cosmos (bank module, staking module are native). The standards that every ecosystem needs should be audited once and shared by everyone.

VIPs are for custom application logic -- the workflows unique to YOUR project. Launch a token + create a pool + lock LP in one atomic transaction? That's a VIP composing VSC-7, VexiDEX, and VSC-88 operations.

VSC vs VIP -- When to Use What

I want to...	Use	Example
Create a token	VSC-7 (`vex_createToken`)	One signed transaction
Mint an NFT	VSC-21 (`vex_mintNft`)	One signed transaction
Create a staking pool	VSC-20 (`vex_createStakingPool`)	One signed transaction
Create a DAO	VSC-99 (`vex_createDAO`)	One signed transaction
Launch token + pool + lock LP atomically	**VIP**	Compose 3 VSC operations
Payment escrow with multi-sig release	**VIP**	Compose VSC-88 + Transfer
Clan staking with treasury governance	**VIP**	Compose VSC-20 + VSC-88
Tracked shipment with custody chain	**VIP**	Compose VSC-21 + VSC-55 + VSC-88 + Anchor

One operation? Use a VSC directly. Multiple operations that must execute atomically? Write a VIP.

EVM Comparison

Feature	Vextaris (VSC + VIP)	Ethereum (ERC + Solidity)
Asset creation	One signed transaction (VSC-7/21/55)	Deploy Solidity contract
Staking pool	One signed transaction (VSC-20)	Deploy custom staking contract
Governance	One signed transaction (VSC-88/99)	Deploy Governor + Timelock contracts
Custom logic	Register VIP (JSON template)	Deploy compiled Solidity bytecode
Audit surface	Protocol-level (audited once)	Per-contract (each needs audit)
Gas cost	Minimal (no bytecode execution)	High (contract deployment + calls)

Feature	Vextaris (VSC + VIP)	Ethereum (ERC + Solidity)
Composability	VIP template expansion	Contract-to-contract calls (delegatecall)
Readability	JSON source = audit source	Must decompile bytecode or trust verification

---

VSC-20: PROGRAMMABLE STAKING POOLS

VSC-20 enables any token to have staking pools with configurable reward rates, lock periods, and capacity limits. Two reward models: Treasury (creator-funded) and Mint (inflationary). Read more →

VSC-8: PROTOCOL-NATIVE STABLECOINS

VSC-8 is a dedicated standard for regulated fiat-backed stablecoins with 6 protocol-enforced security layers. Planned for mainnet: stablecoins only from chartered issuers, no bridge-minted stablecoins. 18 stablecoins across 10 currencies pre-mapped at genesis. Read more →

VSC-88: ON-CHAIN GOVERNANCE + MULTI-SIG

VSC-88 adds protocol-level governance with stake-weighted validator voting, plus M-of-N multi-sig shared wallets. Protocol Governance → | Multi-Sig Wallets →

EVERY WRITE IS A SIGNED TRANSACTION

Since 31 October 2026 validators refuse unsigned transactions. The write methods listed on this page (`vex_createToken`, `vex_transfer`, `vex_mintNft`, `vex_swap`, …) build an unsigned transaction on the node and are refused with `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`. Build the same operation with the SDK or the CLI (`--key <path>`), or use a wallet such as VexSpark, and submit the signed bundle with `vex_submitBundle`. Read methods (`vex_getTokenInfo`, `vex_listTokens`, `vex_getBalance`, …) work as before.

VSC-7: Fungible Tokens

VSC-7 tokens are fungible assets with configurable name, symbol, decimals, and supply. Created in a single signed transaction, immediately tradeable.

Create a Token

```
vexidus --rpc https://testnet.vextaris.io --key owner.key \
  token create --name MyToken --symbol MTK --decimals 9 --supply 1000000
--owner OWNER_ADDRESS
```

The CLI builds the token-creation operation, signs it with the key in `owner.key`, and submits it with `vex_submitBundle`.

Parameter	Description
`name`	Token name (e.g. "MyToken")
`symbol`	Ticker symbol (e.g. "MTK")
`decimals`	Decimal places (typically 9 for Vextaris)
`supply`	Total supply in **human units** (the chain adds decimals automatically)
`owner`	Address to receive initial supply

> **ABOUT DECIMALS**
>
> When creating a token, supply is in **human units**. Creating 1,000,000 tokens with 9 decimals means the chain stores 1,000,000,000,000,000 internally. You don't add zeros manually.
>
> When transferring or querying balances, amounts are in **raw units** (with decimals). 1 VXS = 1,000,000,000 raw units.

RPC Methods

Method	Description
`vex_createToken`	Create a new fungible token
`vex_getTokenInfo`	Query token info by mint address or symbol
`vex_listTokens`	List all tokens
`vex_transfer`	Transfer tokens between addresses
`vex_getBalance`	Query balance for address + token

VSC-21: Non-Fungible Tokens (NFTs)

Each NFT is a unique asset identified by `(collection_mint, token_id)`. Collections support metadata (icons, banners, royalties).

RPC Methods

Method	Description
`vex_createNftCollection`	Create a new NFT collection with metadata
`vex_mintNft`	Mint an NFT within a collection
`vex_transferNft`	Transfer an NFT to another address
`vex_burnNft`	Burn an NFT (owner only)
`vex_getNft`	Query NFT by collection + token ID
`vex_listNftsByOwner`	List NFTs owned by an address
`vex_getNftCollection`	Collection metadata and token count

VSC-55: Multi-Token Standard

A single collection containing both fungible and non-fungible token types with batch operations. A game might have "Gold Coins" (fungible) and "Legendary Sword" (non-fungible, supply=1) in one collection.

RPC Methods

Method	Description
`vex_createMultiTokenCollection`	Create a multi-token collection
`vex_defineTokenType`	Define a new token type
`vex_mintMultiToken`	Mint tokens of a type
`vex_batchMintMultiToken`	Batch mint
`vex_transferMultiToken`	Transfer tokens
`vex_batchTransferMultiToken`	Batch transfer
`vex_burnMultiToken`	Burn tokens

VexBridge v3: Cross-Chain Bridge

Cross-chain verification layer. On testnet, bridged tokens were minted as VSC-7 assets; inbound deposits are **paused** since 31 October 2026 while attestation moves to an attester quorum. Planned for mainnet: no unbacked bridged tokens -- every token is native or issued by a named party obliged to redeem it 1:1, and major foreign assets are reached by swapping through Atomence. See VexBridge documentation.

VexiDEX: On-Chain AMM

Constant-product AMM (x*y=k) built as a native protocol primitive. Pools created with a signed transaction -- no contract deployment. 0.3% swap fee.

Method	Description
`vex_createPool`	Create a liquidity pool
`vex_addLiquidity`	Add liquidity
`vex_removeLiquidity`	Remove liquidity
`vex_swap`	Swap tokens
`vex_getPool`	Pool info by token pair
`vex_listPools`	List all pools
`vex_quoteSwap`	Get swap quote (read-only)

VSC-8: Protocol-Native Stablecoin Standard

> **THE ONLY STABLECOIN STANDARD WITH ALL SIX**
>
> Quantum-resistant minting. Unbreakable governance-controlled supply caps. Mint-only emergency freezes. Epoch rate-limited allowances. Protocol-native multi-sig. Timelocked role changes with guardian override. All enforced by the protocol -- not a contract that a proxy admin can rewrite.

VSC-8 is Vextaris's dedicated stablecoin standard -- purpose-built for regulated fiat-backed tokens. Unlike EVM chains where stablecoins are smart contracts that can be exploited via proxy upgrades, flash loans, or raw calls, VSC-8 is enforced at the **protocol level** by the state machine itself.

No bytecode. No proxy admin risk. No bypassing blacklists. Six security layers that cannot be circumvented because they are the protocol.

Why VSC-8 Exists

As stablecoin regulation matures globally -- MiCA in Europe, the GENIUS Act in the US, MAS frameworks in Singapore -- issuers face increasing scrutiny over reserve protection, operational security, and key management. Current smart contract architectures cannot provide protocol-level guarantees for these requirements.

On Ethereum, anyone can deploy an ERC-20 and call it "USDC." The real Circle USDC contract has no protocol-level advantage over a fake one -- both are just bytecode. Stablecoin issuers must rely on contract-level logic that a proxy admin could theoretically rewrite.

Vextaris eliminates this entirely. VSC-8 tokens are:

- **Canonical** -- one token per stablecoin, enforced by the protocol. No impersonation possible.
- **Issuer-controlled** -- the real issuer holds all operational roles (mint, pause, blacklist) with protocol-level authority.

- **Issuer-backed** -- at mainnet, supply of a VSC-8 stablecoin comes only from its chartered issuer, who is obliged to redeem it 1:1; stablecoins are not minted by the bridge. One canonical token per stablecoin -- no fragmented wrapped versions.
- **Quantum-ready** -- optional Dilithium3 post-quantum key requirement per token, available today.

Six Security Layers

Every VSC-8 token has six independent security layers, all enforced by the state machine.

Layer 1: Epoch Rate Limiting

Every minter has an **epoch limit** -- a maximum amount that can be minted per epoch (default epoch_length: 720 blocks, ~24 minutes at ~2s). Even if a minter key is compromised, damage is capped.

Parameter	Description	Default
`epoch_limit`	Maximum mint per epoch	Configurable per minter
`epoch_length`	Blocks per epoch	720 (~24 minutes at ~2s)
`allowance_remaining`	Total remaining allowance	Set by master minter

ERC-20 comparison: Flat allowance -- a compromised minter can drain the entire allowance in one transaction. VSC-8 limits per-epoch exposure.

Layer 2: Multi-Signature for High-Value Mints

Large mints can require **N-of-M cosigner authorization** at the protocol level. Not an external multisig wallet (like Gnosis Safe on Ethereum) -- built into the token itself.

Parameter	Description
`mint_threshold`	Amount above which multi-sig is required
`required_signers`	Number of cosigners needed (N)

Parameter	Description
`authorized_cosigners`	List of authorized cosigner addresses (M)

Layer 3: Timelocked Role Changes

All role changes (transferring owner, master_minter, pauser, or blacklister) are subject to a **configurable timelock**. During the delay, the owner or any guardian can cancel the change.

Operation	Timelock	Emergency Override
Transfer any role	Yes (default 720 blocks)	Owner or guardian can cancel
Add minter	Yes	Owner or guardian can cancel
Remove minter	**Immediate**	Emergency response shouldn't be delayed
Full handover	**Always timelocked**	Highest-impact operation

ERC-20 comparison: No timelock. A compromised admin key can instantly reassign all roles.

Layer 4: Guardian Emergency Freeze

Designated **guardian addresses** can immediately freeze minting without affecting existing transfers. This allows security monitoring systems to halt minting if an anomaly is detected, without disrupting holders.

State	Minting	Transfers
Normal	Allowed	Allowed
Paused (`paused=true`)	Blocked	**Blocked**
Mint frozen (`mint_frozen=true`)	**Blocked**	Allowed

- Any guardian can freeze minting **immediately** (no timelock)
- Only the owner can unfreeze
- `paused` and `mint_frozen` are independent controls

Layer 5: Quantum-Resistant Key Requirement

Issuers can enable **Dilithium3 post-quantum signatures** for all minting operations. When enabled, minter transactions signed with classical Ed25519 keys are rejected -- only Dilithium3-signed operations are accepted.

This makes VSC-8 the only stablecoin standard in crypto with native post-quantum security.

Layer 6: Governance-Controlled Supply Cap

An optional **hard supply cap** that only governance can increase. The token owner cannot raise it unilaterally. This provides an additional safety net -- even with valid minter keys, the total supply cannot exceed the governance-approved limit.

VSC-8 vs ERC-20 vs SPL Token

Security Feature	ERC-20 (Ethereum)	SPL Token (Solana)	VSC-8 (Vextaris)
Minter allowance	Flat number, drainable in 1 tx	Single authority, no allowance	Epoch rate-limited (max per window)
Multi-sig for minting	External (Gnosis Safe)	No native support	Protocol-native, configurable threshold
Timelocked role changes	No	No	Configurable delay, guardian can cancel
Emergency mint freeze	No (only full pause)	Freeze = freeze everything	Mint-only freeze, transfers unaffected
Quantum-resistant keys	No	No	Dilithium3 requirement per token
Supply hard cap	No (infinite mint if compromised)	Optional but owner-controlled	Governance-controlled, owner cannot increase
Proxy/upgrade exploit	Proxy admin can rewrite all rules	N/A	Not applicable (protocol-native)
Blacklist bypass	Flash loan/raw call can bypass	N/A	Protocol-level, checked before any state change

Security Feature	ERC-20 (Ethereum)	SPL Token (Solana)	VSC-8 (Vextaris)
Symbol squatting	Anyone can deploy "USDC"	Anyone can create "USDC"	Reserved symbols, only governance can assign

---

Issuer Roles

VSC-8 follows the **Circle FiatTokenV2 role model** -- four distinct roles with separation of concerns:

Role	Responsibility	Can Be Transferred
Owner	Transfer any role, configure timelock, unfreeze minting	Yes (timelocked)
Master Minter	Add/remove minters, set allowances and epoch limits	Yes (timelocked)
Pauser	Pause/unpause all transfers	Yes (timelocked)
Blacklister	Add/remove addresses from blacklist	Yes (timelocked)
Guardian	Emergency mint freeze (immediate, no timelock)	Added by owner

---

Genesis Stablecoins

Vextaris launches with **18 pre-mapped stablecoins** across 10 currencies and 6 continents. Each has a canonical protocol-enforced address, source chain contract mappings, and the "Official Stablecoin" badge in explorers.

USD Stablecoins

Token	Issuer	Regulation	Chains
USDC	Circle	US (multiple licenses), MiCA EMI	Ethereum, Solana, Base, Arbitrum, Polygon, Avalanche
USDT	Tether	BVI	Ethereum, Tron, Solana, Arbitrum, Polygon

Token	Issuer	Regulation	Chains
PYUSD	Paxos/PayPal	NYDFS	Ethereum, Solana
FDUSD	First Digital	Hong Kong	Ethereum, BNB Chain
USDe	Ethena	--	Ethereum
DAI	MakerDAO	Decentralized	Ethereum

EUR Stablecoins (MiCA Era)

Token	Issuer	Regulation	Chains
EURC	Circle	MiCA EMI (France)	Ethereum, Solana, Base
EURCV	SG-Forge (Societe Generale)	MiCA EMI (French ACPR)	Ethereum
EURe	Monerium	MiCA EMI (Iceland)	Ethereum, Polygon, Arbitrum
EURS	STASIS	Malta regulated	Ethereum, Polygon, Arbitrum

Asia-Pacific

Token	Currency	Issuer	Regulation	Chains
JPYC	JPY	JPYC Inc.	Japan Payment Services Act	Ethereum, Polygon, Avalanche
XSGD	SGD	StraitsX	MAS (Singapore)	Ethereum, Polygon, Avalanche, Arbitrum
IDRT	IDR	PT Rupiah Token Indonesia	Indonesian ecosystem	Ethereum, BNB Chain, Polygon

Americas

Token	Currency	Issuer	Regulation	Chains
BRZ	BRL	Transfero	Swiss entity	Ethereum, Solana, BNB, Polygon, Avalanche
CADC	CAD	PayTrie	Canadian MSB	Ethereum, Polygon, Arbitrum, Base

Europe, Africa & Oceania

Token	Currency	Issuer	Regulation	Chains
TRYB	TRY	BiLira	Self-attested	Ethereum, Avalanche, BNB, Polygon
ZARP	ZAR	Rand Reserve	South African	Ethereum, Base, Polygon
NZDS	NZD	Techemynt	NZ FSP registered	Ethereum

Reserved Symbols

An additional **18 symbols** are reserved for stablecoins that are pre-launch, institutional-only, or awaiting regulatory clarity. These cannot be registered as VSC-7 tokens -- they are held exclusively for future VSC-8 issuance.

Reserved: GBPA, KRWQ, MXNT, CNHT, BIDR, CNGN, DDSC, AEUR, TGBP, ARC, TUSD, FRAX, LUSD, BUSD, GUSD, USDP, CUSD, SUSD, RAI, GHO.

Cross-Chain Stablecoins

At mainnet, a stablecoin on Vextaris is only ever **issued by its chartered issuer**. The bridge does not mint stablecoins.

- **Supply** -- new VSC-8 supply is minted only by the issuer's authorized minters, under the security layers above. The issuer is the party obliged to redeem it 1:1.
- **Moving value across chains (planned for mainnet)** -- until an issuer runs its own mint/redeem on Vextaris, users move between a stablecoin on another chain and an asset on Vextaris by **swapping through Atomence**, with escrow and refund guarantees on every leg -- not by bridging the foreign token in.

- **Source chain mappings** -- each VSC-8 token records the issuer's official contract addresses on other chains. They identify the genuine token (for badges, impersonation detection and reputation scoring). They do not authorize anyone but the issuer to mint.

> **TESTNET**
>
> On testnet, an earlier VexBridge path could credit bridged stablecoin deposits to the canonical VSC-8 token. Inbound bridge deposits are **paused on testnet** (since 2026-10-31), and bridge-minted stablecoins are **not part of the mainnet model** -- see VexBridge.

Impersonation Detection

If a token's symbol matches a reserved stablecoin symbol but the source contract address is **not** in the official source mappings, the token is flagged with an impersonation warning. It will not receive the "Official Stablecoin" badge and will score 0 points on the verification factor of the reputation system.

Launchpad Origin Detection

Tokens originating from known launchpad contracts (e.g., pump.fun on Solana) are automatically tagged with `launch_origin: "launchpad"` and receive the "Memecoin" badge in explorers.

Listing Process for Stablecoin Issuers

Path A: Pre-Mapped Issuers (Genesis Stablecoins)

For the 18 stablecoins already pre-mapped at genesis, the process is streamlined:

Step 1 -- Initial Contact

Reach out to the Vexidus Foundation or respond to our outreach. Your token is already registered with a canonical address, source chain mappings, and the "Official Stablecoin" badge.

Step 2 -- Legal Verification

Prove your identity as the authorized representative of the issuing entity. Accepted methods:

- **Domain verification** -- publish a TXT record or `.well-known/vexidus.json` on your official domain containing a Vextaris-provided challenge string
- **Source chain signature** -- sign a message from the admin/owner address of your token's contract on Ethereum, Solana, or another source chain
- **Corporate documentation** -- provide incorporation documents, regulatory licenses, and authorized signatory proof (for regulated issuers)

Two of the three methods are required for handover.

Step 3 -- Key Ceremony

Generate your operational keypairs:

- **Ed25519** (standard) -- compatible with existing infrastructure
- **Dilithium3** (optional, recommended) -- post-quantum security for minting operations

The Vexidus Foundation can assist with key generation or the issuer can generate independently using the Vextaris SDK or CLI.

Step 4 -- Role Handover

The Foundation executes a `StableHandover` transaction, which is **always timelocked** (minimum 720 blocks / ~24 minutes at ~2s). During the timelock period:

- The handover details are publicly visible on-chain
- The Foundation or any guardian can cancel if an issue is detected
- After the timelock expires, all four roles (owner, master_minter, pauser, blacklister) transfer atomically to the issuer's keys

Step 5 -- Configuration

The issuer now has full autonomous control. Recommended first steps:

1. Configure minter addresses and epoch limits
2. Set up guardian addresses for emergency freeze capability
3. Optionally enable multi-sig for high-value mints
4. Optionally enable Dilithium3 quantum key requirement
5. Set a governance-approved supply cap (via Foundation governance initially)

Step 6 -- Operational

The issuer operates their stablecoin independently. The Vextaris protocol enforces all security layers automatically. No further Foundation involvement is required for day-to-day operations.

Path B: New Stablecoin Issuers

For stablecoins not in the genesis registry:

Step 1 -- Application

Submit an application to the Vexidus Foundation including:

- Token name, symbol, decimals
- Issuing entity name and jurisdiction
- Regulatory licenses held (if any)
- Source chain contract addresses (official deployments on other chains)
- Intended use case and target market
- Reserve attestation or audit reports

Step 2 -- Governance Review

The application is reviewed by the Vexidus Foundation (during testnet/early mainnet) or by protocol governance vote (post-activation via VSC-88).

Review criteria:

- **Regulatory standing** -- licensed issuers are prioritized
- **Reserve transparency** -- proof of reserves or regular attestations
- **Technical readiness** -- source chain contracts are deployed and operational
- **Market demand** -- demonstrated user base or clear market need
- **Symbol uniqueness** -- no conflict with existing or reserved symbols

Step 3 -- Token Creation

Upon approval, the governance authority executes `CreateStableToken`. This:

- Creates a canonical VSC-8 token with a deterministic mint address
- Registers the issuer's source chain contract mappings
- Adds the symbol to the reserved list

- Assigns the "Official Stablecoin" badge

Step 4 -- Handover & Configuration

Same as Path A Steps 2-6. Legal verification, key ceremony, timelocked handover, issuer configuration.

Timeline

Phase	Path A (Pre-Mapped)	Path B (New Issuer)
Initial contact	Day 1	Day 1
Legal verification	Days 2-5	Days 2-5
Governance review	N/A (already approved)	Days 5-14
Key ceremony	Day 5-6	Day 14-15
Timelocked handover	Day 6-7 (~24min timelock)	Day 15-16
Operational	Day 7+	Day 16+

RPC Endpoints

Write Operations

Method	Description	Auth
`vex_createStableToken`	Create a new VSC-8 token	Governance only
`vex_stableMint`	Mint tokens	Authorized minter
`vex_stableBurn`	Burn tokens	Any holder
`vex_stablePause`	Pause/unpause transfers	Pauser role
`vex_stableBlacklist`	Add/remove from blacklist	Blacklister role

Method	Description	Auth
`vex_stableTransferRole`	Transfer a role (timelocked)	Owner only
`vex_stableConfigureMinter`	Add/remove/configure minter	Master minter
`vex_stableHandover`	Atomic handover of all roles (timelocked)	Owner only

Read Operations

Method	Description
`vex_getStablecoinInfo`	Full stablecoin state (roles, config, security settings)
`vex_listStablecoins`	List all VSC-8 tokens
`vex_isBlacklisted`	Check if an address is blacklisted for a specific token

Try It: List All Genesis Stablecoins

Query the live testnet to see all 18 pre-mapped VSC-8 stablecoins:

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_listStablecoins",
    "params": [50],
    "id": 1
  }' | jq .
```

Try It: Query a Specific Stablecoin

Pass a symbol or mint address to inspect its full state -- roles, security config, source chain mappings:

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_getStablecoinInfo",
    "params": ["USDC"],
    "id": 1
  }' | jq .
```

LIVE TESTNET

These examples hit the live Vextaris testnet at `testnet.vextaris.io`. Run them from any terminal to see real VSC-8 token state. All 18 stablecoins are queryable with full security configuration.

Reputation Scoring

VSC-8 tokens benefit from **Factor 8 (Verification Status)** in the Vextaris reputation system, worth up to 15 points:

Status	Points	Description
Issuer-Held	15	Issuer has completed handover and controls the token
Verified	12	Governance has verified the issuer
Pending	10	Pre-mapped at genesis, awaiting issuer handover
Native VSC-7	8	Standard native token (not VSC-8)
Bridged (clean)	5	Bridged token with no warnings
Launchpad origin	3	Originated from a known launchpad
Impersonation	0	Symbol matches reserved but source contract doesn't match

This means VSC-8 stablecoins with completed issuer handover automatically receive the highest verification score, while impersonation attempts are scored at zero.

Contact

Stablecoin issuers interested in listing on Vextaris can reach the Foundation at:

- **Email**: TBA
- **Twitter/X**: @VexidusCorp
- **GitHub**: github.com/vexidus-labs/vexidus-sdk

VSC-20: Programmable Staking Pools

VSC-20 is a protocol-level standard for creating programmable staking pools on Vextaris. Any token holder can create a staking pool for any VSC-7 fungible token with configurable reward rates, lock periods, and capacity limits.

Unlike validator delegation staking which is limited to VXS, VSC-20 pools are **general-purpose** -- any project, community, or individual can create one.

Why VSC-20?

- **DeFi infrastructure** -- Every L1 needs programmable staking
- **Project incentives** -- Token projects can incentivize holding with yield
- **Community pools** -- Groups can create staking pools for their members
- **Game economies** -- Games can reward players with staking yields on in-game tokens
- **Composable** -- VexFi lending/yield protocols can build on top of pools

How It Works

Create a Pool

```
vexidus staking-pool create \
  --stake-token VXS \
  --reward-token VXS \
  --reward-rate 1000000000 \
  --lock-period 2592000 \
  --source treasury \
  --from <wallet>
```

Fund the Treasury

For treasury-funded pools (non-inflationary), the creator deposits reward tokens upfront:

```
vexidus staking-pool fund --pool <id> --amount 1000000000000 --from
<wallet>
```

Stake Tokens

```
vexidus staking-pool stake --pool <id> --amount 500000000000 --from
<wallet>
```

Claim Rewards

```
vexidus staking-pool claim --pool <id> --from <wallet>
```

Pool Configuration

Parameter	Description	Example
`stake_token`	Token users stake	VXS, MSHU, any VSC-7
`reward_token`	Token paid as rewards	Can be same or different
`reward_rate`	Rewards per day per staked token (raw units)	`1000000000` = 1 token/day
`lock_period`	Minimum lock duration in seconds	`2592000` = 30 days
`min_stake`	Minimum stake amount	`1000000000` = 1 token
`max_stake_per_user`	Per-user cap (optional)	Prevents whale dominance
`max_pool_capacity`	Total pool cap (optional)	Creates scarcity
`start_time`	When staking opens (optional)	Scheduled launches
`end_time`	When pool closes (optional)	Seasonal pools

Reward Sources

Treasury (Recommended)

The pool creator deposits reward tokens upfront. Rewards are distributed from this treasury. The pool pauses when treasury runs dry. The creator can top up at any time.

This is the safe, non-inflationary model. Suitable for most use cases.

Mint

The pool mints new reward tokens as rewards. Requires the pool creator to hold mint authority on the reward token. This is inflationary -- use with caution.

Suitable for: Protocol-level incentive programs where controlled inflation is part of the tokenomics design.

Reward Calculation

Rewards accrue linearly over time:

```
rewards = staked_amount × reward_rate × time_elapsed / 86400
```

- Rewards compound when claimed (claim, then re-stake)
- Partial unstaking is supported
- Rewards continue accruing even if not claimed

Security

- **Lock enforcement** -- Cannot unstake before the lock period expires
- **Treasury protection** -- Treasury pools never distribute more than deposited
- **Mint authority check** -- Mint pools verify creator has mint authority
- **Creator limits** -- Maximum 10 pools per creator to prevent spam
- **Overflow protection** -- All calculations use checked arithmetic

Operations

Disc	Operation	Gas	Description
90	`CreateStakingPool`	150,000	Create a new pool
91	`FundStakingPool`	75,000	Deposit reward tokens
92	`StakeToPool`	100,000	Stake tokens
93	`UnstakeFromPool`	100,000	Withdraw staked tokens
94	`ClaimPoolRewards`	75,000	Claim accumulated rewards
95	`UpdateStakingPool`	50,000	Modify pool settings
96	`CloseStakingPool`	100,000	Close empty pool

IntentVM Support

Staking operations can be expressed as natural language intents:

```
"Stake 500 VXS into Cherry Blossom pool"
"Claim my staking rewards"
"Unstake all my MSHU from Torii Domain"
```

IntentVM resolves pool names to IDs and handles multi-pool operations atomically.

Transaction Cost

~$0.00003 per staking operation at current gas prices. Making micro-staking economically viable -- stake $1 worth of tokens without fees eating the yield.

Vextaris Token Metadata Standard

> Version 1.0

Overview

The Vextaris Token Metadata Standard defines the canonical schema for token metadata stored on-chain in RocksDB. It covers native tokens (VSC-7), bridged tokens (testnet), and the VXS native token, with clear mutability rules and a three-tier resolution system.

Three-Tier Metadata Resolution

1. **On-chain** (RocksDB) -- source of truth, highest priority
2. **Off-chain URI** (IPFS/Arweave) -- extended metadata via `metadata_uri`
3. **Community registry** (future GitHub repo) -- fallback for tokens missing icons

---

On-Chain Metadata Schema

Stored as JSON in RocksDB via `token:<mint_hex>` keys.

Immutable Fields (set at creation, never changeable)

Field	Type	Max Length	Description
`name`	string	64	Token name
`symbol`	string	10	Token ticker
`decimals`	u8	-	Decimal precision
`total_supply`	string	-	Total supply (raw units)

Field	Type	Max Length	Description
`initial_supply`	string/u64	-	Initial supply at creation
`creator`	string	-	Creator address (0x hex)
`mint_address`	string	-	Mint address (0x hex or Vx1)
`mint_address_hex`	string	-	Mint address (0x hex, always present)
`created_at`	u64	-	Unix timestamp of creation
`standard`	string	-	`VSC-7`, `Native`, `VSC-21`, `VSC-55`
`features.mintable`	bool	-	Whether new tokens can be minted
`features.burnable`	bool	-	Whether tokens can be burned
`features.pausable`	bool	-	Whether transfers can be paused
`features.freezable`	bool	-	Whether accounts can be frozen

Mutable Fields (changeable if `is_mutable = true`)

Field	Type	Max Length	Description
`metadata_uri`	string	512	Off-chain JSON metadata URI (IPFS/Arweave/HTTPS)
`image`	string	512	Direct URL to token logo
`description`	string	1024	Token description
`external_url`	string	256	Project website
`twitter`	string	256	Twitter/X link
`discord`	string	256	Discord invite link
`telegram`	string	256	Telegram link
`github`	string	256	GitHub repo link

System Fields (set by protocol, not user-editable)

Field	Type	Description
`is_mutable`	bool	`true` = metadata can be updated; `false` = permanently locked. Default: `true` for native tokens, `false` for bridged tokens.
`badge`	string	`native` (VXS), `bridge` (bridged tokens). Set by protocol.
`origin`	string	`bridged` for bridge-created tokens
`source_chain`	string	Origin chain (bridged tokens only)
`source_contract`	string	Origin contract address (bridged tokens only)
`bridge_tier`	string	`tier2` or `tier3` (bridged tokens only)

Mutability Lifecycle

```
Token Created --> is_mutable: true --> Creator updates metadata freely
                                              |
                                              v
                                   Lock metadata (signed, one-way)
                                              |
                                              v
                                   is_mutable: false (permanent)
                                   No further updates possible
```

Rules

1. All new native tokens (VSC-7) start with `is_mutable: true`
2. All bridged tokens are created with `is_mutable: false` (permanently immutable)
3. Only the token creator can lock metadata (a signed lock-metadata transaction)
4. Once locked (`is_mutable: false`), it cannot be reversed -- ever
5. Every metadata update checks `is_mutable` before allowing changes

6. Missing `is_mutable` field is treated as `true` (backward compatibility)

Off-Chain Metadata Schema (at metadata_uri)

Compatible with the Metaplex token metadata standard:

```
{
  "name": "string (required)",
  "description": "string (required)",
  "image": "URI (required, PNG/JPG/SVG)",
  "external_url": "URI (optional)",
  "attributes": [
    { "trait_type": "string", "value": "string|number" }
  ],
  "properties": {
    "files": [
      { "uri": "string", "type": "MIME type" }
    ],
    "category": "string (image|video|audio)"
  }
}
```

Token Type Examples

Native Token (VSC-7)

```
{
  "name": "My Token",
  "symbol": "MTK",
  "decimals": 9,
  "total_supply": "1000000000000000000",
  "initial_supply": 1000000000,
  "creator": "0xabc...",
  "mint_address": "0xdef...",
  "created_at": 1739477600,
  "standard": "VSC-7",
  "is_mutable": true,
  "features": {
    "mintable": false,
    "burnable": true,
    "pausable": false,
    "freezable": false
  },
  "image": "https://example.com/logo.png",
  "description": "A community token",
  "twitter": "https://x.com/mytoken"
}
```


Bridged Token

Bridged tokens exist on testnet only. Inbound bridge deposits are paused since 31 October 2026, and the mainnet plan is no unbacked bridged tokens -- every token native or issued by a named party obliged to redeem it 1:1.

```
{
  "name": "Bridged USDC (ethereum)",
  "symbol": "USDC",
  "decimals": 9,
  "origin": "bridged",
  "source_chain": "ethereum",
  "source_contract": "0xa0b8...",
  "bridge_tier": "tier2",
  "mint_address": "Vx1abc...",
  "mint_address_hex": "0xdef...",
  "created_at": 1739477600,
  "standard": "VSC-7",
  "is_mutable": false,
  "badge": "bridge"
}
```


VXS (Native Token)

VXS metadata is hardcoded in the RPC layer (not stored in token registry). It cannot be modified by a metadata update.

Updating and Locking Metadata

Metadata changes are signed transactions from the token creator, submitted with `vex_submitBundle`. Validators refuse unsigned transactions, so the convenience methods `vex_updateTokenMetadata` and `vex_lockTokenMetadata` -- which build an unsigned transaction on the node from a sender address -- are refused with `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`.

With the CLI (signs with `--key`):

```
# Update mutable metadata fields. Creator only. Fails if is_mutable =
false.
vexidus --rpc https://testnet.vextaris.io --key creator.key \
  token update-metadata --mint <MINT> --sender <CREATOR_ADDRESS> \
  [--metadata-uri <URI>] [--image <URL>] [--description <TEXT>] [--
external-url <URL>] \
  [--twitter <URL>] [--discord <URL>] [--telegram <URL>] [--github <URL>]

# Permanently lock token metadata. Creator only. One-way.
vexidus --rpc https://testnet.vextaris.io --key creator.key \
  token lock-metadata --mint <MINT> --sender <CREATOR_ADDRESS>
```

The same operations can be built and signed with the SDK or a wallet.

Reputation Impact

The `metadata_integrity` reputation factor (max 10 points) rewards tokens with complete, locked metadata:

Condition	Points
Metadata permanently locked (`is_mutable: false`)	10/10
Has image + description + at least one social link, but mutable	5/10
Incomplete metadata	0/10

Metadata Sanitization (VSC-SAFE)

All metadata submitted on-chain is validated to prevent XSS, injection, and phishing attacks. This applies to token creation, metadata updates, NFT minting, and collection creation.

URI Field Validation

All URI fields (`metadata_uri`, `image`, `base_uri`, `icon_uri`, `banner_uri`, `external_url`) must pass:

Rule	Details
Allowed schemes	`https://`, `ipfs://`, `ar://` (Arweave)
Blocked schemes	`data:` (inline payloads), `javascript:`, `http://` (insecure)
Pattern scanning	Rejects `<script`, `onerror=`, `onclick=`, `<iframe`, `eval(`, `document.cookie`, `&#` (HTML entities), and 17+ dangerous patterns
Max length	512 characters (URIs), 256 characters (external URLs)

Text Field Validation

All text fields (`description`, attribute keys/values) must pass:

Rule	Details
Null bytes	Rejected (prevents string truncation attacks)
Pattern scanning	Same 17+ dangerous patterns as URI validation
Max length	2,048 chars (descriptions), 64 chars (attribute keys), 256 chars (attribute values)
Max attributes	50 per token

What Gets Rejected

```
data:text/html,<script>alert(1)</script>    -> Blocked: data: URI
https://evil.com/<script>alert(1)           -> Blocked: <script pattern
https://evil.com/img.png?onerror=alert(1)   -> Blocked: onerror= pattern
javascript:alert(1)                         -> Blocked: not https/ipfs/ar
Hello\0World                                -> Blocked: null bytes
```

What Passes

```
https://example.com/token-logo.png          -> OK
ipfs://QmSomeHash/metadata.json             -> OK
ar://abc123def456                           -> OK
A beautiful NFT collection of 10,000 pieces -> OK
```

Sanitization runs at the state machine level, so it applies regardless of whether metadata is submitted via RPC, SDK, CLI, or any other path.

Future

- **Community token registry**: `vexidus-labs/vexidus-token-list` GitHub repo with JSON files per token, PR-based curation, CI validation
- **Governance override**: Allow protocol governance (VSC-88) to force-update metadata on scam tokens

VSC-88: Protocol Governance

VSC-88 adds on-chain protocol governance to Vextaris, enabling validators to propose and vote on protocol parameter changes, treasury spending, and community signal votes. It extends the existing VexVisor upgrade governance into a general-purpose system.

How It Works

1. **Propose** -- An active validator submits a proposal (parameter change, treasury spend, text, contract safety, collection safety, state correction, account/token freeze, or emergency mint)
2. **Vote** -- Validators vote with stake-weighted power (same model as VexVisor upgrade voting)
3. **Approve** -- Quorum met + majority yes = Approved
4. **Execute** -- Anyone calls execute to apply the approved change on-chain

Every proposal follows the same propose → vote → approve → execute lifecycle, with stake-weighted voting and quorum enforcement.

Proposal Types

Parameter Change

Adjusts on-chain protocol parameters. Auto-executed on approval -- the new value takes effect immediately.

Governable parameters:

Parameter	Description	Default	Bounds
`gas_price`	Base gas price (nanoVXS/gas)	10	1 -- 1,000
`max_txs_per_block`	Max transactions per block	50,000	100 -- 100,000
`min_stake`	Minimum VXS to become a validator	1,000 VXS	100 -- 100,000 VXS

Parameter	Description	Default	Bounds
`max_commission_bps`	Max validator commission (basis points)	5,000 (50%)	100 -- 10,000
`voting_period_blocks`	Governance voting duration	500 blocks	100 -- 50,400
`quorum_percent`	Required approval quorum	67%	51 -- 100%
`foundation_reward_bps`	Foundation share of block rewards	1,500 (15%)	0 -- 5,000
`foundation_fee_bps`	Foundation share of transaction fees	2,000 (20%)	0 -- 5,000
`foundation_sponsor_first_n`	"First N free" sponsored transactions per new account	10	0 -- ∞ (0 disables)

Not governable (require binary upgrade via VexVisor):

- Block time (consensus-level coordination)
- Total VXS supply (1.618B: genesis allocation plus the 10-year block-reward schedule; never burned)
- Block reward schedule

Treasury Spend

Transfers VXS from a governance-controlled treasury account to a specified recipient. Auto-executed on approval.

Treasury accounts (governance-controlled):

- Foundation Treasury -- 161.8M VXS (Vexidus Corp operations)
- Ecosystem & DeFi -- 242.7M VXS (grants, LP incentives)

Not governance-controlled: Public Distribution, Team (vesting), Marketing, SAFT, Testnet Rewards, Staking Reserve.

Text Proposal

Signal vote with no on-chain effect. Used for community sentiment ("should we prioritize feature X?"). Recorded on-chain for transparency.

Contract Safety (VSC-SAFE)

Reclassify a bridge contract's safety level. Used to upgrade legitimate projects to Verified status, or block known scam contracts. Auto-executed on approval -- the safety registry is updated immediately.

Required fields: `--safety-chain`, `--safety-contract`, `--safety-level` (Verified/Standard/Warning/Blocked)

See VexBridge Contract Safety for details on the scoring system.

Collection Safety (VSC-SAFE)

Flag or freeze a bridged NFT collection. Used to protect users from malicious collections with phishing metadata or impersonation attempts. Auto-executed on approval.

Required fields: `--safety-collection` (collection mint address)

Account Freeze

Freeze or unfreeze a specific account address. Frozen accounts cannot send transactions. Used for emergency response to compromised accounts or protocol exploits.

Required fields: `--freeze-target` (account address), `--freeze-action` (true=freeze, false=unfreeze)

Token Freeze

Freeze or unfreeze a token mint. Frozen tokens cannot be transferred by any holder. Used to halt trading of compromised or malicious tokens.

Required fields: `--freeze-token` (token mint address), `--freeze-action` (true=freeze, false=unfreeze)

State Correction

Directly modify on-chain state to fix data integrity issues. Supports four correction types:

Correction Type	What It Does	Required Fields
`set_balance`	Set an account's token balance	`--correction-account`, `--correction-amount`, optional `--correction-token`
`set_pool_reserve_a`	Set a pool's token A reserve	`--correction-pool-pair`, `--correction-amount`
`set_pool_reserve_b`	Set a pool's token B reserve	`--correction-pool-pair`, `--correction-amount`
`set_token_supply`	Set a token's total supply	`--correction-token`, `--correction-amount`

Pool corrections automatically recalculate LP supply using `sqrt(reserve_a * reserve_b)` to maintain the AMM invariant.

```
# Example: rebalance a mispriced pool
vexidus governance create-proposal \
  --type state-correction \
  --title "Fix VXS/BNB pool reserves" \
  --description "Correct VXS reserve to match market price" \
  --correction-type set_pool_reserve_a \
  --correction-amount 122452151947925 \
  --correction-pool-pair "TOKEN_A_HEX_TOKEN_B_HEX"
```

Emergency Mint

Mint VXS to a specified account for emergency protocol operations. Strictly controlled and auditable on-chain.

Required fields: `--correction-account` (recipient), `--correction-amount` (raw VXS amount)

Proposal Lifecycle

```
Created (Voting)
    |
    +-- Validators vote (stake-weighted)
    |       |
    |       +-- Quorum met + majority yes --> Approved
    |       |       |
    |       |       +-- ExecuteProposal called --> Executed
    |       |
    |       +-- Quorum met + majority no --> Rejected
    |
    +-- Voting period expires without quorum --> Expired
    |
    +-- Proposer cancels --> Cancelled
```

A proposal reaches Approved once it meets quorum with a majority of yes votes, after which anyone can execute it.

CLI Usage

Create a Proposal

```
# Parameter change: reduce gas price to 5
vexidus governance create-proposal \
  --type param \
  --title "Reduce gas price" \
  --description "Lower base gas price from 10 to 5 nanoVXS" \
  --param-key gas_price \
  --param-value 5

# Treasury spend: fund a grant
vexidus governance create-proposal \
  --type treasury \
  --title "Ecosystem Grant" \
  --description "Fund the VexSwap liquidity program" \
  --recipient Vx0RecipientAddress... \
  --amount 500000000000000

# Text proposal: signal vote
vexidus governance create-proposal \
  --type text \
  --title "Reduce block time?" \
  --description "Community sentiment check on reducing block time from 2s
to 1s"

# Contract safety: verify a bridge contract
vexidus governance create-proposal \
  --type contract-safety \
  --title "Verify USDC on Base" \
  --description "Circle's official USDC deployment on Base" \
  --safety-chain base \
  --safety-contract 0x833589fcd6edb6e08f4c7c32d4f71b54bda02913 \
  --safety-level Verified

# Collection safety: flag a malicious NFT collection
vexidus governance create-proposal \
  --type collection-safety \
  --title "Freeze phishing collection" \
  --description "Collection metadata contains XSS payloads" \
  --safety-collection Vx0CollectionMintAddress...
```

Vote

```
vexidus governance vote --id 1 --approve
vexidus governance vote --id 1 --reject
```

Execute an Approved Proposal

```
vexidus governance execute --id 1
```

Cancel (Proposer Only)

```
vexidus governance cancel --id 1
```

View Proposals

```
vexidus governance list
vexidus governance list --status voting
vexidus governance show --id 1
```

RPC Methods

Method	Type	Description
`vex_createProposal`	Write	Submit a governance proposal
`vex_voteProposal`	Write	Vote on an active proposal
`vex_executeProposal`	Write	Execute an approved proposal
`vex_cancelProposal`	Write	Cancel a proposal (proposer only)
`vex_getProposal`	Read	Get proposal details by ID
`vex_listProposals`	Read	List proposals (filter by status)

Example: Create a Proposal

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_createProposal",
    "params": [
      "VALIDATOR_ADDRESS",
      "ParameterChange",
      "Reduce gas price",
      "Lower base gas price from 10 to 5 nanoVXS",
      "gas_price",
      "5",
      null,
      null
    ],
    "id": 1
  }' | jq .
```

Example: List Active Proposals

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_listProposals",
    "params": ["voting"],
    "id": 1
  }' | jq .
```

SDK Usage

```rust
use vexidus_sdk::BundleBuilder;

// Create a parameter change proposal
let bundle = BundleBuilder::new(&validator_addr)?
    .create_proposal(
        "ParameterChange",
        "Reduce gas price",
        "Lower from 10 to 5 nanoVXS",
        Some("gas_price".to_string()),
        Some("5".to_string()),
        None, None,
    )
    .nonce(nonce)
    .sign(&wallet);

// Vote on a proposal
let bundle = BundleBuilder::new(&validator_addr)?
    .vote_proposal(1, true)  // proposal_id=1, approve=true
    .nonce(nonce)
    .sign(&wallet);

// Execute an approved proposal
let bundle = BundleBuilder::new(&any_addr)?
    .execute_proposal(1)
    .nonce(nonce)
    .sign(&wallet);
```

Constraints

Constraint	Value
Max active proposals	5
Title length	1 -- 128 characters
Description length	1 -- 1,024 characters
Who can propose	Active, non-jailed validators
Who can vote	Active, non-jailed validators

Constraint	Value
Vote weight	Validator total stake (own + delegated)
Who can execute	Anyone (approval is the authorization)
Who can cancel	Original proposer only (unless already Executed)
Gas: CreateProposal	80,000
Gas: VoteProposal	40,000
Gas: ExecuteProposal	60,000
Gas: CancelProposal	40,000

Relationship to VexVisor

VexVisor handles **binary upgrades** -- scheduling, voting, halt-at-height, and auto-restart via the VexVisor upgrade workflow.

VSC-88 handles **protocol governance** -- parameter changes, treasury spending, and signal votes.

Both use stake-weighted voting, quorum logic, and double-vote prevention. They are complementary systems.

Evolution

VSC-88 protocol governance is the foundation for the upcoming VSC-99 DAO Program, which extends the same voting engine to token-weighted community governance for user-created DAOs.

VSC-88: Multi-Sig Wallets

Multi-sig accounts require M-of-N approvals before executing transactions. This is the second tier of VSC-88, providing shared account control for teams, treasuries, and compliance-controlled assets.

Use Cases

- **Foundation treasury** -- 3-of-5 core team approval for spending
- **Stablecoin issuer accounts** -- 2-of-3 compliance officers
- **Bridge oracle key rotation** -- 4-of-7 oracle operators
- **Project team treasuries** -- Shared control with no single point of failure
- **Escrow accounts** -- Neutral third-party involved in release

How It Works

1. **Create** -- Define signers (2-20 addresses) and threshold (M approvals required)
2. **Propose** -- Any signer proposes a transaction (proposer auto-approves)
3. **Approve** -- Other signers approve. When threshold is met, the transaction executes automatically
4. **Execute** -- Inner operations run with the multi-sig address as the sender

The multi-sig can execute **any operation** -- transfers, token creation, staking, DEX operations, bridge actions, and more.

Multi-Sig Address

The multi-sig account address is **deterministic** -- derived from the sorted signer list and threshold. This means you can compute the address locally before creating it on-chain.

> **INFO**

> The same set of signers with the same threshold always produces the same multi-sig address, regardless of who creates it or when.

CLI Usage

Create a Multi-Sig Account

```
# Create a 2-of-3 multi-sig
vexidus multisig create \
  --signers Vx0Signer1...,Vx0Signer2...,Vx0Signer3... \
  --threshold 2 \
  --label "Team Treasury"
```

Propose a Transaction

```
# Propose a VXS transfer from the multi-sig
vexidus multisig propose \
  --account Vx0MultiSigAddr... \
  --operations '[{"Transfer":
{"to":"Vx0Recipient...","token_symbol":"VXS","amount":50000000000}}]' \
  --description "Monthly operational budget"
```

Approve a Pending Transaction

```
# Second signer approves -- if threshold met, tx executes automatically
vexidus multisig approve \
  --account Vx0MultiSigAddr... \
  --tx-id 0
```

Revoke an Approval

```
# Revoke before execution
vexidus multisig revoke \
  --account Vx0MultiSigAddr... \
  --tx-id 0
```

View Multi-Sig Details

```
# Show account info (signers, threshold, label)
vexidus multisig show --account Vx0MultiSigAddr...

# List all transactions (pending, executed)
vexidus multisig list-txs --account Vx0MultiSigAddr...
```

RPC Methods

Method	Type	Description
`vex_createMultiSig`	Write	Create an M-of-N multi-sig account
`vex_proposeMultiSigTx`	Write	Propose a transaction (operations as JSON)
`vex_approveMultiSigTx`	Write	Approve a pending transaction
`vex_revokeMultiSigApproval`	Write	Revoke an approval before execution
`vex_getMultiSig`	Read	Get multi-sig account details
`vex_listMultiSigTxs`	Read	List transactions for a multi-sig

Example: Create a Multi-Sig

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_createMultiSig",
    "params": [
      "CREATOR_ADDRESS",
      ["SIGNER_1", "SIGNER_2", "SIGNER_3"],
      2,
      "Team Treasury"
    ],
    "id": 1
  }' | jq .
```

Example: Get Multi-Sig Info

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_getMultiSig",
    "params": ["MULTISIG_ADDRESS"],
    "id": 1
  }' | jq .
```

Response:

```
{
  "address": "Vx0MultiSig...",
  "signers": ["Vx0Signer1...", "Vx0Signer2...", "Vx0Signer3..."],
  "threshold": 2,
  "label": "Team Treasury",
  "created_at_height": 145000
}
```

SDK Usage

```rust
use vexidus_sdk::BundleBuilder;
use vexidus_types::{Address, Operation};

// 1. Create a 2-of-3 multi-sig
let bundle = BundleBuilder::new(&creator)?
    .create_multisig(
        vec![signer_a, signer_b, signer_c],
        2,
        "Team Treasury".to_string(),
    )
    .nonce(nonce)
    .sign(&creator_wallet);

// 2. Propose a transfer from the multi-sig
let ops = vec![Operation::Transfer {
    to: recipient,
    token_symbol: "VXS".to_string(),
    amount: 50_000_000_000,  // 50 VXS
}];

let bundle = BundleBuilder::new(&signer_a)?
    .propose_multisig_tx(multisig_addr, ops, "Q1 budget".to_string())
    .nonce(nonce)
    .sign(&signer_a_wallet);

// 3. Approve (second signer -- triggers execution at threshold)
let bundle = BundleBuilder::new(&signer_b)?
    .approve_multisig_tx(multisig_addr, 0)  // tx_id=0
    .nonce(nonce)
    .sign(&signer_b_wallet);

// 4. (Optional) Revoke approval before execution
let bundle = BundleBuilder::new(&signer_b)?
    .revoke_multisig_approval(multisig_addr, 0)
    .nonce(nonce)
    .sign(&signer_b_wallet);
```

Transaction Lifecycle

```
Proposed (Pending)
    |  proposer auto-approved (1/M)
    |
    +-- Signer B approves (2/M)
    |       |
    |       +-- Threshold met? --> Executed (operations run)
    |       |
    |       +-- Not yet --> still Pending
    |
    +-- Signer revokes --> remove approval, back to Pending
```

If threshold is 1 (single-signer multi-sig), the transaction executes immediately on proposal.

Constraints

Constraint	Value
Min signers	2
Max signers	20
Min threshold	2
Max threshold	Number of signers
Label length	1 -- 64 characters
Description length	1 -- 256 characters
Duplicate signers	Not allowed
Zero addresses	Not allowed
Supported inner operations	All (transfer, stake, token create, DEX, bridge, etc.)
Self-modification	Not supported (cannot change signers/threshold)

TO CHANGE SIGNERS

> Create a new multi-sig account with the updated signer set and migrate funds from the old one. This prevents malicious signer reconfiguration.

Best Practices

1. **Use odd thresholds** -- 2-of-3, 3-of-5, 4-of-7 prevent deadlocks where exactly half approve
2. **Store keys separately** -- Different devices, different people, different physical locations
3. **Label descriptively** -- The label is stored on-chain and visible in explorers
4. **Monitor pending transactions** -- Signers should regularly check `vex_listMultiSigTxs`
5. **Start with higher thresholds** -- You can always create a new multi-sig with a lower threshold, but you can't raise it without migration
6. **Test on testnet first** -- Create a test multi-sig, propose, approve, and verify execution before using for real assets

Events

Multi-sig operations emit the following on-chain events (visible via `vex_getTransactionReceipt`):

Event	When
`MultiSigCreated`	Account created with signers and threshold
`MultiSigTxProposed`	New transaction proposed by a signer
`MultiSigTxApproved`	Signer approved (includes current approval count)
`MultiSigTxExecuted`	Threshold met, inner operations executed
`MultiSigApprovalRevoked`	Signer revoked their approval

VSC-99: DAO Program

> **LIVE ON TESTNET**
>
> VSC-99 is deployed and operational on the Vextaris testnet. Create and manage DAOs at daimyodao.com or via the CLI.

VSC-99 defines a protocol-native DAO program for **token-weighted community governance** on Vextaris. Unlike smart-contract DAOs on EVM chains, VSC-99 DAOs are built into the state machine -- creators configure parameters and deploy with a single operation. No Solidity. No audits. No bytecode.

Relationship to VSC-88

```
VSC-88  (disc 60-67)  -- Protocol governance + multi-sig (validator
stake-weighted)  [LIVE]
                          | shares voting engine
VSC-99  (disc 97-103) -- DAO program (token balance-weighted, user-
created)  [LIVE]
                          | reads balances from
VSC-7   (disc 1-4)    -- Fungible token standard (voting token)  [LIVE]
```

VSC-88 is for protocol-level decisions made by validators. **VSC-99** is for community-level decisions made by token holders. VSC-99 reuses the same voting engine (proposal lifecycle, quorum math, double-vote prevention) but with token-weighted voting instead of stake-weighted.

Key Features

One-Operation DAO Creation

A DAO is created with a single `CreateDAO` operation. The creator defines all governance rules upfront:

- Which VSC-7 token determines voting power (or VXS native)
- Minimum token balance to propose / vote

- Voting period, quorum percentage, approval threshold
- Optional execution delay (timelock)
- Treasury address and per-proposal spending cap
- Allowed proposal types (treasury spend, text, config change)
- Optional elder council with veto/fast-track powers

Token-Weighted Voting

Voting power = governance token balance. If you hold 1% of the token supply, your vote carries 1% weight.

Snapshot modes:

- **At Proposal Creation** (recommended) -- Voting power locked at the block the proposal was created. Prevents vote-buying (buy tokens, vote, sell).
- **At Vote Time** -- Simpler but potentially gameable via flash loans or short-term accumulation.

Proposal Types

Type	Description	Effect
Treasury Spend	Transfer tokens from DAO treasury	Auto-executed on approval
Text Proposal	Community signal vote	No on-chain state change
Config Change	Modify DAO parameters (quorum, voting period, etc.)	DAO config updated
Council Action	Council-specific operations	Veto, fast-track, member changes

Elder Council (Optional)

DAOs can optionally configure an **elder council** -- a small group (up to 11 members) with special powers:

Power	Description
Veto	Kill any active proposal

Power	Description
Fast-Track	Skip voting period, approve immediately
Propose-Only	Restrict proposal creation to council members
Member Management	Add/remove council members (requires M-of-N council approval)

The council is optional and fully configurable. A DAO can launch with no council (pure token democracy), a council with limited powers (veto only), or a council with broad authority.

Operations

Operation	Disc	Gas	Description
`CreateDAO`	97	150,000	Deploy a new DAO with full configuration
`DAOPropose`	98	80,000	Submit a DAO proposal
`DAOVote`	99	40,000	Vote on a DAO proposal (token-weighted)
`DAOExecute`	100	60,000	Execute an approved proposal after timelock
`DAOCancel`	101	40,000	Cancel a proposal (proposer or council veto)
`DAOUpdateConfig`	102	80,000	Update DAO config (internal, via approved ConfigChange)
`DAOCouncilAction`	103	60,000	Council veto, fast-track, or member changes

RPC Methods

Method	Type	Description
`vex_createDAO`	Write	Deploy a new DAO
`vex_daoPropose`	Write	Submit a DAO proposal

Method	Type	Description
`vex_daoVote`	Write	Vote on a proposal
`vex_daoExecute`	Write	Execute an approved proposal
`vex_daoCancel`	Write	Cancel a proposal
`vex_daoCouncilAction`	Write	Council veto/fast-track/member changes
`vex_getDAO`	Read	Get DAO config by address
`vex_listDAOs`	Read	List all DAOs (optional token filter)
`vex_getDAOProposal`	Read	Get proposal details
`vex_listDAOProposals`	Read	List proposals for a DAO (optional status filter)
`vex_getDAOVotingPower`	Read	Get voting power, eligibility, and council status

CLI Commands

```
# Create a DAO
vexidus dao create \
  --name "MyProject DAO" \
  --token <governance_token_mint> \
  --min-propose 1000 \
  --min-vote 1 \
  --quorum 10 \
  --threshold 51 \
  --voting-period 7200 \
  --from <creator_address>

# Submit a treasury spend proposal
vexidus dao propose \
  --dao <dao_address> \
  --type treasury_spend \
  --title "Fund marketing campaign" \
  --description "Allocate 10,000 tokens for Q2 marketing" \
  --recipient <address> \
  --amount 100000000000000 \
  --from <proposer_address>

# Vote
vexidus dao vote --dao <dao_address> --id 0 --approve --from
<voter_address>

# Execute an approved proposal
vexidus dao execute --dao <dao_address> --id 0 --from <executor_address>

# View DAO details
vexidus dao show --dao <dao_address>

# List proposals
vexidus dao proposals --dao <dao_address> --status Voting

# Check voting power
vexidus dao voting-power --dao <dao_address> --holder <address>
```

DaimyoDAO Web Interface

daimyodao.com provides a full web interface for VSC-99 DAOs:

- **Browse DAOs** -- Search and filter all on-chain DAOs

- **Create DAO** -- Multi-section form (identity, token, voting rules, council, treasury)
- **DAO Detail** -- 4-tab view (Overview, Proposals, Governance, My Power)
- **Propose & Vote** -- Submit proposals, cast votes, execute approved proposals
- **Council Actions** -- Veto, fast-track, member management (for council members)

No backend required -- all data is read from and written to the Vextaris blockchain via VSC-99 RPC.

Proposal Lifecycle

```
Created (Voting)
    |
    +-- Token holders vote (balance-weighted)
    |       |
    |       +-- Quorum met + threshold reached --> Approved
    |       |       |
    |       |       +-- Execution delay passes
    |       |       |       |
    |       |       |       +-- DAOExecute called --> Executed
    |       |       |
    |       |       +-- (waiting for timelock)
    |       |
    |       +-- Quorum met + below threshold --> Rejected
    |
    +-- Voting period expires without quorum --> Expired
    |
    +-- Proposer cancels --> Cancelled
    |
    +-- Council veto --> Vetoed (if council enabled)
    |
    +-- Council fast-track --> Approved immediately
```

Security Design

Concern	Protection
Vote buying	Snapshot at proposal creation block -- buying tokens after doesn't help
Flash loan attacks	Same-block manipulation impossible with creation-time snapshot
Council abuse	All council actions logged on-chain, visible to community
Treasury drain	Per-proposal spending cap + execution delay for community reaction time
DAO takeover	Config changes go through full voting process
DAO spam	Max 10 DAOs per creator, creation fee (100 VXS on mainnet)

Competitive Advantage

Feature	Ethereum DAOs	Solana DAOs	Vextaris VSC-99
Deployment cost	$50 -- 500 (gas)	$2 -- 5	~$0.0002
Audit needed	Yes (contract bugs)	Yes	No (protocol-enforced)
Elder council	Custom contract	SPL Governance	Built-in, configurable
Voting snapshots	Off-chain (Snapshot.org)	On-chain	On-chain, protocol-native
Cost per vote	$5 -- 50	$0.01 -- 0.05	~$0.0002
DAO creation UX	Complex (deploy + verify)	Moderate (SPL CLI)	One CLI command or web form

VexBridge -- Cross-Chain Verification

> ⚠ **TESTNET: INBOUND DEPOSITS ARE PAUSED**
>
> Since **2026-10-31**, inbound VexBridge deposits are **paused** on testnet while attestation moves to the attester quorum used by Atomence. Deposit requests are not processed during the pause. Bridge read methods (status, history, fees, safety registry) remain available.

VexBridge is the part of Vextaris that **proves facts about other chains** -- "this transaction happened on chain X, paid this address, and is final". On mainnet, that proof is used to move value through escrowed swaps, not to mint copies of foreign tokens.

The Mainnet Rule: Nothing Comes to Vextaris That Can't Leave

> ⓘ **PLANNED FOR MAINNET**
>
> This section describes the adopted policy for mainnet. It is a plan, not a description of the current testnet, and no launch dates are implied.

Every token on Vextaris is either:

1. **Native** -- VXS and any token created on Vextaris (VSC-7, VSC-21, VSC-55, ...), or
2. **Issued by a named party obliged to redeem it 1:1** -- a chartered stablecoin issuer, and later possibly a chartered custodian.

Anything else is not brought over as a token. A synthetic copy of a foreign asset that nobody is obliged to redeem is only worth what someone will pay for it; if the bridge or its market disappears, holders are stuck. Vextaris does not create such tokens at mainnet.

What this means for each kind of asset

Asset	Example	How it moves to/from Vextaris (planned)
Native	VXS, tokens created on Vextaris	Lives on Vextaris
Chartered stablecoin	An issuer-minted stablecoin under a VSC-8 charter	The issuer mints and redeems; cross-chain movement via Atomence swaps until the issuer runs its own mint/redeem on Vextaris
Major foreign assets	BTC, ETH, SOL, USDC on other chains	**Not held as wrapped tokens.** Users **swap** into a native asset (e.g. VXS or a chartered stablecoin) through Atomence, and swap back the same way -- with escrow and refund guarantees on every leg
Chartered wrapped asset (later, optional)	A custodian-backed BTC	Only if a regulated custodian commits to reserves and 1:1 redemption under a charter
Long-tail foreign tokens	Other ERC-20 / SPL tokens	**Not supported at mainnet**

Stablecoins come only from chartered issuers. No stablecoin is minted by the bridge.

What VexBridge becomes

At mainnet VexBridge is a **verification layer**, not a minting layer:

- A **quorum of attesters** on separate infrastructure (testnet: operated by the Vexidus team; independent outside operators planned before mainnet) checks events on foreign chains and signs only **final** ones (Ethereum finalized checkpoint, Solana finalized commitment, Bitcoin at least 6 confirmations).
- The attester set is changed only by **on-chain governance**.
- **Atomence** uses those attestations to settle escrowed swaps in both directions:
  - **Outbound** (Vextaris asset → foreign asset): the user escrows on Vextaris, a bonded market maker pays them on the foreign chain, attesters confirm, the escrow releases. No proof by the deadline → the user is refunded plus compensation from the market maker's bond. *(Live under the new Atomence rules from block 30,655,525.)*
  - **Inbound** (foreign asset → Vextaris asset, planned): the market maker locks the Vextaris-side asset in escrow **first**; the user pays on the foreign chain only after seeing it locked; attesters confirm and the escrow releases to the user. No proof by the deadline → the market maker is refunded.
- Chartered issuers can use the same verification for their own mint/redeem flows.

On testnet today, five attesters run on five separate machines. Independent outside operators are planned before mainnet. See Atomence for the quorum and finality rules.

Testnet Bridge Design (Deposits Paused)

The testnet VexBridge was built to bring foreign tokens onto Vextaris by verifying that they were **burned** on the source chain (for tokens with a burn function) or sent to a **Vextaris-format address that cannot exist on the source chain** (for native coins such as SOL, ETH and BTC, which have no burn function), and then crediting an equivalent token on Vextaris.

This deposit path is **paused on testnet** and is **not the mainnet model**: tokens created this way have no party obliged to redeem them, and for native coins the source-chain coins are unrecoverable. At mainnet, bridge minting of foreign tokens is off; majors move through Atomence swaps instead (see above).

Bridgeability warnings

The testnet bridge classifies tokens before a user acts, and `vex_getExecutionPlan` returns the warning:

Level	Meaning
Full Round-Trip	Token can come in and go back out
Limbo-Locked	Native coin (SOL, ETH, BTC, ...): can come in; going back out requires a market maker to pay on the source chain
One-Way In	No established path back

Bridge Fee Model (testnet deposits)

Chain	Fee (bps)	Minimum Fee
Ethereum	30 (0.3%)	1 unit
Solana	10 (0.1%)	0.1 unit

Chain	Fee (bps)	Minimum Fee
Bitcoin	50 (0.5%)	5 units
Default	25 (0.25%)	0.5 unit

Fees are credited to the Foundation treasury.

RPC Endpoints

Endpoint	Purpose
`vex_getBridgeStatus`	Query bridge operation status
`vex_getBridgeHistory`	List bridge operations for an address
`vex_getBridgeFees`	Get fee config for a chain
`vex_getPendingReceives`	List pending auto-detected receives
`vex_getExecutionPlan`	Preview a bridge intent + bridgeability warnings
`vex_bridgeToken` / `vex_bridgeWithdraw` / `vex_autobridge`	Server-built deposit / withdrawal writers. **Refused:** every validator rejects unsigned transactions, and inbound deposits are paused on testnet

All write operations must be signed client-side (SDK, CLI `--key`, or VexSpark) and submitted with `vex_submitBundle`.

Contract Safety Registry (VSC-SAFE)

VexBridge includes an on-chain safety registry that automatically scores bridge contracts using heuristic analysis. This protects users from interacting with malicious or impersonating tokens being bridged into Vextaris.

How Scoring Works

Every bridged contract receives a heuristic safety score (0-100) based on five factors:

Factor	Effect	Description
Known whitelist	+30	Contract is on the verified list (USDC, USDT, WETH, LINK, DAI, UNI, WBTC on Ethereum; USDC/USDT SPL on Solana; USDC on Base/Arbitrum)
Impersonation	-40	Contract symbol matches a reserved stablecoin symbol (e.g., someone bridging a fake "USDC")
Launchpad origin	-20	Contract originated from pump.fun, Raydium launchpad, Moonshot, or similar platforms
Deposit history	+10/+20	5+ deposits: +10, 50+ deposits: +20 (established usage pattern)
Source chain tier	+10	Originated from a Tier 2 chain (Ethereum, Solana, Base, Arbitrum, Polygon, Avalanche, Optimism)

Base score starts at 50. Final score is clamped to 0-100.

Safety Levels

Level	Score Range	Behavior
Verified	80-100	Full bridge access, "verified_bridge" reputation badge
Standard	50-79	Normal bridge access
Warning	25-49	Bridge allowed with user warning, reduced reputation
Blocked	0-24	Bridge deposits rejected

Governance Reclassification

Validators can propose to reclassify a contract's safety level via VSC-88 governance:

- **ContractSafety proposal** -- Reclassify a bridge contract (e.g., upgrade a legitimate project from Standard to Verified, or block a scam contract)
- **CollectionSafety proposal** -- Flag or freeze a bridged NFT collection

```
# Reclassify a contract to Verified
vexidus governance create-proposal \
  --type contract-safety \
  --title "Verify USDC on Base" \
  --description "Circle's official USDC deployment on Base" \
  --safety-chain base \
  --safety-contract 0x833589fcd6edb6e08f4c7c32d4f71b54bda02913 \
  --safety-level Verified

# Block a scam token
vexidus governance create-proposal \
  --type contract-safety \
  --title "Block fake USDC on unknown chain" \
  --description "Impersonating USDC symbol, no liquidity" \
  --safety-chain unknown \
  --safety-contract 0xdeadbeef... \
  --safety-level Blocked
```

RPC Endpoints

Method	Description
`vex_getContractSafety`	Get safety record for a specific contract. Params: `[chain, contract_address]`
`vex_listContractSafety`	List all safety records (optional chain filter). Params: `[chain?, limit]`

Reputation Integration

Contract safety levels feed into the token reputation system (VSC-REP):

- **Verified** contracts receive the `verified_bridge` reputation badge
- **Warning** contracts receive reduced reputation scores
- **Blocked** contracts cannot bridge at all

NFT Metadata Hardening (VSC-SAFE)

All NFT and token metadata submitted on-chain is sanitized to prevent XSS, injection, and phishing attacks.

URI Validation

All URI fields (`metadata_uri`, `image`, `base_uri`, `icon_uri`, `banner_uri`, `external_url`) are validated:

- **Allowed schemes:** `https://`, `ipfs://`, `ar://` (Arweave)
- **Blocked:** `data:` URIs (prevents inline payload attacks), `javascript:`, `http://`
- **Pattern scanning:** Rejects URIs containing `<script`, `onerror=`, `onclick=`, `<iframe`, `eval(`, `document.cookie`, and 17+ other dangerous patterns
- **Length limit:** 512 characters for URIs, 256 for external URLs

Text Validation

All text fields (`description`, `name`, attribute values) are sanitized:

- **Null byte rejection** -- prevents string truncation attacks
- **Dangerous pattern blocking** -- same 17+ patterns as URI validation
- **Length limits:** 2,048 chars for descriptions, 64 chars for attribute keys, 256 chars for attribute values
- **Attribute count limit:** Max 50 attributes per token

Why This Matters

NFT metadata is displayed in wallets, explorers, and marketplaces. Without sanitization, a malicious NFT could inject JavaScript into VexScan, VexSpark, or any frontend that renders token metadata. VSC-SAFE ensures all on-chain metadata is safe to render in any context.

Implementation Status

Live on Testnet

- Inbound deposits: **paused** (since 2026-10-31)

- Attester set: five attesters on five separate machines, registered by on-chain governance (used for Atomence settlement from block 30,655,525)
- Bridgeability classification and user warnings
- Bridged token registry
- Per-chain fee configuration
- Bridge pause (from block 30,655,525 also stops new Atomence orders and matches; in-flight orders still settle or refund)
- VSC-SAFE contract safety registry with heuristic auto-scoring
- VSC-SAFE NFT metadata hardening (URI/text sanitization)
- Governance reclassification (ContractSafety/CollectionSafety proposals)

Planned for Mainnet

- Bridge minting of foreign tokens **off**; VexBridge as the verification layer only
- Independent outside attester operators, bonded and rotated by governance
- Inbound escrowed swaps through Atomence (market maker locks first)
- Per-asset rate limits / daily caps
- Stablecoins only from chartered issuers
- Longer term: on-chain light clients per chain where feasible (Bitcoin first)

Industry Context

How Others Solve This Problem

Bridge	Model	Trust	Risk
Wormhole	Vault + guardian multisig	13-of-19 guardians	$320M hack (Feb 2022)
Ronin	Vault + validator multisig	5-of-9 validators	$625M hack (Mar 2022)
Chainlink CCIP	Oracle DON + burn/mint	31+ oracle nodes	Oracle collusion
LayerZero	Relayer + oracle	Configurable verifier set	Verifier compromise

Bridge	Model	Trust	Risk
Multichain	Custodial MPC + wrapped tokens	MPC signers	Bridged tokens on other chains de-pegged when it failed (2023)
Vextaris (mainnet plan)	**No vault, no wrapped copies. Escrowed swaps + attester quorum**	Attester quorum (e.g. 3-of-5) + bonded market makers	A failed or absent market maker → user refunded from escrow + bond

Why No Vault and No Wrapped Copies

Every major bridge hack has targeted the **vault** -- the contract holding locked tokens:

- Wormhole ($320M): Attacker minted tokens without depositing to vault
- Ronin ($625M): Compromised validator keys controlling the vault
- Nomad ($190M): Faulty Merkle root allowed draining the vault

And when a bridge fails, holders of its **wrapped copies** are left with tokens nobody is obliged to redeem.

The Vextaris mainnet model avoids both: there is no vault to drain, and no wrapped copy to strand. Users hold either native assets or assets a named issuer must redeem, and every cross-chain swap is backed by an on-chain escrow that ends in settlement or refund.

PATENT NOTICE

VexBridge is protected by U.S. Patent Application 63/987,929, assigned to Vexidus Corporation. See vextaris.com for details.

Vextaris Validator Setup Guide

Run a validator node on the Vextaris network and earn VXS rewards for securing the chain.

Table of Contents

1. Overview
2. System Requirements
3. Install Vextaris
4. Generate Your Validator Key
5. Bootstrap Your Node
6. Configure Your Node
7. Start Your Node (VexVisor)
8. Stake and Register as a Validator
9. Connect to the Network
10. Storage and Pruning
11. Monitoring
12. Automatic Upgrades (VexVisor)
13. Security Best Practices
14. Maintenance
15. Troubleshooting
16. Testnet Rewards

Overview

Vextaris validators produce blocks adaptively (2s nominal, 500ms-12s range) and earn rewards from two sources:

- **Block rewards** -- Time-weighted from a 10-year emission schedule (13M VXS/yr year 1-2). 85% to validators (density-weighted 0.5x-2.0x), 15% to Foundation Treasury
- **Transaction fees** -- 80% to block proposer, 20% to Foundation Treasury (no burn)

Parameter	Value
Minimum stake	1,000 VXS
Unbonding period	21 days
Max active validators	100
Block time	~2s adaptive (500ms-12s)
Epoch duration	1 hour (1,800 blocks)
Jailing	Double-sign detection → escalating jail (1h → 24h → 7d → 30d)
Slashing	**None** -- Vextaris uses jailing, not economic penalties

Validators use Ed25519 keys for block signing and vote participation in the HyperSync consensus protocol with Vexcel attestation DAG for global latency fairness. P2P transport uses QUIC (TLS 1.3 + multiplexing over UDP).

System Requirements

Minimum

Resource	Spec
CPU	4 cores / 4 threads
RAM	8 GB
Storage	100 GB SSD -- of which **~38 GB must be free at first start** for the snapshot restore (the archive plus the state it extracts to; the archive is deleted afterwards)
Network	100 Mbps, static IP or NAT with port forwarding
OS	Ubuntu 22.04+ / Debian 12+ (x86_64)

Recommended

Resource	Spec
CPU	8+ cores / 16 threads
RAM	16 GB+
Storage	500 GB NVMe SSD
Network	1 Gbps, static IP, low latency
OS	Ubuntu 24.04 LTS (x86_64)

Required Ports

Port	Protocol	Purpose
9946	UDP (QUIC)	P2P -- QUIC transport (TLS 1.3 + multiplexing). Must be open to all peers.
9933	TCP	RPC -- restrict to localhost or trusted IPs

Required Tools

These are installed automatically by the quick install script, or you can install them manually:

Tool	Purpose
`curl`	Downloads and RPC calls
`jq`	JSON formatting for RPC responses
`coreutils` (`sha256sum`)	Verifying the released binary against the published checksum

The node ships as a pre-built binary, so no compiler, Rust toolchain or build dependencies are needed. (This table previously listed `build-essential`, `cmake`, `libclang-dev` and `rustc` for a build-from-source path that is not available publicly during the beta programme.)

Install Vextaris

Quick Install (one command)

```
curl -sSf https://vexidus.io/releases/install-validator.sh | bash
```

This downloads the released binary (checksum-verified), the genesis files, VexVisor, and **the current network flag set**, creates the system user and directories, writes a systemd unit, and prints next steps. It does **not** generate keys or start the node -- those steps require your input.

The flag set it fetches (`validator-flags.txt`) is generated from the validators that are actually running, and your `ExecStart` is written from it. You set only the identity flags (`--data-dir`, `--rpc-port`, `--p2p-port`, `--validator-key`, `--external-addr`).

The installer **refuses to write a unit** if that flag set is missing, or if it was generated for a different binary than the one it just installed. Both refusals are deliberate: a node missing a consensus activation height either sits at height 0 forever or diverges from the network and stops. If you hit either refusal, report it -- do not hand-edit the flags.

> To inspect the script before running: `curl -sSf https://vexidus.io/releases/install-validator.sh | less`

Manual Install

1. Install system dependencies

```
sudo apt update && sudo apt install -y \
  build-essential \
  pkg-config \
  libssl-dev \
  libclang-dev \
  cmake \
  git \
  curl \
  jq
```

2. Download the released binary

The node is distributed as a pre-built binary -- there is no public build-from-source path during the beta programme, and you do not need a Rust toolchain.

```
curl -fSL -o vexidus-node https://vexidus.io/releases/vexidus-node
curl -fsSL -o SHA256SUMS https://vexidus.io/releases/SHA256SUMS
grep vexidus-node SHA256SUMS | sha256sum --check -    # must print:
vexidus-node: OK
```

> **VERIFY THE CHECKSUM**
>
> If the check does not print `OK`, stop. Do not run the binary -- report it to validators@vexidus.com. The quick install does this step for you.

3. Install the binary

```
chmod +x vexidus-node
sudo cp vexidus-node /usr/local/bin/
vexidus-node --help
```

4. Fetch the network flag set

Your node must run every consensus activation height the network is enforcing. That list changes as the network activates protocol upgrades, so it is **generated from the running validators**, not copied from a document:

```
curl -fsSL -o validator-flags.txt https://vexidus.io/releases/validator-
flags.txt
head -7 validator-flags.txt      # provenance: which binary and height it
was generated from
```

Every `--` line in that file belongs in your `ExecStart`, alongside your own identity flags (`--data-dir`, `--rpc-port`, `--p2p-port`, `--validator-key`, `--external-addr`). A node missing one of them either sits at height 0 forever or diverges from the network and stops.

> **LET THE INSTALLER DO THIS**
>
> The quick install fetches this file and writes the unit from it, and refuses to proceed if the flag set is missing or was generated for a different binary. Assembling the unit by hand is the step that goes wrong.

5. Create system user and directories

```
sudo useradd --system --no-create-home --shell /usr/sbin/nologin vexidus
sudo mkdir -p /opt/vexidus/data
sudo chown -R vexidus:vexidus /opt/vexidus
```

Generate Your Validator Key

Your validator key is an Ed25519 keypair used to sign blocks and votes. **If you lose your secret key, your staked VXS is unrecoverable.**

Generate the key

```
openssl rand -hex 32 > validator.key
chmod 600 validator.key
```

This creates a 64-character hex file containing your 32-byte Ed25519 secret key.

Move to secure location

```
sudo mv validator.key /opt/vexidus/validator.key
sudo chown vexidus:vexidus /opt/vexidus/validator.key
sudo chmod 600 /opt/vexidus/validator.key
```

View your public key

Your node logs the derived public key on startup when `--validator-key` is provided. You can preview it with a quick test run:

```
vexidus-node --validator-key /opt/vexidus/validator.key --data-dir
/tmp/vex-keytest 2>&1 | head -20
rm -rf /tmp/vex-keytest
```

Look for: `Validator public key: 0x...` -- this is your validator address.

Back up your key

Copy your `validator.key` to a secure offline location:

- Encrypted USB drive
- Password manager with file attachments
- Hardware security module (HSM) for production deployments

Never share your secret key, paste it in chat, or commit it to version control.

Bootstrap Your Node

New validators do **not** need genesis files and do not run anything by hand to bootstrap. The flag set from step 4 includes `--snapshot-url`, so the first time your node starts with an empty data directory it:

1. Downloads the published state snapshot (`.tar.zst`, a few minutes depending on state size)
2. Verifies the archive against its published checksum and restores the chain state
3. Jumps to the snapshot height, then syncs the remaining blocks from peers
4. Is ready for you to stake and register

FINDING SNAPSHOT URLS

The supported snapshot is published at `https://vexidus.io/releases/vexidus-snapshot-latest.tar.zst`, restaged regularly, with its height and checksum in `vexidus-snapshot-latest.info`.

DO NOT BOOTSTRAP FROM ANOTHER VALIDATOR'S PORT

Earlier versions of this page pointed at a running validator's in-node snapshot port. That endpoint is unauthenticated and is not part of the supported install path -- use the published URL above.

:::

THE SNAPSHOT IS THE ONLY WAY IN

Joining by replaying the chain from block 0 is not supported on the current network. Earlier versions of this page offered "P2P sync (no snapshot)" and "genesis start" options; neither produces a working validator. If the snapshot download fails, fix that -- do not remove `--snapshot-url`.

Serve Snapshots to Other Validators

If you want to help new validators bootstrap faster, enable snapshot serving:

```
vexidus-node ... --serve-snapshots
```

This creates periodic checkpoints and serves compressed `.tar.zst` archives over plain HTTP on your RPC port + 10 (e.g. RPC `:9933` -> snapshots on `:9943`).

KEEP THAT PORT PRIVATE

The snapshot endpoint has **no authentication and no rate limiting**, and serving a snapshot is expensive for the node doing it. Do not expose it to the internet: bind it behind your firewall and share it only with peers you choose. New validators should use the published snapshot URL above. Checkpoints also accumulate on disk -- watch free space, since a full disk stops your node.

Configure Your Node

The node is configured entirely by command-line flags in its systemd unit -- there is no config file. Those flags come in two parts:

- **Your identity** (you set these): `--data-dir`, `--rpc-port`, `--p2p-port`, `--validator-key`, `--external-addr`.
- **The network's flags** (you copy these, never edit them): every `--` line of the `validator-flags.txt` you fetched in step 4 -- the bootnodes, the snapshot URL and the consensus activation heights.

DO NOT WRITE THIS UNIT BY HAND

A unit with only the handful of flags you would type yourself starts, connects to peers, and then stays at height 0 forever with no error. A unit missing one activation height computes a different state from the rest of the network and stops. Build it from the flag set, as below.

Create the systemd service

Run this in the directory where you saved `validator-flags.txt`. It writes the same unit as the quick install:

```
MY_IP=$(curl -4 -s ifconfig.me)   # your server's public IPv4 -- check it
before continuing
echo "$MY_IP"

{
cat <<EOF
[Unit]
Description=Vextaris Validator Node
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=vexidus
Group=vexidus
WorkingDirectory=/opt/vexidus
ExecStart=/opt/vexidus/tools/vexvisor.sh \\
  --data-dir /opt/vexidus/data \\
  --rpc-port 9933 \\
  --p2p-port 9946 \\
  --validator-key /opt/vexidus/validator.key \\
  --external-addr /ip4/${MY_IP}/udp/9946/quic-v1 \\
EOF
grep '^--' validator-flags.txt | sed 's/^/  /; s/$/ \\/'
cat <<'EOF'
  --metrics
Restart=always
RestartSec=5
LimitNOFILE=1048576
Environment=RUST_LOG=info

# Hardening
ProtectSystem=full
ProtectHome=read-only
NoNewPrivileges=true

[Install]
WantedBy=multi-user.target
EOF
} | sudo tee /etc/systemd/system/vexidus-validator.service > /dev/null
```

Then check it. The second number must be the first plus 6 (your five identity flags and `--metrics`):

```
sudo systemd-analyze verify /etc/systemd/system/vexidus-validator.service
# no output = OK
grep -c '^--' validator-flags.txt
grep -c '^  --' /etc/systemd/system/vexidus-validator.service
```

The unit runs the node through `vexvisor.sh` (set up next) so network upgrades are applied automatically, and `--metrics` exposes Prometheus metrics.

> **WHEN THE NETWORK'S FLAGS CHANGE**
>
> Protocol upgrades add activation heights. When an upgrade announcement asks you to refresh your flags, fetch `validator-flags.txt` again and regenerate the unit the same way, keeping your identity flags. Network-wide rule changes -- for example the Atomence settlement rules activating at block 30,655,525 -- need nothing else by hand: the published flag set is republished when an activation is configured.

Start Your Node (VexVisor)

All production validators should run via the **VexVisor wrapper**. VexVisor monitors your node and handles automatic binary upgrades -- when a network upgrade is approved, VexVisor downloads the new binary, verifies its checksum, swaps it in, and restarts your node. No manual intervention required.

Set up VexVisor

Run this in the directory where you downloaded `SHA256SUMS` in step 2:

```
curl -fsSL -o vexvisor.sh https://vexidus.io/releases/vexvisor.sh
grep vexvisor.sh SHA256SUMS | sha256sum --check -    # must print:
vexvisor.sh: OK

sudo mkdir -p /opt/vexidus/tools /opt/vexidus/vexvisor/current
sudo install -m 755 vexvisor.sh /opt/vexidus/tools/vexvisor.sh
sudo cp /usr/local/bin/vexidus-node
/opt/vexidus/vexvisor/current/vexidus-node
sudo chown -R vexidus:vexidus /opt/vexidus
```

Keep the script at `/opt/vexidus/tools/vexvisor.sh`: VexVisor finds the node binary (`/opt/vexidus/vexvisor/current/`) and the data directory relative to the folder above its own.

The unit you created already starts the node through VexVisor, which passes all flags through to the binary and only intercepts exit code 42 (the upgrade signal) to swap in the new binary.

> **WHY VEXVISOR MATTERS**
>
> Without VexVisor, you must manually download, verify, and restart your node for every upgrade. Missing an upgrade can cause your node to halt at the upgrade height and stay down until you intervene -- potentially getting jailed for missed blocks. VexVisor handles this automatically.

Start and enable

```
sudo systemctl daemon-reload
sudo systemctl enable vexidus-validator
sudo systemctl start vexidus-validator
```

Verify it is running

```
sudo systemctl status vexidus-validator
```

Watch the logs

```
sudo journalctl -u vexidus-validator -f
```

You should see:

```
[VexVisor] Starting vexidus-node...
Vextaris Blockchain Node v0.1.0
Loading validator key from "/opt/vexidus/validator.key"
Validator public key: 0xabc123...
P2P Network started successfully
Listening on /ip4/0.0.0.0/udp/9946/quic-v1
RPC server listening on 0.0.0.0:9933
Block #12345 produced ...
```

Wait for sync

Before staking, ensure your node is fully synced. Check your current block height:

```
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}' |
jq .result
```

Compare with the network tip on vexscan.io. Once the heights match, your node is synced.

Stake and Register as a Validator

Once synced, stake VXS to join the active validator set.

> **STAKING AND VALIDATOR SETTINGS ARE SIGNED TRANSACTIONS**
>
> Validators refuse unsigned transactions (since 31 October 2026). Write methods that build a transaction on the node from an address you name -- `vex_stake`, `vex_setCommission`, `vex_setValidatorMetadata`, `vex_unjail` and similar -- are refused with `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`. Use the `vexidus` CLI with `--key <path>` (it signs locally and submits with `vex_submitBundle`), the SDK, or a wallet. `--key` takes the secret-key file (64-char hex Ed25519) of the account that holds the VXS. CLI amounts are in **raw units** (1 VXS = 1,000,000,000).

Get Testnet VXS

Before staking, you need VXS tokens. Visit the testnet faucet at vexswap.xyz:

1. Use a staking account whose secret key you hold -- this is the key you will pass to the CLI as `--key`
2. Go to https://vexswap.xyz and enter your address to receive testnet VXS
3. Rate limit: 100 VXS per address per hour -- request multiple times to accumulate the 1,000 VXS minimum stake

Check your balance

```
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getBalance","params":
["YOUR_ADDRESS","VXS"],"id":1}' | jq .
```

Stake VXS (minimum 1,000)

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake stake --from YOUR_STAKER_ADDRESS --amount 1000000000000 --pubkey
YOUR_VALIDATOR_PUBKEY_HEX
```

Parameter	Description
`YOUR_STAKER_ADDRESS`	The account holding your VXS -- the account whose key is in `staker.key`
`1000000000000`	Amount to stake in raw units (1,000 VXS, the minimum)
`YOUR_VALIDATOR_PUBKEY_HEX`	64-char hex public key derived from your `validator.key`

Verify your registration

```
# Check your validator entry
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getValidator","params":
["YOUR_ADDRESS"],"id":1}' | jq .

# List all active validators
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_listValidators","params":
[100],"id":1}' | jq .

# Network staking summary
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_stakingInfo","params":[],"id":1}' |
jq .
```

Set your validator profile

Set on-chain metadata so delegators can identify your validator:

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake set-metadata --from YOUR_ADDRESS --name "My Validator" \
  --description "Reliable validator node" --website https://mysite.com --
avatar-url https://mysite.com/avatar.png
```

Set commission rate

Commission is the percentage of delegator rewards you keep (in basis points, max 5000 = 50%):

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake set-commission --from YOUR_ADDRESS --rate 500
```

`500` = 5% commission. Default is 0%.

> **COMMISSION COOLDOWN**

> Commission changes have a **7-day cooldown** (~50,400 blocks). After changing your commission rate, you must wait 7 days before changing it again. This prevents commission manipulation (e.g., advertising low commission to attract delegators, then raising it).

Auto-Compound Rewards

Delegators can enable auto-compound to automatically restake their earned rewards instead of manually claiming:

```
# Enable auto-compound for your delegation (signed by the delegator's key)
vexidus --rpc http://localhost:9933 --key /path/to/delegator.key \
  stake set-auto-compound --validator YOUR_VALIDATOR_ADDRESS --enable --from YOUR_DELEGATOR_ADDRESS
```

When enabled, rewards are restaked at the end of each epoch -- no additional transactions or gas needed.

Staking Pool Configuration (Validators)

Validators can configure their staking pool to attract delegators:

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake set-pool-config --from YOUR_VALIDATOR_ADDRESS --pool-name "My Staking Pool" \
  --min-delegation 100000000000 --auto-compound-default
```

Parameter	Description
`--pool-name`	Display name for your staking pool
`--min-delegation`	Minimum VXS required to delegate (raw units, 9 decimals; default 0 = no minimum)
`--auto-compound-default`	If present, new delegators default to auto-compound

Pool settings are visible via `vex_getValidator` and `vex_listValidators`.

Gas and Transaction Fees

Vextaris uses a gas-based fee model similar to Ethereum, but with Solana-competitive pricing:

Parameter	Value
Base gas price	10 nanoVXS per gas unit (configurable via `--gas-price`)
Transfer gas	~21,000 gas units
Cost per transfer	~~0.00021 VXS~~ ($0.0002 at $1/VXS)
Fee recipient	80% to block proposer, 20% Foundation Treasury (no burn)

Set `--gas-price 0` in your systemd service for free testnet transactions.

Connect to the Network

Bootstrap Nodes

Your node needs at least one bootstrap peer to discover the network. Use all three for best redundancy:

NA (Beauharnois, Canada) -- note UDP port 9944:

```
/ip4/158.69.203.54/udp/9944/quic-v1/p2p/12D3KooWQH31LHN8xWj183oZSkDLA8FeNfCPsNJXww5xyQsPNozh
```

EU (Zurich, Switzerland):

```
/ip4/141.227.150.129/udp/9945/quic-v1/p2p/12D3KooWAf1EaQ9fa7UqtTK9x7w8jKh1BdcU7epfrwgyXskx2QTS
```

Asia (Singapore):

```
/ip4/15.235.210.254/udp/9945/quic-
v1/p2p/12D3KooWQjomQQ3gGobaK11TcjTWbrDESvTFY1SiN4JnZKzAMhhp
```

Pass all three as one comma-separated `--bootnodes` value. **Keep the `/p2p/<PeerId>` suffix:** without it your node cannot tell that it is already connected to a seed, redials it about once a minute, and each redial leaves an extra connection open for up to 24 hours, so memory grows until the node restarts.

A seed's PeerId changes if its data is ever reset, and a PeerId that no longer matches makes the dial fail. If your node logs dial errors to a seed, take the list again from the flag set below.

The current bootstrap peers ship in the published flag set, `validator-flags.txt` -- the `--bootnodes` line. It is generated from the running network, so it is the list to trust over any copy pasted elsewhere.

NAT and Firewalls

If your server is behind NAT:

1. **Forward port 9946 (UDP)** from your router to your server
2. **Set `--external-addr`** so peers can find you:

```
--external-addr /ip4/YOUR_PUBLIC_IP/udp/9946/quic-v1
```

Verify peer connectivity

```
sudo journalctl -u vexidus-validator --no-pager | grep -i
"peer\|connect\|dial"
```

Firewall setup (ufw)

```
# Allow P2P from anywhere (QUIC uses UDP)
sudo ufw allow 9946/udp

# Block public RPC access (use localhost only)
sudo ufw deny 9933/tcp

# Enable firewall
sudo ufw enable
```

If you need remote RPC access from specific IPs:

```
sudo ufw allow from YOUR_TRUSTED_IP to any port 9933 proto tcp
```

Storage and Pruning

Vextaris stores chain state in RocksDB, and every node currently keeps the **full history**: the database grows for as long as the node runs.

> ⚠ **PRUNING IS NOT ACTIVE IN THE CURRENT RELEASE**
>
> The node accepts a `--pruning` flag, but it has no effect at present -- setting it does not reduce disk usage. It is not part of the network flag set, so leave it out of your unit. Earlier versions of this page recommended `--pruning validator` and quoted fixed storage sizes for it; those sizes did not apply.

Size your disk from the System Requirements and watch free space: a full disk stops your node, and the symptom looks like a consensus problem rather than a disk problem.

```
df -h /opt/vexidus
sudo du -sh /opt/vexidus/data
```

Node Modes

Mode	Flag	Can Produce Blocks	Full State	Use Case
Full validator	*(default)*	Yes	Yes	Standard validator
Light node	`--light`	No	No (headers only)	Monitoring, RPC relay

Light mode (`--light`) runs a header-only node that cannot produce blocks or query account state. It is useful for monitoring network health or as a lightweight RPC relay.

Prometheus Metrics

Enable metrics export for Grafana, Prometheus, or other monitoring tools:

```
--metrics                    # Enable Prometheus metrics endpoint
--metrics-port 9100          # Port for metrics (default: 9100)
--metrics-external           # Allow external access (default: localhost only)
```

Metrics include block height, peer count, mempool size, block production latency, and more.

Monitoring

Quick health check

```
curl -s http://localhost:9933 \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}' |
jq .result
```

Logs

```
# Last 50 lines
sudo journalctl -u vexidus-validator -n 50 --no-pager

# Live follow
sudo journalctl -u vexidus-validator -f
```

Monitoring script

Save as `/opt/vexidus/monitor.sh`:

```
#!/bin/bash
# Vextaris Validator Health Monitor

RPC="http://localhost:9933"

rpc_call() {
  curl -sf "$RPC" \
    -H "Content-Type: application/json" \
    -d "{\"jsonrpc\":\"2.0\",\"method\":\"$1\",\"params\":$2,\"id\":1}"
2>/dev/null
}

block=$(rpc_call "eth_blockNumber" "[]" | jq -r '.result //
"unreachable"')
validators=$(rpc_call "vex_listValidators" "[100]" | jq -r '.result |
length // 0')
staking=$(rpc_call "vex_stakingInfo" "[]" | jq -r '.result.total_staked
// "0"')
status=$(systemctl is-active vexidus-validator)
uptime=$(systemctl show vexidus-validator --property=ActiveEnterTimestamp
--value)

echo "=============================="
echo "  Vextaris Validator Status"
echo "=============================="
echo "  Service:       $status"
echo "  Block height:  $block"
echo "  Validators:    $validators"
echo "  Total staked:  $staking VXS"
echo "  Up since:      $uptime"
echo "=============================="
```

Security Best Practices

Key Management

- Store `validator.key` with permissions `600`, owned by `vexidus` user only
- Maintain an encrypted offline backup (USB, password manager, or HSM)
- Never share your secret key or commit it to git
- Key rotation is available via `RotateConsensusKey` (disc 119) — queued and applied at next epoch boundary, max 1 per epoch

Network

- **RPC port (9933):** Never expose to the public internet. Bind to `127.0.0.1` or restrict via firewall.
- **P2P port (9946):** Must be open to all peers for consensus participation.

System Hardening

- Keep your OS updated: `sudo apt update && sudo apt upgrade -y`
- Enable unattended security updates: `sudo apt install -y unattended-upgrades`
- Use SSH key authentication and disable password login
- Run the node as the dedicated `vexidus` user, never as root
- The systemd service includes `ProtectSystem=full`, `ProtectHome=read-only`, and `NoNewPrivileges=true` by default

Automatic Upgrades (VexVisor)

Network upgrades are coordinated on-chain via VexVisor governance. When an upgrade is approved, **every validator running VexVisor upgrades automatically** -- no manual intervention needed.

How it works

1. The upgrade authority (or validator vote in `full` mode) schedules an upgrade at a future block height
2. All nodes see the approved upgrade plan on-chain
3. At the target height, your node writes `data/upgrade-info.json` and exits with code **42**
4. VexVisor detects exit(42) and:
   - Downloads the new binary from the URL in the upgrade plan
   - Verifies the **SHA256 checksum** (rejects tampered binaries)
   - Backs up your current binary
   - Swaps in the new binary and restarts
5. If the new binary crashes within 60 seconds, VexVisor **automatically rolls back** to the previous version

Grace period

After an upgrade, there is a **100-block grace period** (~20 minutes) where missed-block tracking is suppressed. You will not be jailed for blocks missed during the upgrade window.

What you need to do

Nothing. If you set up VexVisor as described in the Start Your Node section, upgrades are fully automatic. You can monitor the upgrade via logs:

```
sudo journalctl -u vexidus-validator -f | grep -i "upgrade\|vexvisor"
```

Governance modes

Mode	How upgrades are approved	Use case
`seed-authority`	A designated upgrade authority approves scheduled upgrades	Bootstrapping / private networks
`full`	Stake-weighted validator vote (67% quorum, configurable)	Production networks

For full details on the governance protocol, see VexVisor Upgrade Governance.

Chain-Data Integrity Tools

Three optional flags help you catch RocksDB corruption before consensus consumes it. All are off by default — opt in as you see fit.

```
# One-shot scan at startup. Walks every well-known structured-record
prefix
# and exits with a non-zero status if any entry fails to deserialise.
vexidus-node --verify-on-start

# Background sweep on a configurable interval. Reports anomalies via the
# node log without restarting. Recommended for validators: every 1 hour.
vexidus-node --rocksdb-verify-interval-secs 3600

# Dry-run a state.json snapshot into a temporary database. Live data dir
# is untouched. Useful before any real --import-state.
vexidus-node --dry-run-import ./state-export.json
```

A non-zero exit from `--verify-on-start` or `--dry-run-import` means a malformed record was found. Investigate before serving traffic — log lines list up to the first ten offending keys.

Maintenance

Manual upgrades

VexVisor applies scheduled upgrades on its own. If you must swap the binary by hand, verify it and replace the one your unit actually runs:

```
curl -fSL -o vexidus-node https://vexidus.io/releases/vexidus-node
curl -fsSL -o SHA256SUMS https://vexidus.io/releases/SHA256SUMS
grep vexidus-node SHA256SUMS | sha256sum --check -    # must print:
vexidus-node: OK

sudo systemctl stop vexidus-validator
sudo install -m 755 -o vexidus -g vexidus vexidus-node
/opt/vexidus/vexvisor/current/vexidus-node
sudo systemctl start vexidus-validator
sudo journalctl -u vexidus-validator -n 20 --no-pager
```

Copying the binary to `/usr/local/bin` has no effect -- the unit runs `/opt/vexidus/vexvisor/current/vexidus-node`. If the upgrade also changed the network flags, regenerate your unit as described in Configure Your Node.

> **WARNING**
>
> Without VexVisor you risk being **jailed** if you miss an upgrade window. Run it on production validators.

Unstaking

To leave the validator set, begin the 21-day unbonding process:

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake unstake --from YOUR_ADDRESS --amount 1000000000000
```

After 21 days, submit a `ClaimUnstake` transaction to withdraw your VXS.

Full resync (wipe chain data)

If your node is corrupted or you need a clean start:

```
sudo systemctl stop vexidus-validator
sudo rm -rf /opt/vexidus/data
sudo systemctl start vexidus-validator
```

Your `validator.key` is preserved -- only chain state is wiped. The node will re-sync from peers automatically. For faster recovery, use `--snapshot-url` to bootstrap from a snapshot instead of syncing block-by-block.

Troubleshooting

Node will not start

```
sudo systemctl status vexidus-validator
sudo journalctl -u vexidus-validator -n 100 --no-pager
```

Symptom	Fix
Permission denied on key file	`sudo chown vexidus:vexidus /opt/vexidus/validator.key && sudo chmod 600 /opt/vexidus/validator.key`
Port already in use	Check for conflicts: `sudo lsof -i :9946`
Missing data directory	`sudo mkdir -p /opt/vexidus/data && sudo chown vexidus:vexidus /opt/vexidus/data`
State not loading	Use `--snapshot-url` to bootstrap from an existing validator, or ensure P2P sync is working

No peers connecting

- Verify port 9946/udp is open: `nc -zuv YOUR_PUBLIC_IP 9946` from another machine
- Confirm `--external-addr` matches your actual public IP
- Check a bootnode is reachable: `nc -zuv 158.69.203.54 9944` (NA seed; Zurich and Singapore use 9945)
- Inspect dial attempts: `sudo journalctl -u vexidus-validator --no-pager | grep -i "dial\|error\|failed"`

Node not producing blocks

- Ensure your node is fully synced (block height matches the network)
- Verify you are in the active set: `vex_listValidators`
- Confirm your stake meets the 1,000 VXS minimum: `vex_getValidator`
- Check if your validator is jailed (see below)
- Check for epoch transitions in logs

Validator is jailed

Validators are jailed for double-sign violations (producing conflicting blocks at the same height). Jailed validators cannot produce blocks or earn rewards. Jail durations escalate: 1h → 24h → 7d → 30d. After the jail period expires, unjail yourself:

```
vexidus --rpc http://localhost:9933 --key /path/to/staker.key \
  stake unjail --from YOUR_ADDRESS
```

Check jail status with `vex_getValidator` -- look for `is_jailed` and `jail_release_time` fields.

RPC not responding

- Confirm the service is running: `systemctl is-active vexidus-validator`
- Check port is bound: `ss -tlnp | grep 9933`
- Test locally: `curl http://localhost:9933 -d '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}'`

Key Files Reference

File	Purpose
`/opt/vexidus/validator.key`	Ed25519 secret key (64 hex chars). **Keep secret.**
`/opt/vexidus/tools/vexvisor.sh`	VexVisor upgrade wrapper script
`/etc/systemd/system/vexidus-validator.service`	Node configuration (all flags)
`/opt/vexidus/vexvisor/current/vexidus-node`	Active node binary (managed by VexVisor)
`/opt/vexidus/data/node_key`	libp2p identity key (auto-generated, stable PeerId)
`/opt/vexidus/data/`	RocksDB chain state
`/opt/vexidus/data/upgrade-info.json`	Written by node at upgrade height (consumed by VexVisor)
`/opt/vexidus/data/checkpoints/`	Periodic state snapshots (if `--serve-snapshots` enabled)
`/usr/local/bin/vexidus-node`	Node binary (or use VexVisor path instead)

CLI Reference

```
vexidus-node [OPTIONS]

Core:
  -d, --data-dir <PATH>              Data directory [default: ./data]
  -v, --verbose                      Enable verbose logging
      --rpc-port <PORT>              RPC listen port [default: 9933]
      --p2p-port <PORT>              P2P listen port, QUIC/UDP [default:
9944]
      --bootnodes <MULTIADDRS>       Comma-separated bootstrap peer
multiaddrs
      --validator-key <PATH>         Path to Ed25519 signing key file
      --external-addr <MULTIADDR>    Public address to advertise (NAT
traversal)

Consensus:
      --gas-price <N>                Base gas price in nanoVXS/gas
[default: 10]
      --block-time <SECS>            Block production interval [default:
12]
      --max-txs-per-block <N>        Max transactions per block [default:
10000]

Bootstrap:
      --snapshot-url <URL>           Download and restore state snapshot
on first start
      --serve-snapshots              Serve snapshots to other nodes (port
= RPC + 10)

Storage:
      --pruning <MODE>               Accepted but currently has no effect
(see Storage and Pruning)
      --light                        Run as header-only light node (no
state execution)
      --elastic-state                Enable ElasticState 3-tier storage
(hot/warm/cold)
      --backfill-indexes             Backfill explorer indexes on startup

Metrics:
      --metrics                      Enable Prometheus metrics endpoint
      --metrics-port <PORT>          Metrics listen port [default: 9100]
      --metrics-external             Allow external access to metrics
endpoint

Governance (VexVisor):
      --governance-mode <MODE>       Governance mode: seed-authority |
```

```
full [default: seed-authority]
      --upgrade-authority <HEX>      Hex pubkey of the upgrade authority
      --auto-vote-seed <HEX>         Auto-vote seed for deterministic
voting
      --voting-period <BLOCKS>       Upgrade voting period in blocks
[default: 500]
      --upgrade-quorum <PERCENT>     Required approval quorum for
upgrades [default: 67]

Security:
      --enforce-validator-allowlist  Only allow authenticated validator
peers to gossip blocks
      --rpc-rate-limit <N>           Per-IP RPC rate limit in
requests/second [default: 100, 0 = unlimited]

  -h, --help                         Print help
```

Testnet Rewards

Testnet validators who maintain high performance earn **mainnet VXS airdrops** at launch. Rewards are based on your on-chain reputation score (VSC-REP), which tracks 7 factors:

- Uptime and block production consistency
- Vote participation rate
- Stake amount and duration
- Commission rate fairness
- Transaction processing volume
- Network contribution (peer connectivity)
- Governance participation (future)

Check your validator's reputation score via `vex_getValidator` -- the `performance_score` field (0.5 to 1.0) directly affects your leader selection weight and reward share. New validators start with a neutral reputation score of 50 (on the 0-100 scale).

Links

- **SDK & tooling:** github.com/vexidus-labs/vexidus-sdk (the core node repository is private during the beta programme)
- **Explorer:** vexscan.io
- **Faucet:** vexswap.xyz (testnet VXS for staking)
- **Developer Studio:** vexforge.xyz

Vextaris is its own L1 blockchain. Native addresses use the `Vx` prefix. The `0x` format exists only for EVM compatibility and internal system addresses.

Validator Server Hardening Guide

Production validators are high-value targets. A compromised validator can be jailed, have its stake at risk, or be used to attack the network. This guide covers the security hardening every Vextaris validator should apply before going live.

Quick Hardening Checklist

Step	Priority	Time
Firewall (UFW)	Critical	5 min
SSH Hardening	Critical	10 min
Fail2ban	Critical	5 min
Dedicated User	Critical	5 min
Automatic Updates	High	2 min
RPC Lockdown	High	5 min
Key Permissions	High	2 min
Systemd Hardening	Medium	5 min
Log Monitoring	Medium	10 min
Backup Strategy	Medium	15 min

Firewall (UFW)

Every validator must run a firewall. The principle: **deny everything, allow only what's needed.**

Install and configure

```
sudo apt update && sudo apt install -y ufw

# Default policy: deny all incoming, allow all outgoing
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Allow SSH (required for remote management)
sudo ufw allow 22/tcp

# Allow P2P (required for consensus — QUIC uses UDP)
# Use your actual P2P port (default 9944)
sudo ufw allow 9944/udp

# Allow HTTP/HTTPS only if you're running a web server (explorer, docs,
etc.)
# Most validators do NOT need these:
# sudo ufw allow 80/tcp
# sudo ufw allow 443/tcp

# Enable the firewall
sudo ufw enable
sudo ufw status verbose
```

What NOT to expose

Port	Service	Expose?
9944/udp	P2P (QUIC)	**Yes** — required for consensus
9933/tcp	RPC	**No** — localhost only (see RPC Lockdown)
9100/tcp	Prometheus metrics	**No** — localhost only
22/tcp	SSH	**Yes** — but harden it (see below)

Verify

```
sudo ufw status numbered
```

You should see only SSH and your P2P port allowed.

SSH Hardening

SSH is the #1 attack vector for internet-facing servers. These settings eliminate the most common attacks.

Prerequisites

Before changing SSH config, **ensure you have key-based auth working**:

```
# On your LOCAL machine, generate a key if you don't have one
ssh-keygen -t ed25519 -C "validator-admin"

# Copy your public key to the server
ssh-copy-id -i ~/.ssh/id_ed25519.pub your_user@your_server_ip

# Test key-based login (should not ask for password)
ssh -i ~/.ssh/id_ed25519 your_user@your_server_ip
```

> **DANGER**
>
> Do NOT disable password auth until you've confirmed key-based auth works. You will lock yourself out.

Apply hardening

Edit `/etc/ssh/sshd_config`:

```
sudo nano /etc/ssh/sshd_config
```

Set these values (uncomment if needed):

```
# Disable password authentication (key-only)
PasswordAuthentication no
ChallengeResponseAuthentication no

# Disable root login (use a regular user + sudo)
PermitRootLogin no

# Limit authentication attempts
MaxAuthTries 3
MaxSessions 3

# Disable unused features
X11Forwarding no
AllowAgentForwarding no
AllowTcpForwarding no

# Use only protocol 2
Protocol 2

# Idle timeout (disconnect after 10 minutes of inactivity)
ClientAliveInterval 300
ClientAliveCountMax 2
```

If your server only has a `root` user (no separate admin user), set `PermitRootLogin prohibit-password` instead of `no` — this allows key-based root login while still blocking password auth.

Check for cloud-init override

Some cloud providers (OVH, Hetzner) override SSH settings via cloud-init. Check:

```
ls /etc/ssh/sshd_config.d/
cat /etc/ssh/sshd_config.d/*.conf 2>/dev/null
```

If you see `PasswordAuthentication yes` in any file there, change it to `no`:

```
sudo sed -i 's/PasswordAuthentication yes/PasswordAuthentication no/'
/etc/ssh/sshd_config.d/*.conf
```

Restart SSH

```
# Keep your current session open! Test in a NEW terminal first.
sudo systemctl restart sshd

# In a NEW terminal, verify you can still log in:
ssh -i ~/.ssh/id_ed25519 your_user@your_server_ip
```

Fail2ban

Fail2ban automatically bans IPs that show malicious behavior (brute-force SSH attempts, etc.).

Install and configure

```
sudo apt install -y fail2ban
```

Create `/etc/fail2ban/jail.local`:

```
[DEFAULT]
bantime = 86400
findtime = 600
maxretry = 3
backend = systemd

[sshd]
enabled = true
port = ssh
filter = sshd
logpath = /var/log/auth.log
maxretry = 3
bantime = 86400
```

This bans any IP that fails SSH login 3 times within 10 minutes, for 24 hours.

Start and enable

```
sudo systemctl enable fail2ban
sudo systemctl start fail2ban

# Verify
sudo fail2ban-client status sshd
```

Check banned IPs

```
sudo fail2ban-client status sshd
```

Unban an IP (if you accidentally lock yourself out from another machine)

```
sudo fail2ban-client set sshd unbanip YOUR_IP
```

Dedicated System User

Never run your validator as `root`. Create a dedicated system user with no login shell:

```
sudo useradd --system --no-create-home --shell /usr/sbin/nologin vexidus
sudo mkdir -p /opt/vexidus/data
sudo chown -R vexidus:vexidus /opt/vexidus
```

Your systemd service should specify `User=vexidus` and `Group=vexidus`.

Automatic Security Updates

Enable unattended security patches so critical vulnerabilities are patched automatically:

```
sudo apt install -y unattended-upgrades
sudo dpkg-reconfigure -plow unattended-upgrades
```

Verify it's active:

```
cat /etc/apt/apt.conf.d/20auto-upgrades
```

You should see:

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

RPC Lockdown

Your RPC port (default 9933) gives full read access to chain state and can submit transactions. **Never expose it to the public internet.**

Option A: Localhost only (recommended)

The firewall config above already blocks port 9933. If you need to query RPC remotely, use an SSH tunnel:

```
# From your local machine
ssh -L 9933:localhost:9933 your_user@your_server_ip

# Now query via localhost on your machine
curl http://localhost:9933 -d
'{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}'
```

Option B: Nginx reverse proxy with rate limiting

If you need to expose RPC publicly (e.g., for an explorer), put it behind nginx with rate limiting:

```
# /etc/nginx/sites-available/vexidus-rpc
upstream vexidus_rpc {
    server 127.0.0.1:9933;
}

limit_req_zone $binary_remote_addr zone=rpc:10m rate=30r/s;

server {
    listen 443 ssl;
    server_name rpc.yourvalidator.com;

    ssl_certificate
/etc/letsencrypt/live/rpc.yourvalidator.com/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/rpc.yourvalidator.com/privkey.pem;

    location / {
        limit_req zone=rpc burst=60 nodelay;
        proxy_pass http://vexidus_rpc;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

This limits each IP to 30 requests/second with a burst of 60. Adjust based on your expected traffic.

Key File Permissions

Your validator key is the most sensitive file on the server. Lock it down:

```
# Owned by the vexidus user, readable only by owner
sudo chown vexidus:vexidus /opt/vexidus/validator.key
sudo chmod 600 /opt/vexidus/validator.key

# Verify
ls -la /opt/vexidus/validator.key
# Should show: -rw------- 1 vexidus vexidus
```

Also secure any `.env` files or configuration files that contain sensitive data:

```
sudo chmod 600 /opt/vexidus/validator.toml
```

Systemd Hardening

Add security directives to your systemd service file (`/etc/systemd/system/vexidus-validator.service`):

```
[Service]
# ... your existing ExecStart, User, etc. ...

# Security hardening
ProtectSystem=full
ProtectHome=read-only
NoNewPrivileges=true
PrivateTmp=true
ProtectKernelTunables=true
ProtectControlGroups=true
RestrictSUIDSGID=true

# File descriptor limit (RocksDB needs many open files)
LimitNOFILE=65536
```

After editing:

```
sudo systemctl daemon-reload
sudo systemctl restart vexidus-validator
```

Log Monitoring

Watch for suspicious activity

```
# Failed SSH attempts
sudo grep "Failed password" /var/log/auth.log | tail -20

# Successful logins
sudo grep "Accepted" /var/log/auth.log | tail -20

# fail2ban actions
sudo grep "Ban" /var/log/fail2ban.log | tail -20
```

Simple alerting script

Save as `/opt/vexidus/alert-check.sh`:

```
#!/bin/bash
# Run via cron every 5 minutes: */5 * * * * /opt/vexidus/alert-check.sh

RPC="http://localhost:9933"
THRESHOLD=60  # Alert if no new block in 60 seconds

# Check if node is running
if ! systemctl is-active --quiet vexidus-validator; then
    echo "ALERT: vexidus-validator is DOWN" | logger -t vexidus-alert
    exit 1
fi

# Check block production
BLOCK=$(curl -sf "$RPC" \
    -H "Content-Type: application/json" \
    -d '{"jsonrpc":"2.0","method":"vex_getNetworkStats","params":
[],"id":1}' \
    | jq -r '.result.blockHeight // 0')

if [ "$BLOCK" = "0" ]; then
    echo "ALERT: RPC unreachable or returning 0 blocks" | logger -t
vexidus-alert
fi

# Check disk usage
DISK_PCT=$(df /opt/vexidus/data | tail -1 | awk '{print $5}' | tr -d '%')
if [ "$DISK_PCT" -gt 85 ]; then
    echo "ALERT: Disk usage at ${DISK_PCT}%" | logger -t vexidus-alert
fi
```

```
chmod +x /opt/vexidus/alert-check.sh
# Add to cron
(crontab -l 2>/dev/null; echo "*/5 * * * * /opt/vexidus/alert-check.sh")
| crontab -
```

For production alerting, integrate with services like PagerDuty, Grafana alerts, or a simple webhook to Slack/Discord.

Backup Strategy

What to back up

Item	Location	Frequency	Critical?
Validator key	`/opt/vexidus/validator.key`	Once (store offline)	**YES** — loss = unrecoverable stake
Node identity	`/opt/vexidus/data/node_key`	Once	Preserves PeerId across restarts
Chain state	`/opt/vexidus/data/`	Weekly	Can re-sync from peers if lost
Config files	`/opt/vexidus/validator.toml`, systemd service	On change	Easy to recreate

Automated backup script

```
#!/bin/bash
# /opt/vexidus/backup.sh — run weekly via cron
BACKUP_DIR="/opt/vexidus/backups"
DATE=$(date +%Y%m%d)

mkdir -p "$BACKUP_DIR"

# Back up config and key (encrypted)
tar czf "$BACKUP_DIR/config-$DATE.tar.gz" \
    /opt/vexidus/validator.key \
    /opt/vexidus/validator.toml \
    /etc/systemd/system/vexidus-validator.service

# Keep only last 4 backups
ls -t "$BACKUP_DIR"/config-*.tar.gz | tail -n +5 | xargs rm -f
2>/dev/null

echo "Backup complete: $BACKUP_DIR/config-$DATE.tar.gz"
```

OFFSITE BACKUPS

Store an encrypted copy of your validator key in a separate location (encrypted USB, password manager, or cloud storage with client-side encryption). Never store keys in plaintext on cloud services.

Advanced Hardening (Optional)

Restrict SSH to specific IPs

If you always connect from the same IP or VPN:

```
sudo ufw delete allow 22/tcp
sudo ufw allow from YOUR_HOME_IP to any port 22 proto tcp
```

Disable IPv6 (if not needed)

```
echo "net.ipv6.conf.all.disable_ipv6 = 1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

Install and run Lynis (security audit)

```
sudo apt install -y lynis
sudo lynis audit system
```

Review the report and address any warnings relevant to your setup.

Rate limit SSH connections at the firewall level

```
# Limit SSH to 6 connection attempts per 30 seconds per IP
sudo ufw limit ssh
```

This is in addition to fail2ban and provides defense-in-depth.

Verification Checklist

After hardening, verify everything is correct:

```
# Firewall is active and configured
sudo ufw status verbose

# SSH password auth is disabled
sudo sshd -T | grep passwordauthentication
# Should output: passwordauthentication no

# Fail2ban is running
sudo fail2ban-client status sshd

# Node is running as dedicated user
ps aux | grep vexidus-node | grep -v grep
# Should show 'vexidus' user, NOT root

# Key file has correct permissions
ls -la /opt/vexidus/validator.key
# Should show: -rw------- 1 vexidus vexidus

# RPC is not externally accessible
# From another machine:
curl --connect-timeout 5 http://YOUR_SERVER_IP:9933
# Should timeout or connection refused

# P2P port is open
# From another machine:
nc -zuv YOUR_SERVER_IP 9944
# Should show connection succeeded
```

Summary

Layer	What	Why
Firewall	UFW deny-all + whitelist P2P/SSH	Blocks all unnecessary ports
SSH	Key-only, no root, rate limited	Eliminates brute-force attacks
Fail2ban	Auto-ban after 3 failures	Blocks persistent attackers
User isolation	Dedicated `vexidus` user	Limits blast radius of compromise

Layer	What	Why
RPC lockdown	Localhost only	Prevents unauthorized chain queries/transactions
Auto-updates	Unattended security patches	Closes known OS vulnerabilities
Monitoring	Log checks + alerting	Early detection of issues
Backups	Key + config, offsite	Recovery from data loss

These measures follow the principle of defense-in-depth — no single layer is sufficient alone, but together they make your validator significantly harder to compromise.

Questions? Join the Vextaris validator community, email validators@vexidus.com, or open an issue at github.com/vexidus-labs/vexidus-sdk.

Validator Reputation System

Vextaris uses a deterministic, on-chain reputation scoring system for validators. Reputation affects leader selection weight, reward multipliers, and delegator visibility.

Overview

- **7 scoring factors**, 100 points maximum
- **Computed on-chain** during epoch transitions -- deterministic, verifiable
- **Starting score:** 50 (neutral, "unproven")
- **Growth rate:** Reaches 80+ in ~7 days, 90+ in ~30 days if doing everything right

Scoring Factors

Factor	Max Points	What It Measures
Uptime	25	Block production consistency. Measured as ratio of expected vs actual blocks proposed.
Commission Fairness	15	Reasonable commission rates and stability. Frequent changes penalized.
Self-Stake	15	Skin in the game. Higher self-stake relative to delegations scores better.
Governance Participation	15	Voting activity on upgrade proposals (VexVisor). Active voters score higher.
Tenure	10	Time active on the network. Longer tenure = more points (capped).
Delegator Trust	10	Total stake attracted from delegators. More delegators = higher trust signal.
Reliability	10	Historical jail count and total missed blocks. Fewer jails and misses = better.

Grades

Grade	Score Range	Description
A+	95-100	Exemplary validator -- top tier
A	90-94	Excellent -- consistent, reliable
B	75-89	Good -- minor issues
C	55-74	Average -- room for improvement
D	35-54	Below average -- at risk
F	0-34	Failing -- likely jailed or inactive

How Reputation Affects Validators

Leader Selection Weight

Reputation directly multiplies the validator's leader selection weight:

```
effective_weight = stake * performance_score
```

Where `performance_score` ranges from 0.5 to 1.0. A validator with 5,000 VXS staked and a performance score of 0.9 has an effective weight of 4,500, while a validator with 5,000 VXS and 0.5 score has only 2,500. This means higher-reputation validators propose more blocks and earn more rewards.

Future: Reward Multipliers

Planned for post-beta:

Grade	Reward Multiplier
F	25% of base reward
D	60% of base reward
C	70% of base reward

Grade	Reward Multiplier
B	100% of base reward
A/A+	100% of base reward

No Slashing

Vextaris deliberately avoids slashing. Validators who miss blocks are jailed (temporarily removed from rotation), not economically punished. This:

- Protects delegators from losing funds
- Removes "slashing insurance" overhead
- Still incentivizes uptime via opportunity cost (missed rewards) and reputation impact

Anti-Manipulation

- **Commission change penalty:** Frequent commission changes reduce the Commission Fairness score
- **Jail penalty:** Each jail event permanently reduces the Reliability score
- **Minimum tenure:** New validators start at 50, preventing instant high-reputation gaming

Querying Reputation

Via RPC

```
# Get validator details including reputation
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getValidator","params":
["VALIDATOR_ADDRESS"],"id":1}' | jq .
```

Look for the `performance_score` field (0.5 to 1.0) and `reputation` field in the response.

Via CLI

```
vexidus validator info --address VALIDATOR_ADDRESS
```

Integration with IntentVM

IntentVM supports smart delegation via reputation:

```
"stake with best validator"
```

This intent resolves to the validator with the highest reputation score, enabling automatic optimal delegation without users needing to research validators manually.

VexVisor -- Upgrade Governance Module

VexVisor is the on-chain upgrade governance module (discriminants 46-48) that enables coordinated binary upgrades across the validator set without manual intervention.

How It Works

1. **Schedule** -- Authority proposes upgrade: name, target height, binary URL, SHA256 checksum
2. **Vote** -- Validators vote to approve/reject (stake-weighted, 67% quorum in `full` mode)
3. **Halt** -- At target height, node writes `data/upgrade-info.json` and exits with code 42
4. **VexVisor wrapper** -- the VexVisor wrapper detects exit(42), downloads new binary, verifies SHA256, restarts. Auto-rollback if crash within 60s.

Governance Modes

Mode	Behavior	Use Case
`seed-authority`	A designated upgrade authority approves scheduled upgrades	Bootstrapping / private networks
`full`	Stake-weighted validator voting with configurable quorum (default 67%)	Production networks

Configure with `--governance-mode <seed-authority|full>`.

Operations

Schedule, vote and cancel are signed transactions: pass the signing key with `--key <path>`. Validators refuse unsigned transactions (since 31 October 2026).

Schedule an Upgrade

Proposes a new upgrade to the network. The `--from` flag must use the upgrade authority's **pubkey** (not the Blake3 address).

```
vexidus upgrade schedule \
  --rpc http://localhost:9933 \
  --key /path/to/upgrade-authority.key \
  --from 0x<YOUR_UPGRADE_AUTHORITY_PUBKEY> \
  --name "lthash-audit-fix" \
  --height 150400 \
  --binary-url "https://vextaris.io/releases/vexidus-node" \
  --checksum "<RELEASE_SHA256>"
```

Requirements:

- Target height must be at least 100 blocks in the future
- Only one active upgrade plan at a time
- In `seed-authority` mode, must be signed by the upgrade authority
- Gas: 80,000

Vote on an Upgrade

Validators vote to approve or reject a proposed upgrade (only in `full` mode).

```
vexidus --key /path/to/validator-account.key upgrade vote --from
<YOUR_ADDRESS> --name "v0.2.0" --approve
```

- Stake-weighted voting
- Double-vote prevention (each validator votes once)
- Quorum: 67% by default (configurable via `--upgrade-quorum`)
- Gas: 40,000

Cancel an Upgrade

Cancel a scheduled upgrade before it executes.

```
vexidus --key /path/to/upgrade-authority.key upgrade cancel --from
<AUTHORITY> --name "v0.2.0"
```

- Only the upgrade authority can cancel
- Gas: 60,000

View Active Plan

```
vexidus upgrade plan
```

Pre-stage a Binary

Download and verify the upgrade binary before the target height is reached. This eliminates download latency during the actual upgrade.

```
vexidus upgrade prepare --rpc http://localhost:9933
```

The binary is downloaded, checksum-verified, and stored in VexVisor's pre-staged upgrade directory. VexVisor will use the pre-staged binary at upgrade height instead of downloading.

Emergency Rollback

Interactive rollback that lists available backup binaries and lets you select which to restore:

```
vexidus upgrade rollback
```

The command finds all `.pre-*` backup binaries (created automatically by VexVisor on each upgrade), displays them sorted by date, and performs the swap.

RPC Methods

Method	Description
`vex_scheduleUpgrade`	Schedule a network upgrade
`vex_cancelUpgrade`	Cancel a scheduled upgrade
`vex_voteUpgrade`	Vote on a scheduled upgrade

Method	Description
`vex_getUpgradePlan`	View the active upgrade plan

`vex_scheduleUpgrade`, `vex_cancelUpgrade` and `vex_voteUpgrade` build an unsigned transaction on the node and are refused (`refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`). Use the CLI with `--key`, or build and sign the operation with the SDK and submit it with `vex_submitBundle`. `vex_getUpgradePlan` is a read method and is unaffected.

The VexVisor Wrapper

In production, all validators run via the VexVisor wrapper script. This wrapper:

1. **Monitors exit codes** -- exit(42) means "upgrade required"
2. **Downloads the new binary** from the URL specified in the upgrade plan (or uses a pre-staged binary if available)
3. **Verifies SHA256 checksum** -- rejects tampered binaries
4. **Ed25519 signature verification** (Tier 1) -- optionally verifies the binary is signed by the upgrade authority's Ed25519 key. Enable with `--require-signed-upgrades`
5. **Restarts with new binary** -- automatic, no manual intervention
6. **Auto-rollback** -- if the new binary crashes within 60 seconds, rolls back to the previous version
7. **Exponential backoff** -- retries with increasing delays on repeated failures

Upgrade Lifecycle

```
Schedule Upgrade (RPC/CLI)
    |
    v
Voting Period (full mode only)
    |  67% stake-weighted quorum required
    v
Approved -> Target Height Reached
    |
    v
Node writes upgrade-info.json + exits(42)
    |
    v
VexVisor downloads binary + SHA256 verify
    |
    v
Restart with new binary
    |
    v
Grace Period (100 blocks) -- no jailing during upgrade
```

Upgrade Grace Period

After an upgrade halt, a grace period of 100 blocks (~200 seconds) suppresses missed-block tracking. This prevents validators from being jailed simply for performing an upgrade.

Expiry

Upgrade proposals that are not approved before `voting_ends_at_height` are automatically cancelled via `expire_upgrade_voting()`.

CLI Flags

Flag	Default	Description
`--governance-mode`	`seed-authority`	Governance mode (seed-authority or full)
`--upgrade-authority`	none	Hex pubkey of the upgrade authority

Flag	Default	Description
`--voting-period`	500	Voting period in blocks
`--upgrade-quorum`	67	Required approval quorum (percent)
`--auto-vote-seed`	none	Auto-vote seed for deterministic voting

Track Record

VexVisor has been proven in production with **32+ successful upgrades** across 7 validators, zero manual intervention:

#	Name	Block	What Changed
1-5	Early testnet	Various	VexBridge v3, metadata, bridge map registry
6	`bridge-map-registry`	58,250	Canonical bridge map
7	`bridge-map-nonce-fix`	59,300	Foundation nonce fix
8	`dragonfly-stream`	91,323	Mempoolless PQ-sealed pipeline
9	`lthash-state-root`	132,950	LtHash homomorphic state root
10	`lthash-audit-fix`	150,400	6 LtHash bypass fixes + sync asymmetry
11	`delegation-rpc`	161,200	Delegation, pool rewards, auto-compound
12	`dragonfly-phase3`	200,551	Mempool eliminated, sealed forwarding
13	`p2p-auth`	217,748	Validator identity + direct-to-leader
14-15	Goal + case fixes	Various	Goal::Transfer, case-insensitive ops
16	`vexidex-rebuild`	~300K	DEX AMM rebuild + adaptive block timing
17-18	Reward split + LtHash	Various	Foundation 15/85 split, LtHash race fix
19-21	Risk management	Various	Freeze, circuit breaker, state correction

#	Name	Block	What Changed
22	`state-preserving-genesis`	~515,919	Export/import genesis restart
23	`intent-decimals`	~816,300	Per-token decimals in intent parser
24	`ws-subscriptions`	~914,000	WebSocket subscriptions + leader skip

All upgrades: auto-halt at target height, SHA256-verified download, automatic restart. Average downtime per upgrade: ~3 seconds.

Code Signing (Tier 1)

VexVisor supports Ed25519 code signing for upgrade binaries. When enabled, validators reject binaries that are not signed by the upgrade authority's key.

How it works:

1. The upgrade authority signs the binary with their Ed25519 key: `vexidus upgrade sign --binary vexidus-node --key authority.hex`
2. The signature is included in the upgrade schedule transaction
3. VexVisor verifies the signature before accepting the binary

CLI flags:

Flag	Description
`--require-signed-upgrades`	Reject unsigned upgrade binaries
`--upgrade-authority`	Hex pubkey of the signing authority

This is **Tier 1** of the code signing roadmap. Future tiers will add multi-key signing (M-of-N validator committee) and reproducible build attestation.

Auto-Claim Rewards

Validators can set up automatic reward claiming to ensure they always have VXS for gas fees. A cron job runs the `tools/claim-rewards.sh` script daily:

```
# Daily at 06:00 UTC
0 6 * * * /path/to/claim-rewards.sh \
  --validator "0xYOUR_VALIDATOR_ADDRESS" \
  --rpc http://localhost:9933 \
  >> /var/log/vexidus-claim.log 2>&1
```

The script checks pending rewards via `vex_getValidator`, and if rewards are available, calls `vex_claimRewards` to collect them into the validator's balance.

Evolution Plan

1. **Tier 1 (Current):** Seed-authority mode with Ed25519 code signing. 32+ upgrades deployed successfully.
2. **Tier 2:** Full mode with auto-vote from seed -- automated governance
3. **Tier 3:** Real stake-weighted voting with manual validator decisions + multi-key code signing
4. **Tier 4:** VSC-88 protocol governance (parameter changes, treasury, multi-sig) -- **LIVE** + VSC-99 DAO program (token-weighted community governance) -- **LIVE**

SDK

```
use vexidus_sdk::BundleBuilder;

// Schedule an upgrade
let bundle = BundleBuilder::new(&sender)?
    .schedule_upgrade(name, height, url, checksum, info_url)
    .sign(&wallet);

// Cancel an upgrade
let bundle = BundleBuilder::new(&sender)?
    .cancel_upgrade(name)
    .sign(&wallet);

// Vote on an upgrade
let bundle = BundleBuilder::new(&sender)?
    .vote_upgrade(name, approve)
    .sign(&wallet);
```

Vextaris SDK & Wallet Developer Guide

Build wallets, dApps, and tools that interact with the Vextaris blockchain.

Table of Contents

1. JavaScript SDK (Quick Start)
2. Vextaris Address System
3. Wallet Compatibility
4. MetaMask Integration
5. Quick Start
6. Transaction Signing Flow
7. Address Format Conversion
8. Security
9. Troubleshooting

JavaScript SDK

The `@vexidus/sdk` is a pure TypeScript SDK for building on Vextaris. Zero native dependencies — works in Node.js 18+, Bun, Deno, and browsers.

Install

```
npm install @vexidus/sdk
```

> ⚠ **NOT YET ON THE PUBLIC NPM REGISTRY**
>
> `@vexidus/sdk` is **not published on npmjs.com yet**, so the command above will fail with `404 Not Found` today. Beta participants receive the package directly — email

validators@vexidus.com. The API below is current; only the distribution channel is pending.

Quick Start

> ⚠ **SIGN EVERYTHING, SIGN CANONICAL**
>
> The network refuses unsigned transactions (since 2026-10-31): every write is built and signed on the client and submitted with `vex_submitBundle`, exactly as below. Server-built write RPCs such as `vex_transfer` are refused on the public network.
>
> Legacy signatures stop being accepted at block **33,000,000**. In `@vexidus/sdk`, pass `'canonical'` to `sign()` / `signAndSerialize()` as every example on this page does. See Transaction Signing Flow.

```
import { WalletKeypair, BundleBuilder, RpcClient, ONE_VXS } from
'@vexidus/sdk';

// Connect to testnet
const rpc = new RpcClient('https://testnet.vextaris.io');

// Generate a wallet
const wallet = WalletKeypair.generate();
console.log('Address:', wallet.vx0Address());  // Vx0abc...
console.log('EVM:', wallet.evmAddress());       // 0xabc...

// Send 10 VXS
const nonce = await rpc.getNonce(wallet.vx0Address());
const txHash = await rpc.submitBundleHex(
  new BundleBuilder(wallet.vx0Address())
    .transfer('Vx0recipient...', 'VXS', 10n * ONE_VXS)
    .nonce(nonce)
    .signAndSerialize(wallet, 'canonical')
);
console.log('TX:', txHash);
```

What's Included

Module	Purpose
`WalletKeypair`	Ed25519 keypair generation, Vx0/0x address derivation, bundle signing
`BundleBuilder`	Fluent API for Transfer, Swap, Stake, DEX, Governance, VNS, Token creation
`RpcClient`	Typed JSON-RPC wrappers (submitBundle, getBalance, getNonce, quoteSwap, etc.)
Address utilities	Parse/validate/convert all Vx and 0x address formats

Swap Tokens

```
new BundleBuilder(myAddress)
  .swap('VXS', 'Vx1tokenMint...', 1n * ONE_VXS, 0n)
  .nonce(nonce)
  .signAndSerialize(wallet, 'canonical');
```

Stake VXS

```
new BundleBuilder(myAddress)
  .delegate('0xvalidator...', 100n * ONE_VXS)
  .nonce(nonce)
  .signAndSerialize(wallet, 'canonical');
```

Multi-Operation Bundles

Combine multiple operations in a single atomic transaction:

```
new BundleBuilder(myAddress)
  .transfer(recipient1, 'VXS', 5n * ONE_VXS)
  .transfer(recipient2, 'VXS', 3n * ONE_VXS)
  .claimRewards()
  .nonce(nonce)
  .signAndSerialize(wallet, 'canonical');
```

Sponsored Transactions

A bundle can name a different account as gas payer. The sponsor must co-sign the same bundle to consent; the builder records the sponsor but does not produce the sponsor's signature.

```
new BundleBuilder(myAddress)
  .transfer(recipient, 'VXS', 1n * ONE_VXS)
  .gasSponsor(sponsorAddress)
  .nonce(nonce)
  .sign(wallet, 'canonical');   // the sponsor's co-signature is added separately
```

Note that **new accounts get the first 10 bundles auto-sponsored by the Foundation** — set no sponsor at all and gas is silently paid for you while the quota lasts. `gasSponsor` (Rust: `with_sponsor`) is for the *explicit* sponsorship flow (DAO funding a community event, dApp paying for its users, etc.).

Atomence Operations

Atomence (cross-chain escrow swaps) operations are signed like any other bundle. The **Rust SDK** has typed builders for them (see Installation below for setup); the CLI `atomence` commands wrap the same builders.

```rust
// User: open an escrow order
let bundle = BundleBuilder::new(&wallet.vx0_address())?
    .atomence_create_order(
        "VXS", 1_000_000_000,     // token, amount (1 VXS)
        "solana", sol_address,    // destination chain + your address
there
        50,                       // max spread (bps)
        0,                        // expiry secs; 0 = network default (15
min)
    )?
    .nonce(nonce)
    .sign(&wallet);

// Market maker: register / top up the VXS bond (minimum 1,000 VXS;
// tier derives from the total: 1K / 10K / 100K VXS)
let bundle = BundleBuilder::new(&mm.vx0_address())?
    .atomence_register_mm(100_000 * 1_000_000_000)
    .nonce(nonce)
    .sign(&mm);

// User: cancel a pending order, or reclaim a matched order after its
settle deadline
let bundle = BundleBuilder::new(&wallet.vx0_address())?
    .atomence_cancel(order_id)
    .nonce(nonce)
    .sign(&wallet);
```

NEW ORDER RULES FROM BLOCK 30,655,525

From activation, an order must state **what the user receives** -- the destination asset (`"native"` or the token contract / mint on the destination chain) and the **minimum amount**. Orders without these cannot be matched after activation; cancel them while pending, or reclaim them once matched and past the settle deadline. The `atomence_create_order` builder shown above creates orders without these fields. Only registered market makers can match, and a matched order must be settled within one hour or the user can reclaim the full escrow plus 10% of the market maker's bond. Query an order's state with `vex_getAtomenceOrder` (fields in the RPC reference).

Key Details

- **VXS has 9 decimals:** `1 VXS = 1_000_000_000n` raw units (use `ONE_VXS` constant)

- **Bundle = atomic multi-op:** Unlike Ethereum (1 tx = 1 action), Vextaris bundles multiple operations in a single signature
- **Signed only:** every write is signed on the client and submitted with `vex_submitBundle`; the network refuses unsigned transactions
- **Ed25519 signatures:** Not secp256k1. Keys are 32-byte secret + 32-byte public. Sign with the canonical scheme (legacy is refused from block 33,000,000)
- **Borsh serialization:** Bundles are Borsh-encoded (not RLP/ABI) for wire format

For complete API reference, see the SDK repository on GitHub.

Vextaris Address System

Vextaris is **not Ethereum**. It is its own L1 blockchain with Ethereum *compatibility*. This has direct implications for wallet developers.

The Dual-Format Model

Format	Size	Encoding	Used By	Example
Vx0	20-byte payload	Base58 + SHA256 checksum	Native wallets, `vex_*` RPC	`Vx0Hk8pQwB5Z...`
Vx1	20-byte payload	Base58 + SHA256 checksum	Token/contract mints	`Vx1RjT4nQ9...`
0x (32-byte)	32 bytes	Hex	Internal state, system addresses	`0x0000...0001`
0x (20-byte)	20 bytes	Hex	MetaMask, `eth_*` RPC	`0x71C7...976F`

Vx0 is the canonical user-facing format. The `0x` format exists only for EVM tooling compatibility.

How Addresses Are Derived

```
Ed25519 Public Key (32 bytes)
        |
        v
    SHA256 hash
        |
        v
  First 20 bytes (payload)
        |
        v
  [0x01] + [20 bytes] + [SHA256 checksum, 4 bytes]
        |
        v
    Base58 encode
        |
        v
    Prefix "Vx0"
        |
        v
  "Vx0Hk8pQwB5Z..."  (native address)
```

For internal state storage, the 20-byte payload is **right-aligned** in a 32-byte array:

```
[0x00, 0x00, ..., 0x00, payload_byte_1, ..., payload_byte_20]
 |--- 12 zero bytes ---|------------ 20 bytes --------------|
```

For EVM wallets (MetaMask), only the **last 20 bytes** of the 32-byte array are shown:

```
32-byte internal:  0x0000000000000000000000000abcdef...
EVM display:       0xabcdef...  (last 20 bytes)
```

Key Differences from Ethereum

	Ethereum	Vextaris
Address format	`0x` (20 bytes)	`Vx0` (native) or `0x` (EVM compat)
Signature scheme	secp256k1 (ECDSA)	Ed25519
Native decimals	18 (ETH)	9 (VXS)

	Ethereum	Vextaris
Transaction model	Single operation	Bundle (atomic multi-op)
Key derivation	Keccak256(pubkey) -> 20 bytes	SHA256(pubkey) -> 20 bytes -> Vx0

Wallet Compatibility

Wallet	Works?	Method	Notes
Native Vextaris wallet	Full support	`vex_submitBundle`	Vx0 addresses, Ed25519, 9 decimals
MetaMask	Read-only today	`eth_*` reads	20-byte 0x, 18->9 decimal conversion. Sends via `eth_sendRawTransaction` are currently refused by the public network
Trust Wallet	Read-only today	Same as MetaMask	Add as custom network
Phantom	Planned (post-beta)	SVM adapter	Ed25519 compatible -- same curve as Vextaris, needs RPC translation layer
Ledger / Trezor	Planned	SDK bridge	Ed25519 app required

What This Means for Developers

- **Building a native wallet?** Use the Vextaris Wallet SDK directly -- full access to all features (Vx0 addresses, VSA v2 key management, bundle signing).
- **Supporting MetaMask users?** Through the EVM adapter they can view balances and receive, with addresses shown as `0x`. Sending from MetaMask is currently refused by the public network, and some features (VSA v2, multi-op bundles) are not available there; point users who need to send to VexSpark.
- **Both?** Your backend uses the native SDK, your frontend detects MetaMask and falls back to `eth_*` methods.

MetaMask Integration

MetaMask works with Vextaris via the built-in EVM compatibility adapter. Users add Vextaris as a custom network.

> ⚠ **SENDING FROM METAMASK IS CURRENTLY REFUSED**
>
> Since unsigned transactions were removed from the network (2026-10-31), the public testnet refuses transactions sent through `eth_sendRawTransaction`. Balances, blocks and token reads through MetaMask still work. To send, use VexSpark, the SDK, or the CLI.

Adding Vextaris to MetaMask (JavaScript)

```
await window.ethereum.request({
  method: 'wallet_addEthereumChain',
  params: [{
    chainId: '0x18b070',           // 1618032 (testnet)
    chainName: 'Vextaris Testnet',
    nativeCurrency: {
      name: 'VXS',
      symbol: 'VXS',
      decimals: 18,                 // MetaMask expects 18; adapter
scales internally
    },
    rpcUrls: ['https://testnet.vextaris.io'],
    blockExplorerUrls: ['https://vexscan.io']
  }]
});
```

Chain IDs

Network	Decimal	Hex
Testnet	1618032	`0x18b070`
Mainnet	1618033	`0x18b071`

The Decimal Issue

VXS natively has **9 decimals**, but MetaMask's UI works best with 18. The EVM adapter handles this:

- `eth_getBalance` returns VXS balance scaled to 18 decimals (`balance * 10^9`)
- `eth_sendRawTransaction` converts value from 18 decimals back to 9 (`value / 10^9`) (sends through it are currently refused -- see above)
- This means MetaMask displays correct human-readable amounts

If you are building a native wallet, always use 9 decimals. The 18-decimal scaling only applies to MetaMask's EVM adapter.

```
1.0 VXS = 1,000,000,000 base units (9 decimals, native)
1.0 VXS = 1,000,000,000,000,000,000 wei (18 decimals, MetaMask display)
```

Address Mapping

When a MetaMask user connects:

1. MetaMask provides a 20-byte `0x` address (derived from secp256k1, NOT Ed25519)
2. The EVM adapter right-aligns this to 32 bytes for internal use
3. This 32-byte address is the user's Vextaris account for `eth_*` operations

Important: A MetaMask `0x` address and a native `Vx0` address for the same person are **different accounts** (different key derivation). To use the same funds in both, a transfer between them is needed.

Quick Start

Installation

Add to your `Cargo.toml`:

```
[dependencies]
vexidus-sdk = { git = "https://github.com/vexidus-labs/vexidus-sdk",
branch = "main" }
tokio = { version = "1", features = ["full"] }
```

Generate a Wallet

```
use vexidus_sdk::WalletKeypair;

fn main() {
    let wallet = WalletKeypair::generate();
    wallet.save("my_wallet.key").unwrap();

    println!("Native address: {}", wallet.vx0_address());   // Vx0...
    println!("Hex address:    {}", wallet.hex_address());    // 0x...
(32-byte)
    println!("EVM address:    {}", wallet.evm_address());    // 0x...
(20-byte)
}
```

Load an Existing Wallet

```
// From file
let wallet = WalletKeypair::load("my_wallet.key")?;

// From hex string
let wallet = WalletKeypair::from_secret_hex("abcdef1234...")?;
```

Check Balance

```rust
use vexidus_sdk::WalletClient;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let client = WalletClient::new("https://testnet.vextaris.io");
    let wallet = WalletKeypair::load("my_wallet.key")?;

    let balance = client.get_balance(&wallet.vx0_address(),
"VXS").await?;
    println!("Balance: {} VXS", balance);

    Ok(())
}
```

Send a Transfer

```rust
use vexidus_sdk::{WalletKeypair, WalletClient};

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let client = WalletClient::new("https://testnet.vextaris.io");
    let wallet = WalletKeypair::load("my_wallet.key")?;

    // Convenience method: auto-fetches nonce, signs, submits
    let tx_hash = client.transfer(
        &wallet,
        "Vx0RecipientAddress...",
        "VXS",
        5_000_000_000,  // 5.0 VXS (9 decimals)
    ).await?;

    println!("Transaction: {}", tx_hash);
    Ok(())
}
```

Build a Custom Bundle

For more control, use the `BundleBuilder`:

```rust
use vexidus_sdk::{BundleBuilder, WalletKeypair, WalletClient};

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let client = WalletClient::new("https://testnet.vextaris.io");
    let wallet = WalletKeypair::load("my_wallet.key")?;

    let nonce = client.get_nonce(&wallet.hex_address()).await?;

    let bundle = BundleBuilder::new(&wallet.vx0_address())?
        .transfer("Vx0Recipient...", "VXS", 5_000_000_000)?
        .nonce(nonce)
        .max_gas(100_000)
        .valid_for(3600)   // Valid for 1 hour
        .sign(&wallet);    // Ed25519 signature over bundle hash

    let tx_hash = client.submit_bundle(&bundle).await?;
    println!("Transaction: {}", tx_hash);

    Ok(())
}
```

Transaction Signing Flow

Vextaris uses a **bundle model** -- all operations in a bundle are executed atomically.

Bundle Structure

A signed bundle carries everything the network needs to execute and verify it atomically:

Field	Purpose
Sender	The account submitting the bundle
Operations	One or more atomic operations executed together
Gas limit	Maximum gas the bundle may consume
Priority fee	Optional fee to prioritize inclusion

Field	Purpose
Validity window	Expiry time after which the bundle is rejected
Nonce	Replay protection (sequential per account)
Signature	64-byte Ed25519 signature over the bundle hash

The SDK builds and serializes this structure for you via `BundleBuilder` — you never assemble it by hand.

Available Operations

Operation	Description
`Transfer`	Send tokens (VXS or VSC-7) to an address
`Stake`	Register as a validator with VXS
`Unstake`	Begin 21-day unbonding
`ClaimUnstake`	Claim matured unstake
`Delegate`	Delegate VXS to a validator
`Undelegate`	Undelegate from a validator
`ClaimRewards`	Claim validator rewards
`AddKey`	VSA v2: Add a signing key to account
`RemoveKey`	VSA v2: Remove a key by hash
`RotateKey`	VSA v2: Atomic key swap
`SetRecovery`	VSA v2: Configure guardian recovery

Signing Process

```
1. Build the bundle (signature field left empty)
              |
              v
2. Compute the canonical signing message (see the note below)
              |
              v
3. Sign 32-byte hash with Ed25519 -> 64-byte signature
              |
              v
4. Set bundle.signature = Signature(sig_bytes)
              |
              v
5. Borsh-serialize entire bundle -> bytes
              |
              v
6. Hex-encode -> "0x..."
              |
              v
7. Submit via vex_submitBundle("0x...")
```

The SDK handles steps 2-7 automatically when you call `.sign(&wallet)` and `client.submit_bundle()`.

⚠ CANONICAL SIGNING -- LEGACY SIGNATURES ARE REFUSED FROM BLOCK 33,000,000

Two signature schemes exist. **Canonical** signs a domain-separated encoding that covers the operation list unambiguously, alongside the chain id and the sender's public key. **Legacy** is the older encoding. The transaction id is the same under both.

step	height	behaviour
accept	**18,700,000** (live)	both legacy and canonical signatures are accepted
require	**33,000,000**	only canonical signatures are accepted; legacy-signed transactions are refused

- **Rust SDK, CLI and VexSpark** sign canonical by default -- take the current release.
- **`@vexidus/sdk`:** pass `'canonical'` to `sign()` / `signAndSerialize()` (as the examples on this page do).
- **Older builds** of any of these must be updated before block 33,000,000.

- **If you compute the signing message yourself**, you must implement the canonical encoding. The SDKs expose a known-answer test vector to check an independent implementation against; do not infer the encoding from the diagram above.

Pubkey Revelation (First-Time Sends)

Accounts created by *receiving* a transfer have a dummy public key (address bytes, not a real Ed25519 key). Ed25519 cannot recover the public key from a signature alone, so the first signed send from such an account must include the sender's public key for verification.

How it works:

1. The wallet includes the sender's Ed25519 public key alongside the signed bundle
2. The network confirms the public key hashes to the sender's address
3. The network verifies the Ed25519 signature against the provided public key
4. On success, the public key is recorded for the account -- future sends do not need revelation

SDK usage:

```
// First send from a receive-only account
let bundle = BundleBuilder::new(&sender_hex)?
    .transfer(to, "VXS", amount)?
    .sender_pubkey(my_ed25519_pubkey.to_vec())  // Include pubkey for
revelation
    .sign(&wallet);
```

The native wallet **VexSpark** (https://wallet.vexspark.com) handles pubkey revelation automatically -- users do not need to think about it.

Address Format Conversion

Using the SDK

```rust
use vexidus_sdk::address_utils;

// Derive Vx0 from public key
let vx0 = address_utils::vx0_from_pubkey(&pubkey_bytes);

// Vx0 -> 0x hex (32-byte internal format)
let hex_addr = address_utils::vx0_to_hex("Vx0abc...")?;

// Vx0 -> 0x EVM (20-byte, what MetaMask shows)
let evm_addr = address_utils::vx0_to_evm("Vx0abc...")?;

// Validate
assert!(address_utils::is_valid_vx0("Vx0abc..."));
assert!(address_utils::is_valid_hex_address("0x71C7..."));

// Parse any format -> 32-byte Address
let addr = address_utils::parse_address("Vx0abc...")?;   // Works
let addr = address_utils::parse_address("0x71C7...")?;   // Also works
```

Conversion Matrix

From	To	Method	Lossless?
Public key -> Vx0	`vx0_from_pubkey()`	Yes	
Vx0 -> 0x (32-byte)	`vx0_to_hex()`	Yes	
Vx0 -> 0x (20-byte EVM)	`vx0_to_evm()`	Yes	
0x (20-byte) -> 32-byte	`parse_address()`	Yes (right-aligned)	
0x (32-byte) -> Vx0	Not available offline	No -- requires pubkey	

The conversion from `0x` back to `Vx0` is not possible offline because the Vx0 encoding includes a SHA256 checksum computed from the *original* pubkey hash, not from the padded 32-byte form. You would need the original public key or a node-side lookup.

Security

Key Storage

- Save wallet keys with `chmod 600` (the SDK does this automatically on `save()`)
- Never log or print secret keys in production
- Store backups encrypted on offline media
- For web wallets, use the browser's `SubtleCrypto` API to encrypt keys at rest

Testnet vs Mainnet

Behavior	Testnet	Mainnet
Chain ID	1618032 (`0x18b070`)	1618033 (`0x18b071`)
`vex_generateKeypair`	Available	Disabled

Never reuse testnet keys on mainnet. Generate fresh keypairs for production.

Nonce Management

- Nonces are sequential: transaction with nonce=5 only executes after nonce=4
- Always fetch the current nonce immediately before signing
- For sequential transactions, increment locally: `nonce`, `nonce+1`, `nonce+2`
- If a transaction fails, the nonce is not consumed -- retry with the same nonce

Replay Protection

Each bundle includes:

- **nonce**: Prevents duplicate execution
- **valid_until**: Time-bounds the transaction (stale bundles are rejected)
- **chain_id**: Covered by the canonical signature, so a transaction signed for one network cannot be replayed on another

---

Troubleshooting

"Invalid signature"

- Vextaris uses **Ed25519**, not secp256k1 (Ethereum). Make sure you are signing with the correct algorithm.
- Sign with the **canonical** scheme. Legacy signatures are refused from block 33,000,000 -- update an older SDK, CLI or wallet build.
- The signature covers the 32-byte signing message the SDK computes, not the raw bundle bytes.

"refused: unsigned bundle"

- The network accepts only transactions signed on the client. You called a server-built write method (such as `vex_transfer`, `vex_submitIntent` or `vex_bridgeToken`) or submitted a bundle with no signature.
- Build and sign the operation with the SDK, the CLI (`--key`) or VexSpark, and submit it with `vex_submitBundle`.

"Invalid nonce"

- The nonce must exactly match the account's current nonce (not current+1).
- Query with `client.get_nonce()` immediately before signing.
- If you see `expected N, got M`: a previous transaction may be pending in the mempool.

"Address must start with Vx0, Vx1, or 0x"

- Check that you are passing the address in the right format for the RPC method.
- `vex_*` methods accept both Vx0 and 0x.
- `eth_*` methods expect 0x only.

"Vextaris address error: Invalid checksum"

- The Vx0 address has been corrupted (truncated, modified, or copy-paste error).
- Regenerate from the public key using `address_utils::vx0_from_pubkey()`.

Balance shows 0 but I funded the account

- Check you are querying the right address format. A Vx0 address and an 0x address for the same person may map to different internal accounts if derived from different keys.
- Use `vex_getBalance` with the exact address you funded.

MetaMask shows wrong balance

- VXS has 9 decimals natively. The EVM adapter scales to 18 for MetaMask display.
- If your custom RPC does not scale, MetaMask will show 10^9x too small.

Borsh serialization errors

- Ensure your SDK version matches the node version. The bundle Borsh layout must be identical on both sides.
- Update with `git pull && cargo build --release`.

API Reference

WalletKeypair

Method	Returns	Description
`generate()`	`WalletKeypair`	New random Ed25519 keypair
`from_secret_hex(hex)`	`Result<WalletKeypair>`	Load from hex string
`from_secret_bytes(bytes)`	`WalletKeypair`	Load from 32-byte array
`load(path)`	`Result<WalletKeypair>`	Load from file
`save(path)`	`Result<()>`	Save to file (chmod 600)
`vx0_address()`	`String`	Native Vx0 address
`hex_address()`	`String`	0x hex (32-byte)
`evm_address()`	`String`	0x EVM (20-byte)

Method	Returns	Description
`public_key_bytes()`	`[u8; 32]`	Raw public key
`sign(message)`	`Vec<u8>`	64-byte Ed25519 signature
`sign_bundle(bundle)`	`Signature`	Sign a TransactionBundle (canonical scheme)

BundleBuilder

Method	Returns	Description
`new(sender)`	`Result<BundleBuilder>`	Create builder (accepts Vx0 or 0x)
`.transfer(to, token, amount)`	`Result<Self>`	Add transfer operation
`.stake(amount, pubkey)`	`Self`	Add stake operation
`.unstake(amount)`	`Self`	Add unstake operation
`.delegate(validator, amount)`	`Result<Self>`	Add delegate operation
`.claim_rewards()`	`Self`	Add claim rewards operation
`.add_key(pubkey, type, role)`	`Self`	VSA v2: Add key
`.nonce(n)`	`Self`	Set nonce
`.max_gas(g)`	`Self`	Set gas limit
`.valid_for(seconds)`	`Self`	Set validity window
`.build()`	`TransactionBundle`	Build the bundle without a signature (the network refuses it until signed)
`.sign(wallet)`	`TransactionBundle`	Build and sign (canonical scheme)

WalletClient

Method	Returns	Description
`new(rpc_url)`	`WalletClient`	Create client
`get_balance(addr, token)`	`Result<String>`	Query token balance
`get_nonce(addr)`	`Result<u64>`	Query account nonce
`submit_bundle(bundle)`	`Result<String>`	Submit signed bundle
`transfer(wallet, to, token, amt)`	`Result<String>`	Convenience: fetch nonce, sign locally, submit via `vex_submitBundle`
`get_token_info(symbol)`	`Result<Value>`	Token metadata
`list_tokens(limit)`	`Result<Value>`	List registered tokens
`chain_id()`	`Result<String>`	Chain ID (hex)
`block_number()`	`Result<u64>`	Current block height
`is_healthy()`	`bool`	Node reachability check

address_utils

Function	Returns	Description
`vx0_from_pubkey(bytes)`	`String`	Derive Vx0 from public key
`vx0_to_bytes(vx0)`	`Result<[u8; 32]>`	Decode to 32-byte internal
`vx0_to_hex(vx0)`	`Result<String>`	Convert to 0x hex (32-byte)
`vx0_to_evm(vx0)`	`Result<String>`	Convert to 0x EVM (20-byte)
`is_valid_vx0(addr)`	`bool`	Validate Vx0 checksum
`is_valid_hex_address(addr)`	`bool`	Validate 0x format

Function	Returns	Description
`parse_address(input)`	`Result<Address>`	Parse any format to 32-byte

Vextaris is its own L1 blockchain. Native addresses use the `Vx` prefix. The `0x` format exists only for EVM compatibility.

RPC Reference

Vextaris exposes 110+ JSON-RPC endpoints organized into two namespaces: `vex_*` (native) and `eth_*` (EVM compatibility).

> **TIP**
>
> All examples use the public testnet endpoint `https://testnet.vextaris.io`. Replace with your local node address (e.g. `http://localhost:9933`) if running your own validator.

All endpoints accept standard JSON-RPC 2.0 format:

```
curl -s https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"METHOD_NAME","params":[...],"id":1}'
```

> **TRANSACTIONS MUST BE SIGNED (SINCE 2026-10-31)**
>
> Every validator refuses unsigned transactions. An unsigned path that existed for testnet convenience was abused, so it was removed network-wide.
>
> - **To write to the chain**, build and sign the transaction on the client -- with the SDK, the CLI (`--key`), or a wallet such as VexSpark -- and submit it with **`vex_submitBundle`**. You set the nonce yourself (fetch it immediately before signing).
> - **Server-built write methods** -- the ones that take a sender address as a parameter and build the transaction on the node (marked ✍ in the tables below) -- are refused on the public network with: `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`. The CLI and SDK build the same operations locally and sign them.
> - Transactions naming the reserved system account as sender are refused at the RPC.
> - **Read methods are unaffected.**
>
> Signatures must use the **canonical** signing scheme: it is accepted since block **18,700,000**, and **legacy signatures stop being accepted at block 33,000,000**. See Transaction Signing Flow.

> **NOTE**
>
> Methods marked ✍ are listed for reference. Each one corresponds to an operation you can sign with the SDK or CLI and submit through `vex_submitBundle`.

Native (vex_*)

Token and Balance

Method	Description
`vex_getBalance`	Get token balance for an address. Params: `[address, token_symbol]`
`vex_transfer`	✍ Transfer tokens (server-built). Params: `[from, to, token, amount]`. Use a signed `Transfer` via `vex_submitBundle` instead.
`vex_createToken`	✍ Create a VSC-7 fungible token. Params: `[name, symbol, decimals, supply, owner]`
`vex_getTokenInfo`	Get token metadata by symbol or mint address. Params: `[symbol_or_mint]`
`vex_listTokens`	List all registered tokens. Params: `[limit]`
`vex_updateTokenMetadata`	✍ Update mutable token metadata (creator only). Params: `[mint, sender, metadata_uri?, image?, description?, external_url?, twitter?, discord?, telegram?, github?]`
`vex_lockTokenMetadata`	✍ Permanently lock token metadata (one-way). Params: `[mint, sender]`
`vex_getTokenReputation`	Get token reputation score (VSC-REP). Params: `[mint]`

DEX (VexiDEX)

Method	Description
`vex_createPool`	✍ Create a new liquidity pool. Params: `[token_a, token_b, amount_a, amount_b, lp_lock_duration, sender]`
`vex_addLiquidity`	✍ Add liquidity to existing pool. Params: `[token_a, token_b, amount_a, amount_b, min_lp_tokens, sender]`
`vex_removeLiquidity`	✍ Remove liquidity and redeem LP tokens. Params: `[token_a, token_b, lp_amount, min_amount_a, min_amount_b, sender]`
`vex_swap`	✍ Swap tokens through a pool. Params: `[from_token, to_token, amount_in, min_amount_out, sender]`
`vex_getPool`	Get pool info by token pair. Params: `[token_a, token_b]`
`vex_listPools`	List all pools. Params: `[limit]`
`vex_quoteSwap`	Get swap quote (read-only). Params: `[from_token, to_token, amount_in]`

NFT (VSC-21)

Method	Description
`vex_createNftCollection`	✍ Create NFT collection. Params: `[name, symbol, base_uri, max_supply, royalty_bps, owner, description?, icon_uri?, banner_uri?, external_url?]`
`vex_mintNft`	✍ Mint an NFT. Params: `[collection, token_id, metadata_uri, recipient, sender]`
`vex_transferNft`	✍ Transfer an NFT. Params: `[collection, token_id, to, owner]`
`vex_burnNft`	✍ Burn an NFT (owner only). Params: `[collection, token_id, owner]`
`vex_getNft`	Query NFT by collection + token ID. Params: `[collection, token_id]`
`vex_listNftsByOwner`	List NFTs owned by address. Params: `[owner, collection?, limit]`
`vex_getNftCollection`	Collection metadata and stats. Params: `[collection]`

Multi-Token (VSC-55)

Method	Description
`vex_createMultiTokenCollection`	✍ Create multi-token collection. Params: `[name, symbol, base_uri, royalty_bps, owner, description?, icon_uri?, banner_uri?, external_url?]`
`vex_defineTokenType`	✍ Define a token type in collection. Params: `[collection, type_id, name, metadata_uri, max_supply, is_fungible, sender]`
`vex_mintMultiToken`	✍ Mint tokens of a type. Params: `[collection, type_id, recipient, amount, sender]`
`vex_batchMintMultiToken`	✍ Batch mint multiple types. Params: `[collection, mints_array, sender]`
`vex_transferMultiToken`	✍ Transfer tokens of a type. Params: `[collection, type_id, to, amount, from]`
`vex_batchTransferMultiToken`	✍ Batch transfer multiple types. Params: `[collection, transfers_array, sender]`
`vex_burnMultiToken`	✍ Burn tokens. Params: `[collection, type_id, amount, owner]`
`vex_getMultiTokenBalance`	Query balance. Params: `[collection, type_id, owner]`
`vex_getTokenType`	Token type definition. Params: `[collection, type_id]`
`vex_getMultiTokenCollection`	Collection metadata. Params: `[collection]`
`vex_listTokenTypes`	List types in collection. Params: `[collection, limit]`

Presale (VSC-LAUNCH)

Method	Description
`vex_createPresale`	✍ Create a presale with anti-rug protection.
`vex_contributePresale`	✍ Contribute to a presale.
`vex_claimPresale`	✍ Claim tokens after a successful presale.
`vex_finalizePresale`	✍ Finalize presale and auto-create LP.

Method	Description
`vex_cancelPresale`	✍ Cancel a presale.
`vex_refundPresale`	✍ Refund if presale fails.
`vex_getPresaleInfo`	Get presale details. Params: `[presale_mint]`
`vex_listPresales`	List presales. Params: `[limit?, status?]`

Bridge

Method	Description
`vex_bridgeToken`	✍ Deposit tokens from external chain. Params: `[chain, tx_hash, symbol, amount, recipient, proof]`
`vex_bridgeWithdraw`	✍ Withdraw tokens to external chain. Params: `[token_mint, amount, chain, destination, sender]`
`vex_getBridgeStatus`	Query bridge operation status. Params: `[chain, tx_hash]`
`vex_getBridgeHistory`	List bridge operations for address. Params: `[address, limit]`
`vex_getBridgeFees`	Get fee config for a chain. Params: `[chain?]` (null for all)
`vex_autobridge`	✍ Universal receive -- auto-detect and process.
`vex_getPendingReceives`	List pending auto-detected receives.

Anchor (Data Attestation)

Method	Description
`vex_anchor`	✍ Anchor data to the blockchain. Params: `[namespace, key, value_hash, sender]`
`vex_getAnchors`	Query anchored data by namespace. Params: `[namespace, limit]`
`vex_verifyAnchor`	Verify an anchor exists and return its metadata. Params: `[namespace, key]`

IntentVM

Method	Description
`vex_parseIntent`	Parse free text into a structured, signable goal. Executes nothing. Params: `[intent_string]`. Amounts are returned as decimal strings. Supports swap, transfer, stake, add-liquidity and `.vex` name registration phrasings (and combinations of them); other goal types return an error.
`vex_submitIntent`	✍ Submit intent (NL or JSON) built on the server. Params: `[intent_string, sender_address]`. On the public network use `vex_parseIntent` → sign locally → `vex_submitBundle` (see the IntentVM guide).
`vex_simulateIntent`	Dry-run intent against current state. Returns per-pool quotes, price impact (bps), balance checks, auto-routing details, and warnings array. Params: `[intent_string, sender_address]`
`vex_previewIntent`	Human-readable wallet confirmation. Returns summary text, fee in VXS, price impact %, route type, slippage tolerance, and warnings. Designed for wallet confirmation UIs. Params: `[intent_string, sender_address]`
`vex_getExecutionPlan`	Preview execution steps + gas estimate. Params: `[intent_string]`
`vex_getIntentStatus`	Query intent result by hash. Params: `[bundle_hash]`
`vex_getPrice`	Get spot price between two tokens. Returns direct price or auto-routed price through intermediate pools. Params: `[from_symbol, to_symbol]`

VIP (Vextaris Intent Programs)

Method	Description
`vex_registerProgram`	✍ Register a VIP program on-chain. Params: `[program_json, sender]`
`vex_getProgram`	Get a registered VIP program by name. Params: `[name]`
`vex_listPrograms`	List all registered VIP programs. Params: `[limit]`
`vex_simulateProgram`	Simulate VIP execution (dry run). Params: `[name, params, sender]`
`vex_executeProgram`	✍ Execute a VIP program. Params: `[name, params, sender]`

Staking

Method	Description
`vex_stake`	✍ Stake VXS as validator. Params: `[staker_address, amount, validator_pubkey]`
`vex_unstake`	✍ Begin unbonding. Params: `[address, amount]`
`vex_getValidator`	Get validator details. Params: `[address]`
`vex_listValidators`	List all validators. Params: `[limit]`
`vex_stakingInfo`	Network staking summary. Params: `[]`
`vex_setCommission`	✍ Set commission rate (basis points). Params: `[address, commission_bps]`
`vex_unjail`	✍ Unjail validator after cooldown. Params: `[address]`
`vex_setValidatorMetadata`	✍ Set validator profile. Params: `[address, name, description, website, avatar]`
`vex_getDelegations`	Get delegations for address (includes auto_compound status). Params: `[address]`
`vex_setAutoCompound`	✍ Toggle auto-compound for a delegation. Params: `[validator, enable, sender]`
`vex_setPoolConfig`	✍ Configure staking pool settings (validator only). Params: `[pool_enabled, pool_name, min_delegation, auto_compound_default, sender]`

Staking Pools (VSC-20)

Method	Description
`vex_listStakingPools`	List all staking pools. Params: `[limit]`

Creating, staking into, unstaking from and claiming from a VSC-20 pool are operations you sign and submit with `vex_submitBundle`; there is no server-built RPC for them.

Explorer

Method	Description
`vex_getBlockByHeight`	Get block by height (includes tx stats). Params: `[height]`
`vex_getRecentBlocks`	Get recent blocks. Params: `[limit]`
`vex_getRecentTransactions`	Get recent transactions. Params: `[limit]`
`vex_getNetworkStats`	Network statistics. Params: `[]`
`vex_getBlockTransactions`	Transactions in a specific block. Params: `[height]`
`vex_getAddressHistory`	Transaction history for address. Params: `[address, limit]`
`vex_getTokenHolders`	Token holder list. Params: `[token_mint, limit]`
`vex_getTokenTransfers`	Token transfer history. Params: `[token_mint, limit]`
`vex_search`	Search by hash, address, or block. Params: `[query]`
`vex_getTransactionReceipt`	Transaction receipt by hash. Params: `[tx_hash]`
`vex_getAddressTokens`	List tokens held by address. Params: `[address]`

VNS (Vextaris Name Service)

Method	Description
`vex_resolveName`	Resolve `.vex` name to address. Params: `[name]`
`vex_registerName`	✍ Register a `.vex` name. Params: `[name, sender]`
`vex_reverseResolve`	Reverse lookup: address to name. Params: `[address]`
`vex_getNameInfo`	Full name details. Params: `[name]`

Upgrade Governance (VexVisor)

Method	Description
`vex_scheduleUpgrade`	✍ Schedule a network upgrade.
`vex_cancelUpgrade`	✍ Cancel a scheduled upgrade.
`vex_voteUpgrade`	✍ Vote on a scheduled upgrade.
`vex_getUpgradePlan`	View the active upgrade plan.

Protocol Governance (VSC-88)

Method	Description
`vex_createProposal`	✍ Create a governance proposal. Params: `[from, proposal_type, title, description, parameter_key?, parameter_value?, treasury_recipient?, treasury_amount?, safety_chain?, safety_contract?, safety_level?, safety_collection?]`
`vex_votePproposal`	✍ Vote on an active proposal. Params: `[from, proposal_id, approve]`
`vex_executeProposal`	✍ Execute an approved proposal. Params: `[from, proposal_id]`
`vex_cancelProposal`	✍ Cancel a proposal (proposer only). Params: `[from, proposal_id]`
`vex_getProposal`	Get proposal details by ID. Params: `[proposal_id]`
`vex_listProposals`	List proposals (optional status filter). Params: `[status?]`

Multi-Sig (VSC-88)

Method	Description
`vex_createMultiSig`	✍ Create an M-of-N multi-sig account. Params: `[from, signers_array, threshold, label]`

Method	Description
`vex_proposeMultiSigTx`	✍ Propose a transaction from a multi-sig. Params: `[from, multisig_address, operations_json, description]`
`vex_approveMultiSigTx`	✍ Approve a pending multi-sig transaction. Params: `[from, multisig_address, tx_id]`
`vex_revokeMultiSigApproval`	✍ Revoke approval on a pending tx. Params: `[from, multisig_address, tx_id]`
`vex_getMultiSig`	Get multi-sig account details. Params: `[address]`
`vex_listMultiSigTxs`	List transactions for a multi-sig. Params: `[address]`

DAO (VSC-99)

Method	Description
`vex_createDAO`	✍ Create a new DAO. Params: `[from, name, governance_token, min_tokens_to_propose, min_tokens_to_vote, voting_period_blocks, quorum_percent, approval_threshold, execution_delay_blocks, council_members, council_threshold]`
`vex_daoPropose`	✍ Submit a proposal to a DAO. Params: `[from, dao_address, proposal_type, title, description, treasury_recipient?, treasury_amount?, treasury_token?, config_changes?]`
`vex_daoVote`	✍ Vote on a DAO proposal. Params: `[from, dao_address, proposal_id, approve]`
`vex_daoExecute`	✍ Execute a passed DAO proposal. Params: `[from, dao_address, proposal_id]`
`vex_daoCancel`	✍ Cancel a DAO proposal. Params: `[from, dao_address, proposal_id]`
`vex_getDAO`	Get DAO details. Params: `[dao_address]`
`vex_listDAOs`	List DAOs (optional governance-token filter). Params: `[governance_token?]`
`vex_getDAOProposal`	Get proposal details. Params: `[dao_address, proposal_id]`

Method	Description
`vex_listDAOProposals`	List a DAO's proposals. Params: `[dao_address, status?]`
`vex_getDAOVotingPower`	Voting power of a holder. Params: `[dao_address, holder]`

Safety (VSC-SAFE)

Method	Description
`vex_getContractSafety`	Get safety record for a bridge contract. Params: `[chain, contract_address]`
`vex_listContractSafety`	List contract safety records (optional chain filter). Params: `[chain?, limit]`

Dragonfly Stream

Method	Description
`vex_getDragonflyStats`	Dragonfly Stream statistics (forwarded/received counts, seal type, enabled status). Params: `[]`

Vexcel (Attestation DAG)

Method	Description
`vex_getAttestations`	Get attestation blocks at a given height. Params: `[height]`
`vex_getBlockDAG`	Get canonical block + attestation blocks (full DAG structure). Params: `[height]`

Bridge Mappings

Method	Description
`vex_registerBridgeMapping`	✍️ Register a canonical bridge mapping (Foundation-gated). Params: `[chain, contract, mint, sender]`

Method	Description
`vex_getBridgeMappings`	List bridge mappings (optional chain filter). Params: `[chain?]`

Node Health

Method	Description
`vex_validatorHealth`	Node health: block height, peers, sync status, uptime, version. Params: `[]`
`vex_getStorageStats`	Storage tier statistics (ElasticState hot/warm/cold). Params: `[]`
`vex_getCurrentLeader`	Current block leader for a height. Params: `[height?]` (null = current)

Bushido Passport

Method	Description
`vex_getPassport`	Get a Bushido Passport by owner address. Params: `[owner]`
`vex_getPassportByName`	Look up a passport by name. Params: `[name]`
`vex_getPassportStats`	Get global passport statistics. Params: `[]`
`vex_getGameInfo`	Get a registered game. Params: `[game_id]`

Atomence

Read-only query surface for cross-chain / cross-zone settlement. Write operations (create order, match, settle, register market maker, cancel / reclaim) are signed locally and submitted with `vex_submitBundle`; there is no server-built Atomence write method.

Method	Description
`vex_getAtomenceOrder`	Look up a single order by ID. Params: `[order_id]`. Returns the order's full lifecycle record or null (fields below).

Method	Description
`vex_listAtomenceSettlements`	List Settled orders, paginated. Params: `[token?, since_block?, limit?]` (limit default 100, max 1000).
`vex_getMarketMakers`	List active market makers, optionally filtered to tier 1, 2, or 3. Params: `[tier?]`
`vex_atomenceStats`	Aggregate stats over Settled orders. Params: `[token?]`. Returns `{settled_count, total_volume, avg_spread_bps, last_settlement_block, active_mm_count}`.

`vex_getAtomenceOrder` fields: `order_id` (`Vxh…`), `user` (`Vx0…`), `token` (`Vx1…`), `amount` (string, smallest unit), `dest_chain`, `dest_address`, `max_spread_bps`, `expires_at` (unix seconds), `mm_address` (`Vx0…` or null), `created_at_block`, `matched_at_block` (or null), `status` (`Pending`, `Matched`, `Settled`, `Expired`, `Cancelled`), and three fields for the order rules that activate at block `30,655,525`:

Field	Meaning
`dest_asset`	What the user receives on the destination chain: `"native"` or the token contract / mint address there.
`min_dest_amount`	Minimum the user must receive in `dest_asset`, as a string in that asset's smallest unit.
`settle_deadline`	Unix time (seconds) by which the matched market maker must settle -- set at match time, one hour after the match. After it, the user can reclaim the escrow.

All three are `null` for orders that do not carry them (for example, orders created before activation). Such orders cannot be matched after activation; the user cancels them while pending, or reclaims them once matched and past the deadline.

Utility

Method	Description
`vex_submitBundle`	Submit a signed transaction bundle (Borsh hex). Params: `[borsh_hex]`. **The way to write to the chain.**
`vex_generateKeypair`	Generate Ed25519 keypair (testnet only). Params: `[]`

Method	Description
`vex_deriveAddress`	Derive Vextaris Smart Address from a public key. Params: `[pubkey_hex]`

WebSocket Subscriptions

Vextaris supports push-based event streaming via WebSocket on the same RPC port (9933). Connect with any WebSocket client to receive real-time notifications.

Connection

```
# Connect via wscat
wscat -c ws://localhost:9933

# Or use the public endpoint
wscat -c wss://testnet.vextaris.io
```

Subscribe to New Block Headers

Receive a notification each time a new block is finalized.

```
{"jsonrpc":"2.0","method":"vex_subscribe_newHeads","params":[],"id":1}
```

Notification format:

```
{
  "height": 925290,
  "hash": "Vxh...",
  "parentHash": "Vxh...",
  "timestamp": 1773376771,
  "proposer": "Vx0...",
  "stateRoot": "Vxh...",
  "transactionsRoot": "Vxh...",
  "gasUsed": 210000,
  "gasLimit": 5242880,
  "baseFee": 10,
  "bundleCount": 3
}
```

Unsubscribe: `{"jsonrpc":"2.0","method":"vex_unsubscribe_newHeads","params":["SUBSCRIPTION_ID"],"id":2}`

Subscribe to New Transactions

Receive a notification for each transaction bundle included in a new block.

```
{"jsonrpc":"2.0","method":"vex_subscribe_newTransactions","params":[],"id":1}
```

Notification format:

```
{
  "blockHeight": 925290,
  "blockHash": "Vxh...",
  "bundleHash": "Vxh...",
  "bundleIndex": 0,
  "from": "Vx0...",
  "operations": ["Transfer", "Swap"],
  "gasUsed": 42000,
  "timestamp": 1773376771
}
```

Subscribe to Logs (with Filter)

Receive operation-level logs with optional filtering by operation type and/or address.

```
{"jsonrpc":"2.0","method":"vex_subscribe_logs","params":[{"opType":
["Transfer","Swap"]}],"id":1}
```

Filter options:

- `opType` -- Array of operation types: `"Transfer"`, `"Swap"`, `"Stake"`, `"BridgeDeposit"`, etc.
- `address` -- Array of Vx0 addresses to filter by sender

Notification format:

```
{
  "blockHeight": 925290,
  "blockHash": "Vxh...",
  "bundleHash": "Vxh...",
  "bundleIndex": 0,
  "opIndex": 0,
  "opType": "Transfer",
  "from": "Vx0...",
  "data": { "to": "Vx0...", "token": "Vx1...", "amount": 1000000000 },
  "timestamp": 1773376771
}
```

Subscription Methods Summary

Subscribe	Unsubscribe	Item Type
`vex_subscribe_newHeads`	`vex_unsubscribe_newHeads`	Block header
`vex_subscribe_newTransactions`	`vex_unsubscribe_newTransactions`	Transaction bundle
`vex_subscribe_logs`	`vex_unsubscribe_logs`	Operation log (filterable)

Notes

- Subscriptions use a 64-block buffer. Slow consumers that fall behind will skip missed blocks (no backfill).

- During initial sync, subscriptions are paused to avoid flooding with historical blocks.
- All hash and address values use native Vextaris format (`Vxh` for hashes, `Vx0` for addresses).

EVM Compatibility (eth_*)

These endpoints provide MetaMask and EVM tooling compatibility. They translate EVM-format requests into native Vextaris state queries.

Method	Description
`eth_call`	Simulate contract call (ERC-20/721/1155 function selectors supported)
`eth_sendRawTransaction`	Submit an ECDSA-signed transaction. **Currently refused on the public network** -- see the note below.
`eth_getBalance`	Get VXS balance (scaled to 18 decimals for MetaMask)
`eth_getTransactionCount`	Get account nonce
`eth_chainId`	Returns chain ID (0x18b070 testnet, 0x18b071 mainnet)
`eth_blockNumber`	Current block height (hex)
`eth_getBlockByNumber`	Block data by height
`eth_getBlockByHash`	Block data by hash
`eth_getTransactionByHash`	Transaction data by hash
`eth_getTransactionReceipt`	Transaction receipt
`eth_getLogs`	Event logs (ERC-20 Transfer, ERC-721 Transfer, ERC-1155 events)
`eth_getCode`	Returns bytecode stub for VSC tokens (appears as contract to wallets)
`eth_estimateGas`	Gas estimation
`net_version`	Network version

Method	Description
`web3_clientVersion`	Client version string

Notes on EVM Compatibility

- **Decimal scaling:** `eth_getBalance` returns VXS scaled to 18 decimals (native is 9). MetaMask displays correct human-readable amounts.
- **Address format:** `eth_*` methods use 20-byte `0x` addresses. The adapter right-aligns to 32 bytes internally.
- **ERC-20 simulation:** `eth_call` with standard function selectors (`balanceOf`, `name`, `symbol`, `decimals`, `totalSupply`) works on VSC-7 tokens.
- **Sending from MetaMask:** since unsigned transactions were removed (2026-10-31), the public network refuses transactions sent through `eth_sendRawTransaction`. Reading balances and blocks through `eth_*` works as before. To send, use a native wallet (VexSpark), the SDK, or the CLI.
- **No bytecode execution:** Vextaris does NOT run EVM opcodes. The `eth_*` layer translates wallet RPC formats into native state queries.

CLI Reference

The Vextaris CLI (`vexidus`) provides command groups for wallets, tokens, NFTs, DEX pools, staking, governance, Atomence and more. All commands communicate with a node via JSON-RPC. Run `vexidus <group> --help` for the full flag list of any command.

Installation

The CLI binary is built alongside the node:

```
cargo build --release
# Binary at: target/release/vexidus
```

Configuration

By default, the CLI connects to `http://localhost:9933` (your own node). To use the public testnet, pass `--rpc`:

```
vexidus --rpc https://testnet.vextaris.io token balance <address> VXS
```

Signing (required for every write)

The network refuses unsigned transactions (since 2026-10-31). **Every command that changes chain state must be given your key file with `--key <path>`.** The CLI then builds the transaction locally, signs it with your Ed25519 key, and submits it with `vex_submitBundle`; your key never leaves your machine.

```
vexidus --rpc https://testnet.vextaris.io --key ~/.vexidus/my.key \
  token transfer --from Vx0YourAddress... --to Vx0Recipient... --token
VXS --amount 1000000000
```

- The key file holds the 64-character hex Ed25519 secret key. Keep it `chmod 600`.

- `--from` / `--sender` / `--owner` must be the address of that key.
- Without `--key`, a write command falls back to a server-built RPC method, which the public network refuses with `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`.
- The CLI signs with the **canonical** scheme by default. Legacy signatures are refused from block **33,000,000** -- if you have an older build of the CLI, update it before then. (`--signing-scheme legacy` exists only for local test chains.)
- Read commands (`info`, `list`, `balance`, `status`, ...) need no key.

Amounts are in **raw units** (VXS has 9 decimals: `1000000000` = 1 VXS) unless the flag's help says otherwise (`token create --supply` takes whole units).

Command Groups

Write commands below are shown without the global `--rpc` / `--key` flags for brevity -- add `--key <path>` to every one of them.

`wallet` -- Wallet Management

```
vexidus wallet generate                    # Generate a new Ed25519 keypair (testnet only; key is generated by the node)
vexidus wallet derive-address <pubkey>     # Derive the Vx0 address for a public key
vexidus wallet submit-bundle <hex>         # Submit a bundle you signed elsewhere
```

For keys you will hold real value with, generate them offline (SDK `WalletKeypair::generate()` or VexSpark) rather than with `wallet generate`.

`query` / `tx` -- Balances, Blocks, Receipts

```
vexidus query balance <address> --token VXS
vexidus query block-number
vexidus tx status <tx_hash>
```

`token` -- Token Operations (VSC-7)

```
vexidus token create --name "MyToken" --symbol MTK --decimals 9 --supply
1000000 --owner <addr>
vexidus token transfer --from <addr> --to <addr> --token VXS --amount
100000000000
vexidus token update-metadata --mint <mint> --sender <addr> --image
"https://..." --description "..."
vexidus token lock-metadata --mint <mint> --sender <addr>
vexidus token info <symbol_or_mint>
vexidus token list
vexidus token balance <address> <token>
vexidus token reputation <mint>
```

`nft` -- NFT Operations (VSC-21)

```
vexidus nft create-collection --name "My NFTs" --symbol MNFT --base-uri
"ipfs://..." \
  --max-supply 10000 --royalty-bps 500 --owner <addr>
vexidus nft mint --collection <addr> --token-id 1 --metadata-uri
"ipfs://..." --recipient <addr> --sender <addr>
vexidus nft transfer --collection <addr> --token-id 1 --to <addr> --
sender <addr>
vexidus nft burn --collection <addr> --token-id 1 --sender <addr>
vexidus nft info <collection> <token_id>
vexidus nft list-by-owner <owner>
vexidus nft collection <collection>
```

`multi-token` -- Multi-Token Operations (VSC-55)

```
vexidus multi-token create-collection --name "Game Items" --symbol GAME -
-base-uri "ipfs://..." \
  --royalty-bps 250 --owner <addr>
vexidus multi-token define-type --collection <addr> --type-id 1 --name
"Gold" \
  --metadata-uri "ipfs://..." --max-supply 0 --fungible --sender <addr>
vexidus multi-token mint --collection <addr> --type-id 1 --to <addr> --
amount 1000 --sender <addr>
vexidus multi-token transfer --collection <addr> --type-id 1 --to <addr>
--amount 50 --sender <addr>
vexidus multi-token balance --collection <addr> --type-id 1 --owner
<addr>
```

`pool` -- DEX Operations (VexiDEX)

```
vexidus pool create --token-a VXS --token-b <mint> --amount-a <raw> --
amount-b <raw> --lp-lock 0 --sender <addr>
vexidus pool add-liquidity --token-a VXS --token-b <mint> --amount-a
<raw> --amount-b <raw> --min-lp 0 --sender <addr>
vexidus pool remove-liquidity --token-a VXS --token-b <mint> --lp-amount
<raw> --min-a 0 --min-b 0 --sender <addr>
vexidus pool swap --from VXS --to <mint> --amount <raw> --min-out <raw> -
-sender <addr>
vexidus pool quote --from VXS --to <mint> --amount <raw>      # read-only
vexidus pool info <token_a> <token_b>
vexidus pool list
```

`presale` -- Presale Operations (VSC-LAUNCH)

```
vexidus presale create --token <mint> --allocation <raw> --price <raw> --
min-contrib <raw> --max-contrib <raw> \
  --soft-cap <raw> --hard-cap <raw> --start <unix> --end <unix> --lp-
percent 60 --lp-lock <secs> --owner <addr>
vexidus presale contribute --presale <addr> --amount <raw> --sender
<addr>
vexidus presale claim --presale <addr> --sender <addr>
vexidus presale finalize --presale <addr> --sender <addr>
vexidus presale refund --presale <addr> --sender <addr>
vexidus presale info <presale>
vexidus presale list
```

`bridge` -- Bridge Operations

```
vexidus bridge withdraw --token <mint> --amount <raw> --chain ethereum --
dest-address 0x... --sender <addr>
vexidus bridge status <chain> <tx_hash>
vexidus bridge history <address>
vexidus bridge fees [chain]
```

Inbound bridge deposits (`bridge deposit`) are **paused on testnet** (since 2026-10-31) while deposit attestation moves to an attester quorum.

`stake` -- Validator Staking and Delegation

```
vexidus stake stake --from <addr> --amount <raw> --pubkey
<validator_pubkey_hex>
vexidus stake unstake --from <addr> --amount <raw>
vexidus stake delegate --from <addr> --amount <raw> --pubkey
<validator_pubkey_hex>
vexidus stake undelegate --from <addr> --amount <raw> --pubkey
<validator_pubkey_hex>
vexidus stake set-commission --from <addr> --rate 500
vexidus stake unjail --from <addr>
vexidus stake set-metadata --from <addr> --name "My Node" --description
"..." --website "https://..." --avatar-url "https://..."
vexidus stake set-auto-compound --from <addr> --validator
<validator_addr> --enable
vexidus stake validator <address>
vexidus stake list
vexidus stake info
vexidus stake delegations <address>
```

Validator onboarding (`init` / `register` / `status` / `join`) lives in the `validator` group -- see the validator setup guide.

`intent` -- IntentVM

```
vexidus intent plan "stake 1000 VXS"       # execution plan + gas
estimate (read-only)
vexidus intent status <bundle_hash>
```

`vexidus intent submit` uses the server-built `vex_submitIntent` RPC and is **refused on the public network**. To execute an intent, parse it with `vex_parseIntent`, sign it with the SDK and submit it with `vex_submitBundle` -- see the IntentVM guide.

`upgrade` -- VexVisor Upgrade Governance

```
vexidus upgrade plan                                    # show the
scheduled upgrade (read-only)
vexidus upgrade vote --from <addr> --name "v0.2.0" --approve
vexidus upgrade prepare                                 # pre-download and
verify the upgrade binary
```

Scheduling and cancelling upgrades (`upgrade schedule` / `upgrade cancel`) are restricted governance actions; most users only need `plan`.

`governance` -- Protocol Governance (VSC-88)

```
vexidus governance create-proposal \
  --from <addr> \
  --type param \                    # param | treasury | text |
contract-safety | collection-safety
  --title "Reduce gas price" \
  --description "Lower from 10 to 5 nanoVXS" \
  --param-key gas_price \
  --param-value 5

vexidus governance vote --from <addr> --id 1 --approve     # omit --
approve to vote no
vexidus governance execute --from <addr> --id 1            # Execute an
approved proposal
vexidus governance cancel --from <addr> --id 1             # Cancel
(proposer only)
vexidus governance list                                    # List all
proposals
vexidus governance list --status voting                    # Filter by
status
vexidus governance show --id 1                             # Show
proposal details
```

`multisig` -- Multi-Sig Wallets (VSC-88)

```
vexidus multisig create \
  --from <addr> \
  --signers Vx0Addr1...,Vx0Addr2...,Vx0Addr3... \
  --threshold 2 \
  --label "Team Treasury"

vexidus multisig propose \
  --from <addr> \
  --account Vx5MultiSig... \
  --operations '[{"Transfer":
{"to":"Vx0...","token_symbol":"VXS","amount":50000000000}}]' \
  --description "Monthly budget"

vexidus multisig approve --from <addr> --account Vx5MultiSig... --tx-id 0
vexidus multisig revoke --from <addr> --account Vx5MultiSig... --tx-id 0
vexidus multisig show --account Vx5MultiSig...
vexidus multisig list-txs --account Vx5MultiSig...
```

Each signer approves with their own `--key`.

`dao` -- DAOs (VSC-99)

```
vexidus dao create --from <addr> --name "My DAO" --token <mint> ...   #
see `vexidus dao create --help`
vexidus dao vote --dao <addr> --id 1 --approve --from <addr>
vexidus dao show <dao>
```

`vip` -- Vextaris Intent Programs

```
vexidus vip register --program
'{"namespace":"myapp","name":"reward_and_stake",...}' --from <addr>
vexidus vip simulate myapp reward_and_stake --params
'[["amount","1000000000"]]'   # dry run
vexidus vip show myapp reward_and_stake
vexidus vip list --namespace myapp
vexidus vip verify <namespace> <key_hex> <hash>
# check an anchor
```

Executing a VIP and writing anchors are done with the SDK -- see the VIP guide.

`atomence` -- Cross-Chain Escrow Swaps

```
# Read-only queries
vexidus atomence list-settlements --token VXS --since-block 2900000 --
limit 50
vexidus atomence get-order Vxh<order-id>
vexidus atomence list-mms --tier 1
vexidus atomence stats

# Market makers: register or top up the VXS bond
vexidus --key keys/mm.key atomence register-mm \
  --from Vx0MarketMaker... \
  --stake 100000000000000

# Users: create an escrow order
vexidus --key keys/user.key atomence create-order \
  --from Vx0User... \
  --token VXS \
  --amount 1000000000 \
  --dest-chain solana \
  --dest-address <sol-address> \
  --max-spread-bps 50

# Users: cancel a pending order, or reclaim a matched order after its
settle deadline
vexidus --key keys/user.key atomence cancel \
  --from Vx0User... \
  --order-id Vxh<order-id>
```

Atomence write commands always require `--key` (there is no server-side fallback). A market maker's bond must total at least **1,000 VXS**; the tier derives from the total (1K / 10K / 100K VXS).

> ⚠ **NEW ORDER RULES FROM BLOCK 30,655,525**
>
> - **Orders state what the user receives:** the destination asset (`"native"` or the token contract / mint on the destination chain) and the minimum amount. Orders without them -- including orders created with the `create-order` command above -- cannot be matched after activation; cancel them while pending, or reclaim them once matched and past the settle deadline.
> - **Only registered market makers can match.**

- **One-hour settle deadline:** if a matched order is not settled in time, `atomence cancel` reclaims the full escrow plus 10% of the market maker's bond.
- `get-order` shows the order; the RPC `vex_getAtomenceOrder` also returns `dest_asset`, `min_dest_amount` and `settle_deadline`.

Global Options

Flag	Description
`-r, --rpc <URL>`	RPC endpoint URL (default: `http://localhost:9933`; public testnet: `https://testnet.vextaris.io`)
`-k, --key <PATH>`	Wallet secret key file (Ed25519, 64-char hex). Required for every write.
`--signing-scheme <legacy|canonical>`	Signature scheme for `--key` (default: `canonical`; also read from `$VEXIDUS_SIGNING_SCHEME`). Use the default on the public network.
`-h, --help`	Print help
`-V, --version`	Print version

IntentVM Developer Guide

IntentVM is Vextaris's declarative transaction engine. Instead of building low-level operations manually, developers describe **what they want** and the engine figures out the optimal execution path.

> ⚠ **INTENTS MUST BE SIGNED**
>
> The network refuses unsigned transactions (since 2026-10-31). `vex_submitIntent`, which builds and submits the intent on the node, is therefore **refused on the public network**. Every intent is now parsed (or built) on the client, **signed by the user**, and submitted with `vex_submitBundle` -- see Signed Submission. Signatures must use the canonical scheme; legacy signatures are refused from block 33,000,000.

Quick Start

Using the Rust SDK

```rust
use vexidus_sdk::{BundleBuilder, IntentBuilder, parse_intent};
use vexidus_types::primitives::{Address, Amount, Timestamp};

// Option 1: Fluent builder
let (goal, constraints) = IntentBuilder::new()
    .stake(Amount::from_vxs(1000), None)
    .with_slippage(1)
    .build()
    .unwrap();

// Option 2: Natural language
let parsed = parse_intent("stake 1000 VXS").unwrap();

// Sign and submit the goal (see Signed Submission below)
let bundle = BundleBuilder::new(&wallet.vx0_address())?
    .intent_with(goal, constraints, Some("stake 1000 VXS".into()), None)
    .nonce(nonce)
    .sign(&wallet);                          // canonical scheme
let tx_hash = client.submit_bundle(&bundle).await?;
```

Outcome-Based Intents (Builder API)

```rust
use vexidus_sdk::IntentBuilder;

// Acquire: "get me 10 SOL, max 200 USDC"
let (goal, constraints) = IntentBuilder::new()
    .acquire("SOL", 10.0)
    .with_max_cost("USDC", 200.0)
    .build()
    .unwrap();

// Liquidate: "sell everything for USDC, keep my VXS"
let (goal, constraints) = IntentBuilder::new()
    .liquidate("USDC", 500.0)
    .exclude(&["VXS"])
    .build()
    .unwrap();

// Rebalance portfolio
let (goal, constraints) = IntentBuilder::new()
    .rebalance(&[("VXS", 50), ("ETH", 30), ("USDC", 20)])
    .build()
    .unwrap();
```

Using RPC Directly

```
# Parse an intent into a structured, signable goal (executes nothing)
curl -s https://testnet.vextaris.io -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_parseIntent","params":["swap 100 VXS
for USDC"],"id":1}' | jq .

# Preview for wallet confirmation (human-readable)
curl -s https://testnet.vextaris.io \
  -d '{"jsonrpc":"2.0","method":"vex_previewIntent","params":["swap 100
USDC for VXS","SENDER_ADDRESS"],"id":1}' | jq .

# Simulate execution (detailed dry-run)
curl -s https://testnet.vextaris.io \
  -d '{"jsonrpc":"2.0","method":"vex_simulateIntent","params":["swap 100
USDC for VXS","SENDER_ADDRESS"],"id":1}' | jq .

# Get price quote
curl -s https://testnet.vextaris.io \
  -d '{"jsonrpc":"2.0","method":"vex_getPrice","params":
["USDC","VXS"],"id":1}' | jq .

# Check intent status
curl -s https://testnet.vextaris.io \
  -d '{"jsonrpc":"2.0","method":"vex_getIntentStatus","params":
["VxhBUNDLE_HASH"],"id":1}' | jq .
```

Wallet Integration (Confirm-Before-Submit)

Wallets should follow a two-step flow for intent submission:

Step 1: Preview -- Call `vex_previewIntent` to get a human-readable confirmation:

```
{
  "summary": "Swap 100 USDC -> ~1985.5 VXS (via USDC/VXS pool)",
  "route_type": "direct",
  "estimated_fee_vxs": "0.00042 VXS",
  "price_impact": "0.3%",
  "slippage_tolerance": "1%",
  "balance_sufficient": true,
  "warnings": []
}
```

Step 2: Sign and submit -- If the user approves, build the intent from the same text with `vex_parseIntent`, show the user the structured goal it resolved to, have the user sign it in the wallet, and submit it with `vex_submitBundle` (Signed Submission).

For auto-routed swaps, the preview shows the routing path:

```
{
  "summary": "Swap 100 DAI -> ~4.2 SOL (via DAI -> VXS -> SOL)",
  "route_type": "auto-routed (2 hops)",
  "price_impact": "1.2%",
  "warnings": ["Auto-routed through VXS. Slippage applied per leg."]
}
```

Signed Submission

The node never signs for the user. There are two signed paths:

TypeScript (`@vexidus/sdk`) -- free text via `vex_parseIntent`. The node parses the text and returns a structured goal (amounts as decimal strings, so no precision is lost); the client turns it into an operation, signs it, and submits it.

```
import { BundleBuilder, RpcClient, WalletKeypair,
         intentFromParsed, goalFromJson, intentMaxGas } from
'@vexidus/sdk';

const rpc = new RpcClient('https://testnet.vextaris.io');
const text = 'swap 100 VXS for USDC';

// 1. Parse (executes nothing)
const parsed = await rpc.parseIntent(text);

// 2. Show the user what the text resolved to -- never sign a parse
result unseen
console.log(parsed.goal);

// 3. Build, sign (canonical), submit
const sender = wallet.vx0Address();
const hex = new BundleBuilder(sender)
  .addOperation(intentFromParsed(parsed, { originalText: text }))
  .maxGas(BigInt(intentMaxGas(goalFromJson(parsed.goal))))
  .nonce(await rpc.getNonce(sender))
  .signAndSerialize(wallet, 'canonical');
const txHash = await rpc.submitBundleHex(hex);
```

`vex_parseIntent` returns signable goals for **swap, transfer, stake, add-liquidity and `.vex` name registration** phrasings, and combinations of them (for example `"swap 10 USDC and 10 USDT for VXS"`). Other patterns on this page (acquire, liquidate, rebalance, DCA, conditional, bridge, synthetic) return an error from `vex_parseIntent` that names the unsupported goal type.

Rust SDK -- any goal you can build. Build the goal with `IntentBuilder` (or the natural-language parser), add it with `BundleBuilder::intent_with(goal, constraints, original_text, source_parser)`, then `.sign(&wallet)` and `client.submit_bundle(...)`, as in the Quick Start. The Rust SDK signs canonical by default.

Wallets. VexSpark signs every intent in the wallet; users see the confirmation and sign there.

Supported Intent Patterns

Token Operations

Pattern	Example	Goal Type
Swap	`"swap 100 VXS for USDC"`	Swap
Swap with slippage	`"swap 50 ETH for VXS with 3% slippage"`	Swap + constraints
Buy	`"buy 100 VXS with USDC"`	Swap (reversed)
Sell	`"sell 50 ETH for USDC"`	Swap
Convert	`"convert 200 USDT to VXS"`	Swap
Trade	`"trade 100 SOL for VXS"`	Swap
Multi-swap	`"swap 10 USDC and 10 USDT for VXS"`	Composite (2 swaps)

Staking & Liquidity

Pattern	Example	Goal Type
Stake	`"stake 1000 VXS"`	Stake
Stake with validator	`"stake 500 VXS with validator Vx0abc..."`	Stake
Provide liquidity	`"add liquidity 100 VXS and 50 USDC"`	ProvideLiquidity

Transfers & Bridge

Pattern	Example	Goal Type
Transfer	`"send 100 VXS to Vx0abc..."`	Transfer
Transfer (VNS)	`"send 50 USDC to chris.vex"`	Transfer
Pay	`"pay 25 VXS to Vx0abc..."`	Transfer
Bridge	`"bridge 10 SOL from solana"`	Bridge
Bridge + Swap	`"bridge 10 SOL from solana and swap to VXS"`	Composite

Outcome-Based Intents

Specify **what you want**, not the steps. The planner reverse-engineers the execution.

Pattern	Example	Goal Type
Acquire	`"get me 10 SOL, max 200 USDC"`	Acquire
Acquire (no limit)	`"get me 5 ETH"`	Acquire
Acquire (I need)	`"I need 100 VXS, spending at most 50 USDC"`	Acquire
Liquidate	`"liquidate to 500 USDC"`	Liquidate
Liquidate (keep)	`"sell everything for USDC keeping VXS"`	Liquidate
Liquidate (exclude)	`"cash out to 1000 USDC excluding VXS and ETH"`	Liquidate
Rebalance	`"rebalance to 50% VXS, 30% USDC, 20% ETH"`	Rebalance

How Acquire works: The planner queries pool reserves, uses reverse AMM math to calculate the exact input amount needed to receive your target output, validates against your max cost constraint, and executes the swap. If the required cost exceeds your max, the intent is rejected before execution.

How Liquidate works: The planner queries your on-chain token holdings, excludes any tokens you want to keep, sorts remaining holdings by best swap rate to the target token, and greedily sells from best rate until your target amount is covered.

Portfolio Intelligence

Pattern	Example	Goal Type
Allocate	`"put 100 VXS split 60 stake 40 liquidity"`	Allocate
DCA	`"buy 10 VXS with USDC every day for 7 days"`	Recurring
DCA (hourly)	`"swap 5 USDC for VXS every hour for 24 hours"`	Recurring
DCA (weekly)	`"buy 100 VXS with USDC every week for 4 weeks"`	Recurring

Conditional & Limit Orders

Pattern	Example	Goal Type
Price condition	`"swap 100 USDC for VXS if price below 0.15"`	Conditional
Limit buy	`"limit buy 100 VXS at 0.12 USDC"`	Conditional
Price above	`"sell 50 VXS for USDC if price above 1.5"`	Conditional

Synthetic Assets (VSC-71)

Pattern	Example	Goal Type
Mint synthetic	`"buy 10 AAPL with USDC"`	MintSynthetic
Mint index	`"buy 5 SPY with USDC"`	MintSynthetic
Long perpetual	`"open 5x long ETH perp with 100 USDC"`	OpenPosition
Short perpetual	`"short BTC 3x with 200 USDC"`	OpenPosition

> **NOTE**
>
> VSC-71 synthetic assets are a **licensee interface**. IntentVM parses and accepts these intents, while oracle pricing and margin logic are supplied by the licensee's own integration. See the VSC-71 Integration Guide for how to implement.

VNS Registration

Pattern	Example	Goal Type
Register name	`"register chris.vex"`	Custom (VNS)
Register (no suffix)	`"register myname"`	Custom (VNS)

Inline Constraints

Constraints can be embedded directly in any intent string. The parser extracts them before processing the goal:

Constraint	Syntax	Example
Slippage	`"with N% slippage"`	`"swap 100 VXS for USDC with 2% slippage"`
Deadline	`"within N minutes"`, `"deadline N seconds"`	`"swap 100 VXS for USDC within 5 minutes"`
Max fee	`"max fee N"`, `"gas limit N"`, `"fee cap N"`	`"stake 1000 VXS max fee 50000"`

Multiple constraints can be combined in a single intent:

```
"swap 100 USDC for VXS with 2% slippage within 5 minutes max fee 100000"
```

Multi-Language Support

The NL parser accepts intents in 7 languages. Non-English input is normalized to English equivalents before parsing. All intent types work in all languages -- swaps, transfers, staking, outcome-based intents, and constraints.

Supported Verbs

Language	buy	sell	swap	send	stake
English	buy, get, purchase	sell, dump	swap, convert, trade	send, pay	stake, deposit
Spanish	comprar	vender	intercambiar, cambiar	enviar	apostar
Portuguese	comprar	venda	trocar	envie	--
French	acheter	vendre	échanger	envoyer	--
German	kaufen	verkaufen	tauschen	senden	--

Language	buy	sell	swap	send	stake
Italian	comprare	vendere	scambiare	inviare	--
Japanese	買う (kau)	売る (uru)	交換 (koukan)	送る (okuru)	--

Connector Words

Language	"with"	"for"	"to"	"every"
Spanish	con	por	para	cada
Portuguese	com	--	--	cada
French	avec	pour	--	--
German	mit	für	--	--
Italian	con	--	--	--

Examples by Language

```
# English
"buy 10 VXS with USDC"              → Acquire (reverse-quotes ~1 USDC
from pool)
"swap 100 USDC for VXS"             → Swap (100 USDC input)
"get me 5 SOL max 200 USDC"         → Acquire with cost cap

# Spanish
"comprar 10 VXS con USDC"           → Acquire
"intercambiar 100 USDC por VXS"     → Swap
"enviar 50 VXS para Vx0abc..."      → Transfer

# Portuguese
"trocar 100 USDC por VXS"           → Swap
"comprar 10 VXS com USDC"           → Acquire

# French
"acheter 10 VXS avec USDC"          → Acquire
"échanger 50 ETH pour VXS"          → Swap

# German
"kaufen 10 VXS mit USDC"            → Acquire
"tauschen 100 USDC für VXS"         → Swap

# Italian
"comprare 10 VXS con USDC"          → Acquire
"scambiare 100 USDC per VXS"        → Swap
```

TIP

"buy" vs "swap" semantics: "buy 10 VXS with USDC" produces an **Acquire** intent -- the 10 refers to the desired output, and the planner reverse-calculates the required USDC input from pool reserves. "swap 100 USDC for VXS" produces a **Swap** -- the 100 is the input amount. This distinction applies in all languages.

Adding a Language

The normalizer is a simple string replacement pipeline. To add a new language, map verbs and connectors to their English equivalents in the normalizer. Longer forms must be listed before shorter ones to prevent partial matching (e.g., Italian "comprare" before Spanish "comprar").

Known Token Symbols

Symbol	Token	Decimals	Mint Address
VXS	Native Vextaris	9	`Address::ZERO`
USDC	Bridged USDC	6	Blake3("ethereum_USDC")
USDT	Bridged USDT	6	Blake3("ethereum_USDT")
DAI	Bridged DAI	18	Blake3("ethereum_DAI")
SOL	Bridged SOL	9	Blake3("solana_SOL")
ETH/WETH	Bridged ETH	18	Blake3("ethereum_ETH")
BTC/WBTC	Bridged BTC	8	Blake3("ethereum_WBTC")
BNB	Bridged BNB	18	Blake3("bsc_BNB")

Dynamic decimal resolution: The NL parser resolves each token's decimals from the on-chain token registry at parse time. For example, `"swap 100 USDC for VXS"` correctly uses 6 decimals for USDC (100 × 10^6) and 9 decimals for VXS. Custom tokens created via VexForge are also resolved dynamically.

RPC Methods

Intent Submission & Query

Method	Params	Returns
`vex_parseIntent`	`[intent_string]`	Structured, signable goal + constraints (executes nothing)
`vex_submitBundle`	`[borsh_hex]`	Bundle hash -- submit the signed intent here
`vex_submitIntent`	`[intent_string, sender_address]`	Builds the intent on the node. **Refused on the public network** (unsigned)

Method	Params	Returns
`vex_getExecutionPlan`	`[intent_string]`	Execution steps, estimated gas, savings
`vex_getIntentStatus`	`[bundle_hash]`	Status (pending/completed/failed), gas used

Simulation & Preview

Method	Params	Returns
`vex_simulateIntent`	`[intent_string, sender_address]`	Dry-run result: operations, per-pool quotes, price impact (bps), balance checks, warnings
`vex_previewIntent`	`[intent_string, sender_address]`	Human-readable confirmation: summary text, fee in VXS, price impact %, route type, warnings

Price Queries

Method	Params	Returns
`vex_getPrice`	`[from_symbol, to_symbol]`	Spot price, TWAP, route type (direct or auto-routed)
`vex_quoteSwap`	`[from_token, to_token, amount_in]`	Swap quote with estimated output

Simulation Response Format

`vex_simulateIntent` returns detailed execution analysis:

```
{
  "success": true,
  "operations": [
    {
      "type": "Swap",
      "from_token": "USDC",
      "to_token": "VXS",
      "amount_in": "100000000",
      "estimated_out": "1985500000000",
      "price_impact_bps": 30
    }
  ],
  "estimated_gas": 42000,
  "total_price_impact_bps": 30,
  "warnings": [],
  "balance_check": {
    "sufficient": true,
    "required": "100000000",
    "available": "500000000"
  }
}
```


For auto-routed swaps, each leg is reported separately:


```
{
  "operations": [
    {
      "type": "Swap (Leg 1)",
      "from_token": "DAI",
      "to_token": "VXS",
      "price_impact_bps": 15
    },
    {
      "type": "Swap (Leg 2)",
      "from_token": "VXS",
      "to_token": "SOL",
      "price_impact_bps": 22
    }
  ],
  "total_price_impact_bps": 37,
  "warnings": ["Auto-routed through VXS. Slippage applied per leg."]
}
```

Price Impact & Warnings

The simulation and preview endpoints include a `warnings` array. Common warnings:

Warning	Meaning
"High price impact (>5%)"	Large order relative to pool size
"Auto-routed through VXS"	No direct pool, using intermediate token
"Price impact exceeds 25% -- execution will be rejected"	Order too large for available liquidity
"Insufficient balance"	Sender doesn't have enough tokens

Architecture

```
User Input (NL or JSON, any supported language)
    |
    v
Multi-Language Normalizer
    |  maps non-English verbs to English equivalents
    v
Constraint Extractor
    |  strips deadline, max_fee, slippage from text
    v
NL Parser / JSON Deserializer
    |  pattern matching → Goal enum
    v
IntentBuilder
    |  fluent API, builds Goal + Constraints
    v
RPC: vex_previewIntent (wallet confirmation)
    |  summary, fee, price impact, warnings
    v
RPC: vex_parseIntent  ->  user signs in the wallet / SDK
    |  structured goal, signed client-side
    v
RPC: vex_submitBundle
    |  signed intent bundle, routed to the leader
    v
Execution Engine
    |  decomposes Goal → native Operations
    |  auto-routing through intermediate pools
    |  price impact check (25% max per pool)
    |  constraint enforcement (deadline, fees, slippage)
    v
Block Execution (atomic — all succeed or all revert)
```

Future: LLM Integration

The natural-language parser handles common intent phrasings out of the box. For more complex or free-form intents, integrate an LLM:

```
User: "Buy VXS with half my USDC and stake it with the highest-APY
validator"
  -> LLM extracts: Composite([
       Swap { from: USDC, to: VXS, amount: 50% of balance },
       Stake { token: VXS, amount: all, validator: highest_apy }
     ])
```

The `Goal::Custom(String)` variant is reserved for LLM-processed intents. Integration path:

1. Client sends NL to LLM API (Grok, Claude, etc.)
2. LLM returns structured Goal JSON
3. Client shows the structured goal to the user, builds the intent operation with the SDK, has the user sign it, and submits it via `vex_submitBundle`

Related Documentation

- IntentVM Architecture -- Deep dive into IntentVM design and provenance
- VSC-71 Integration Guide -- Synthetic assets for licensees
- Token Standards -- VexForge token operations that IntentVM composes
- SDK Guide -- BundleBuilder, WalletClient and transaction signing
- RPC Reference -- Full endpoint documentation

Building with VIPs (Vextaris Intent Programs)

This guide shows how to build, register, and invoke VIPs on Vextaris. VIPs replace smart contracts with declarative composition templates -- no compilation, no bytecode, no VM.

> ⚠ **EVERY WRITE IS SIGNED**
>
> The network refuses unsigned transactions (since 2026-10-31). Registering a VIP, executing one, and anchoring data are all signed on the client -- with the CLI (`--key`), the SDK, or a wallet -- and submitted with `vex_submitBundle`. The server-built RPCs `vex_registerProgram`, `vex_executeProgram`, `vex_submitIntent` and `vex_anchor` are refused on the public network. Signatures must use the canonical scheme; legacy signatures are refused from block 33,000,000.

Quick Start

1. Write Your VIP

A VIP is a JSON document that describes what operations to execute and what parameters to accept:

```
{
  "namespace": "myapp",
  "name": "reward_and_stake",
  "version": 1,
  "author": "Vx0YourAddressHere",
  "description": "Mint reward tokens and auto-stake them",
  "params": [
    {
      "name": "reward_collection",
      "type": "Address",
      "required": true,
      "description": "Multi-token collection for rewards"
    },
    {
      "name": "amount",
      "type": "Amount",
      "required": true,
      "description": "Amount to mint and stake"
    },
    {
      "name": "pool",
      "type": "Address",
      "required": true,
      "description": "Staking pool address"
    }
  ],
  "steps": [
    {
      "id": "mint",
      "operation": "MintMultiToken",
      "args": [
        ["collection", "${params.reward_collection}"],
        ["type_id", "0"],
        ["to", "${sender}"],
        ["amount", "${params.amount}"]
      ]
    },
    {
      "id": "stake",
      "operation": "StakeToPool",
      "args": [
        ["pool", "${params.pool}"],
        ["amount", "${params.amount}"]
      ]
    }
```

```
  ],
  "returns": [
    ["mint_result", "${steps.mint.result.tx_hash}"],
    ["stake_result", "${steps.stake.result.tx_hash}"]
  ]
}
```

2. Simulate Before Registering

Use the CLI or RPC to dry-run your VIP:

```
vexidus vip simulate myapp reward_and_stake \
  --params '[["reward_collection","Vx0..."],["amount","1000000000"],
["pool","Vx0..."]]'
```

This shows you the expanded operations, estimated gas, and safety analysis -- without spending anything.

3. Register

```
vexidus --rpc https://testnet.vextaris.io --key ~/.vexidus/my.key vip
register \
  --program '{"namespace":"myapp","name":"reward_and_stake",...}' \
  --from Vx0YourAddress
```

The CLI signs the registration with your key and submits it. With `@vexidus/sdk`, use `BundleBuilder.registerProgram(jsonBytes)` and sign it the same way as the invocation below.

The chain runs static safety analysis. If it passes, your VIP is live. If it fails, you get detailed error messages explaining which rule was violated.

4. Users Invoke Your VIP

The user signs an execution of the program and submits it with `vex_submitBundle`. With `@vexidus/sdk`:

```
import { BundleBuilder, RpcClient } from '@vexidus/sdk';

const rpc = new RpcClient('https://testnet.vextaris.io');
const sender = wallet.vx0Address();

const hex = new BundleBuilder(sender)
  .executeProgram('myapp', 'reward_and_stake', [
    ['reward_collection', 'Vx0...'],
    ['amount', '1000000000'],
    ['pool', 'Vx0...'],
  ])                                    // optional 4th argument: version
(default: latest)
  .nonce(await rpc.getNonce(sender))
  .signAndSerialize(wallet, 'canonical');

const txHash = await rpc.submitBundleHex(hex);
```

Run `vexidus vip simulate` (or `vex_simulateProgram`) first and show the user what the program will do before they sign.

Parameter Types

Type	JSON Value	Example
`String`	`"hello"`	Names, descriptions, labels
`Address`	`"Vx0..."`	Vextaris addresses
`Amount`	`"1000000000"`	Raw amounts (scaled by token decimals)
`U8`	`8`	Small integers (0-255)
`U64`	`12345`	Large integers
`Bool`	`true`	Flags
`AddressArray`	`["Vx0...","Vx0..."]`	Lists of addresses (max 20)
`U64Array`	`[0, 1, 6, 7]`	Lists of integers (max 50)
`StringArray`	`["a","b"]`	Lists of strings (max 20)

Allowed Operations

These operations can be used in VIP steps:

Token Operations: Transfer, CreateToken, CreateNftCollection, MintNft, TransferNft, BurnNft, CreateMultiTokenCollection, DefineTokenType, MintMultiToken, BatchMintMultiToken, TransferMultiToken, BatchTransferMultiToken, BurnMultiToken, UpdateTokenMetadata, LockTokenMetadata

DeFi: CreatePool, AddLiquidity, Swap

Governance: CreateMultiSig, ProposeMultiSigTx, CreateProposal, CreateDAO

Staking: Stake, Delegate, CreateStakingPool, StakeToPool

Other: RegisterName, Anchor

Operations NOT Allowed in VIPs

Bridge operations (BridgeDeposit, BridgeWithdraw), key management (RemoveKey, RotateKey), upgrade operations (ScheduleUpgrade), stablecoin admin (StablePause, StableBlacklist), and Deploy/ContractCall are excluded for safety. Submit these as their own signed transactions, not through a VIP.

Loops

VIPs support iterating over array parameters:

```
{
  "id": "create_types",
  "operation": "DefineTokenType",
  "args": [
    ["collection", "${steps.collection.result.address}"],
    ["type_id", "${item}"],
    ["name", "Type ${item}"]
  ],
  "loop_over": "params.checkpoint_types"
}
```

If the user passes `checkpoint_types: [0, 1, 6, 7]`, this step executes 4 times, once for each value.

Limit: Arrays can have at most 50 elements. Total expanded operations across all steps cannot exceed 50.

Step References

Steps can reference results from prior steps:

```
{
  "id": "collection",
  "operation": "CreateNftCollection",
  "args": [["name", "My Collection"]]
},
{
  "id": "mint",
  "operation": "MintNft",
  "args": [
    ["collection", "${steps.collection.result.address}"],
    ["recipient", "${sender}"]
  ]
}
```

Rules:

- Steps can only reference steps that come BEFORE them
- Forward references are rejected at registration
- Self-references are rejected

Safety Rules

Your VIP must pass these rules to be registered:

1. **No hardcoded recipient addresses** in Transfer, MintNft, MintMultiToken, etc. Use `${sender}` or `${params.*}`.
2. **No hidden approvals** -- Approve spender must be `${params.*}` (user explicitly provides it).
3. **All operations must be in the allowed list** -- no bridge admin, no key management.

4. **Max 50 expanded operations** -- loops count against this.
5. **Max 5M estimated gas** -- complex programs get a warning.
6. **Valid template expressions** -- all `${params.X}` must reference declared parameters.
7. **Safe URIs** -- no XSS, no `javascript:`, no `data:` schemes.
8. **Unique step IDs** -- no duplicates, no empty IDs.
9. **Valid namespace** -- 3-32 chars, starts with a letter, alphanumeric + hyphens.

Data Anchoring

The Anchor operation lets you write a BLAKE3 hash to the chain without token semantics:

```
// Anchor a document hash (signed, submitted with vex_submitBundle)
const hex = new BundleBuilder(sender)
  .anchor(
    'myapp',                                          // namespace
    new TextEncoder().encode('invoice-2026-001'),  // key
    documentHash,                                     // 32-byte BLAKE3 hash
(Uint8Array)
    'ipfs://QmDocHash',                               // optional metadata
URI
  )
  .nonce(await rpc.getNonce(sender))
  .signAndSerialize(wallet, 'canonical');
await rpc.submitBundleHex(hex);
```

```
# Verify later (read-only)
vexidus vip verify myapp $(echo -n "invoice-2026-001" | xxd -p)
VxhABC123...
```

Gas cost: 5,000 (10x cheaper than token mints). Use this for document attestation, data integrity proofs, supply chain checkpoints, or any scenario where you need to prove "this data existed at this time."

Example: Supply Chain Shipment VIP

```
{
  "namespace": "logistics",
  "name": "create_tracked_shipment",
  "description": "Creates a shipment with NFT identity, checkpoint
collection, and custody multi-sig",
  "params": [
    { "name": "contents", "type": "String", "required": true,
"description": "Cargo description" },
    { "name": "parties", "type": "AddressArray", "required": true,
"description": "Custody chain participants" },
    { "name": "threshold", "type": "U8", "required": true, "description":
"Multi-sig approval threshold" },
    { "name": "checkpoint_types", "type": "U64Array", "required": false,
"default_value": "0,1,6,7", "description": "Checkpoint type IDs" }
  ],
  "steps": [
    {
      "id": "shipment_collection",
      "operation": "CreateNftCollection",
      "args": [
        ["name", "Ship-${params.contents}"],
        ["symbol", "SHIP"],
        ["base_uri", ""],
        ["max_supply", "1"],
        ["royalty_bps", "0"],
        ["description", "Tracked shipment"],
        ["owner", "${sender}"]
      ]
    },
    {
      "id": "checkpoint_collection",
      "operation": "CreateMultiTokenCollection",
      "args": [
        ["name", "Checkpoints-${params.contents}"],
        ["symbol", "CKP"],
        ["base_uri", ""],
        ["royalty_bps", "0"],
        ["description", "Shipment checkpoints"],
        ["owner", "${sender}"]
      ]
    },
    {
      "id": "define_types",
      "operation": "DefineTokenType",
      "args": [
```

```
        ["collection", "${steps.checkpoint_collection.result.address}"],
        ["type_id", "${item}"],
        ["name", "Checkpoint-${item}"],
        ["metadata_uri", ""],
        ["is_fungible", "true"]
      ],
      "loop_over": "params.checkpoint_types"
    },
    {
      "id": "custody",
      "operation": "CreateMultiSig",
      "args": [
        ["signers", "${params.parties}"],
        ["threshold", "${params.threshold}"],
        ["label", "Custody-${params.contents}"]
      ]
    }
  ],
  "returns": [
    ["shipment_collection",
"${steps.shipment_collection.result.address}"],
    ["checkpoint_collection",
"${steps.checkpoint_collection.result.address}"],
    ["custody_multisig", "${steps.custody.result.address}"]
  ]
}
```

One VIP invocation creates everything a logistics company needs to track a shipment: NFT identity, checkpoint event system, and multi-party custody chain. No smart contract. No compilation. No bytecode verification.

Namespace Ownership

- The first address to register a VIP in a namespace **owns** that namespace
- Only the owner can register new programs in their namespace
- Namespaces `vexidus`, `vex`, `foundation`, `system`, `core` are reserved for governance
- Namespaces must be 3-32 characters, alphanumeric with hyphens, starting with a letter

Versioning

VIPs are versioned automatically. Each registration of the same namespace/name increments the version. Users can invoke a specific version or default to latest:

```
// Invoke latest version
new BundleBuilder(sender).executeProgram('myapp', 'my_program', params)

// Invoke specific version
new BundleBuilder(sender).executeProgram('myapp', 'my_program', params,
2)
```

(Then `.nonce(...)`, `.signAndSerialize(wallet, 'canonical')` and `vex_submitBundle`, as above.)

Intent Studio (VexForge)

VIPs can be registered, browsed, simulated, and invoked through **Intent Studio** on VexForge -- a web UI for building and managing VIPs without using the CLI.

VSC-71 Synthetic Asset Integration Guide

VSC-71 defines the IntentVM interface for **tokenized stocks, commodities, indices, and perpetual contracts** on the Vextaris blockchain. This guide is for licensees and developers who want to implement synthetic asset functionality on their Vextaris-based deployment.

> **INFO**
>
> VSC-71 is a **licensee interface**. The Goal types, NL parsing, and SDK stubs are provided by the protocol. Oracle pricing, margin engines, collateral vaults, and liquidation logic are **your responsibility** to implement and integrate.

Overview

VSC-71 adds three Goal types to IntentVM:

Goal	Purpose	Example Intent
MintSynthetic	Create tokenized representation of a real-world asset	"buy 10 AAPL with USDC"
OpenPosition	Open a leveraged perpetual contract	"open 5x long ETH perp with 100 USDC"
ClosePosition	Close an existing leveraged position	"close position Vx0abc..."

These goals are **parsed and accepted** by IntentVM's NL parser. When a user submits an intent like "buy 10 AAPL with USDC", IntentVM correctly identifies it as a synthetic mint (not a crypto swap) and structures it as a `MintSynthetic` goal with the appropriate symbol, amount, and collateral token.

What the Protocol Provides

1. Natural Language Parsing

The IntentVM NL parser distinguishes synthetic assets from cryptocurrency tokens:

Recognized stock symbols: AAPL, TSLA, GOOGL, AMZN, MSFT, NVDA, META, NFLX, AMD, INTC, and more

Recognized index/commodity symbols: SPY, QQQ, DIA, GOLD, SILVER, OIL

```
"buy 10 AAPL with USDC"          → MintSynthetic { symbol: "AAPL",
amount: 10, collateral: USDC }
"buy 5 SPY with USDC"            → MintSynthetic { symbol: "SPY", amount:
5, collateral: USDC }
"open 5x long ETH perp with 100 USDC"  → OpenPosition { symbol: "ETH",
direction: Long, leverage: 5, ... }
"short BTC 3x with 200 USDC"     → OpenPosition { symbol: "BTC",
direction: Short, leverage: 3, ... }
```

2. Goal Type Definitions

```
MintSynthetic:
  - symbol: String (asset symbol, e.g. "AAPL", "GOLD")
  - amount: Amount (units to mint, scaled by synthetic's decimals)
  - collateral_token: Address (token used to back the synthetic)

OpenPosition:
  - symbol: String (underlying asset symbol)
  - direction: Long | Short
  - leverage: 1-100x
  - collateral: Amount
  - collateral_token: Address

ClosePosition:
  - position_id: Address (hash of the opening transaction)
```

3. SDK Stubs (IntentVM SDK)

The IntentVM SDK (available to all licensees) includes type definitions and parser support for VSC-71 goals. The SDK **does not** include execution logic -- it provides the interface for your implementation to plug into.

4. Constraint Support

All standard IntentVM constraints apply to VSC-71 goals:

- **Slippage** -- Maximum acceptable price deviation from oracle
- **Deadline** -- Execution must complete within time window
- **Max Fee** -- Gas cost cap

What You Must Implement

1. Oracle Integration

VSC-71 requires a price oracle for:

- **Synthetic minting** -- Fair value of the underlying asset (stocks, commodities, indices)
- **Perpetual contracts** -- Mark price for PnL calculation, funding rates
- **Liquidation triggers** -- Margin ratio monitoring

Requirements:

- Oracle must provide signed price attestations (Ed25519 or ECDSA)
- Price staleness check (reject prices older than your chosen threshold)
- Multi-oracle aggregation recommended for manipulation resistance

Common oracle partners:

- Chainlink (Cross-chain price feeds)
- Pyth Network (High-frequency, Solana-native)
- RedStone (Pull-based, gas efficient)
- Band Protocol (Cross-chain IBC)
- Custom API + attestation (for licensed data providers like Bloomberg, Refinitiv)

2. Collateral Vault

Synthetic minting requires over-collateralization:

```
User deposits 150 USDC collateral
Oracle reports AAPL = $195
Mint ratio: 150% (configurable)
User receives: floor(150 / 195 / 1.5) = 0.51 sAAPL
```

Your vault must handle:

- Collateral deposits and withdrawals
- Collateralization ratio enforcement (e.g. 150% minimum)
- Liquidation when ratio drops below threshold
- Collateral token whitelist (which tokens accepted as collateral)

3. Margin Engine (Perpetuals)

For `OpenPosition` and `ClosePosition`:

- **Initial margin** -- Minimum collateral for leverage level
- **Maintenance margin** -- Threshold for liquidation
- **Funding rates** -- Periodic payments between longs and shorts
- **PnL settlement** -- Mark-to-market on close
- **Maximum leverage** -- Cap per asset class (e.g. 20x crypto, 5x stocks)

4. Liquidation System

- Monitor collateralization ratios and margin levels
- Trigger liquidation when thresholds are breached
- Liquidation penalty distribution (protocol, liquidator, insurance fund)
- Insurance fund for socialized losses

Integration Architecture

```
User: "buy 10 AAPL with USDC"
    |
    v
IntentVM NL Parser (provided)
    |  Parses → MintSynthetic { symbol: "AAPL", amount: 10, collateral:
USDC }
    v
Your Execution Handler
    |  1. Query oracle for AAPL price
    |  2. Calculate required collateral (amount × price ×
collateral_ratio)
    |  3. Verify user has sufficient USDC
    |  4. Lock collateral in vault
    |  5. Mint sAAPL tokens to user
    v
On-Chain State Update
    |  Collateral locked, synthetic minted
    v
Event Emission
    |  SyntheticMinted { symbol, amount, collateral, price }
```

Hooking Into IntentVM Execution

When IntentVM encounters a `MintSynthetic`, `OpenPosition`, or `ClosePosition` goal, it delegates to the registered handler for VSC-71 operations. In the base protocol, this returns a "not yet active" response. Your implementation replaces this handler.

Integration points:

1. Register your VSC-71 execution handler with the state machine
2. Your handler receives the parsed Goal with all fields populated
3. Execute your oracle query, collateral logic, and minting/position management
4. Return success/failure events to IntentVM for atomic inclusion in the block

---

Recommended Token Structure

Synthetic Assets (sTokens)

Field	Value
Standard	VSC-7 (fungible token)
Symbol prefix	`s` (e.g. `sAAPL`, `sGOLD`, `sSPY`)
Decimals	8 (recommended for price precision)
Mintable	Yes (by vault contract only)
Burnable	Yes (on redemption)

Position NFTs (Optional)

For perpetual positions, consider using VSC-21 NFTs to represent open positions:

Field	Value
Standard	VSC-21 (NFT)
Metadata	Position details (entry price, leverage, direction, collateral)
Transferable	Optional (enables position trading)
Burnable	On close

Risk Considerations

Regulatory

- Tokenized stocks may be classified as securities in your jurisdiction
- Perpetual contracts face derivatives regulation
- Consult legal counsel before deploying to mainnet
- Implement KYC/AML gates as required by your jurisdiction

Technical

- Oracle manipulation (flash loan attacks on reference prices)
- Liquidation cascades during volatility
- Funding rate manipulation
- Stale price feeds during market closure (stocks)

Operational

- Market hours -- Stock synthetics should respect trading hours or clearly disclose off-hours pricing
- Corporate actions -- Stock splits, dividends require oracle updates and synthetic rebalancing
- Circuit breakers -- Halt minting during extreme volatility

Example: Minimal MintSynthetic Handler

This pseudocode shows the minimum viable implementation:

```
fn handle_mint_synthetic(goal: MintSynthetic, sender: Address, state:
&mut State) -> Result {
    // 1. Get oracle price
    let price = oracle.get_price(goal.symbol)?;
    ensure!(price.age_seconds < MAX_STALENESS, "Stale oracle price");

    // 2. Calculate required collateral
    let collateral_required = goal.amount * price.value *
COLLATERAL_RATIO / PRECISION;

    // 3. Check and lock collateral
    let balance = state.get_balance(sender, goal.collateral_token);
    ensure!(balance >= collateral_required, "Insufficient collateral");
    state.lock_collateral(sender, goal.collateral_token,
collateral_required)?;

    // 4. Mint synthetic token
    let synthetic_mint = state.get_or_create_synthetic(goal.symbol);
    state.mint(synthetic_mint, sender, goal.amount)?;

    // 5. Record position
    state.record_synthetic_position(sender, goal.symbol, goal.amount,
collateral_required, price.value)?;

    Ok(Event::SyntheticMinted { symbol: goal.symbol, amount: goal.amount
})
}
```

SDK Usage

The IntentVM SDK provides types for building VSC-71 intents programmatically:

```rust
// Using the IntentVM SDK (cross-chain adapter)
use intentvm_core::{IntentParser, Goal};

let parser = IntentParser::new(adapter);

// Parse NL → structured goal
let intent = parser.parse("buy 10 AAPL with USDC")?;
// intent.goal = Goal::MintSynthetic { symbol: "AAPL", amount: 10,
collateral: USDC }

// Or build programmatically
let intent = IntentBuilder::new()
    .mint_synthetic("AAPL", 10, usdc_address)
    .with_slippage(1)
    .build()?;
```

SUBMITTING ON VEXTARIS: SIGNED ONLY

Vextaris refuses unsigned transactions (since 2026-10-31), so a VSC-71 intent reaches the chain only as a transaction **signed by the user** and submitted with `vex_submitBundle` -- the server-built `vex_submitIntent` RPC is refused on the public network. Construct the goal in your client and add it to a bundle with the Vextaris Rust SDK's `BundleBuilder::intent_with(...)`, then sign it (canonical scheme; legacy signatures are refused from block 33,000,000). `vex_parseIntent` does not return signable synthetic goals -- it reports them as unsupported. See Signed Submission.

VexFi Native Implementation

Vextaris's own synthetic asset platform (VexFi) will implement VSC-71 with a native oracle network and margin engine. VexFi is separate from the licensee interface -- it uses the same VSC-71 Goal types but with Vextaris-operated infrastructure.

The SDK stubs remain available to all licensees regardless of VexFi's status. Licensees are free to implement their own oracle integrations independently.

Related Documentation

- IntentVM Architecture -- Core IntentVM design
- IntentVM Developer Guide -- NL patterns and RPC usage
- Token Standards -- VSC-7 (fungible) and VSC-21 (NFT) standards used by synthetics
- RPC Reference -- Full endpoint documentation

The Vextaris Ecosystem

A map of what we've built — and how it fits together.

Vextaris is its own Layer-1 blockchain (Ethereum-*compatible*, not Ethereum) where everything other chains deploy as smart contracts — tokens, NFTs, a DEX, staking, governance, identity — is built directly into the protocol as native operations. This page is the high-level tour of the ecosystem that sits on top of it.

The stack at a glance

The ecosystem layers from the protocol up to finished products:

```
  VERTICAL SOLUTIONS   Vexalus · VIMS · GlobalDEX · Vexcelon · VexFi
  APPLICATIONS         VexSpark · VexScan · VexiDEX · VNS · VexForge · DaimyoDAO
  PROGRAMMABILITY      VIP intent programs · IntentVM
  PROTOCOL STANDARDS   tokens · stablecoins · NFTs · DAOs · identity · cross-chain
  EXECUTION & STATE    native state machine · quantum-ready signatures
  CONSENSUS            HyperSync BFT · Vexcel attestation layer
```

The guiding idea: **native standards define what the chain can do; intent programs define what you build with it** — no Solidity, no bytecode, no compiler required.

Layer 1 — The protocol

HyperSync BFT consensus	Leader-based Byzantine Fault Tolerant consensus with fast, stake-weighted finality across validators on four continents.
Vexcel attestation layer	Validators prove liveness continuously, so the network keeps moving even when a leader is slow — no rollbacks, no stalls.

Native state machine	Operations execute directly as protocol primitives rather than as deployed contract bytecode.
Quantum-ready signatures	Dual Ed25519 + post-quantum signatures with built-in key rotation.
IntentVM *(patent-protected)*	Declare one intent, sign once, and the chain executes the many operations needed to fulfil it. "SQL for blockchain." See IntentVM.

Native standards — what the chain can do

All of these are native L1 capabilities, not contracts you deploy:

- **Tokens, stablecoins, NFTs, and multi-tokens** — first-class asset types. See Token Standards.
- **Staking pools, governance & multi-sig, DAOs, presales, reputation** — protocol-native economic and organizational primitives. See Governance.
- **VIDA — sovereign digital identity** — soulbound credentials, an issuer registry, W3C verifiable-credential export, and a `did:vex:` identity method. See VIDA.
- **VNS — .vex name service** — human-readable names that resolve to addresses. See VNS.
- **Anchor — universal data attestation** — commit and verify any data's provenance on-chain.
- **VexBridge** *(patent-protected)* and **Atomence** *(patent-protected)* — cross-chain verification and escrowed cross-chain swaps. Atomence settlements are proven by an attester quorum, and every order has a guaranteed refund path. Adopted mainnet principle: *nothing comes to Vextaris that can't leave* — no unbacked bridged tokens (planned for mainnet). See VexBridge and Atomence.

Programmability — what you build with it

VIP — Vextaris Intent Programs	Declarative JSON templates that compose protocol operations into reusable programs, with a built-in safety engine and governance registry. The smart-contract equivalent for custom application logic. See the VIP guide.

IntentVM	The engine those programs run through — one intent, one signature, many operations. See the IntentVM guide.

Applications — live products

Product	What it is
VexSpark	Multichain non-custodial wallet (Vextaris L1 + leading EVM chains + Solana). Send, receive, stake, manage tokens & NFTs, and carry your VIDA identity. Signs every transaction in the wallet — transfers, names, intents, token creation, NFTs and staking.
VexScan	Block explorer — transactions, accounts, tokens, validators, and pools.
VexiDEX	Native automated-market-maker DEX — pools, swaps, and liquidity.
VNS	Register and manage .vex names.
VexForge	Visual studio for composing and testing VIP intent programs.
DaimyoDAO	Create and govern DAOs on-chain.
VexSwap	Testnet faucet.

Vertical solutions — the ecosystem in use

Solution	What it demonstrates
Vexalus	Enterprise supply-chain tracking — custody, checkpoints, and document attestation — in a live logistics pilot. See Vexalus.
VIMS	A long-horizon multi-currency monetary system (Earth/Moon/Mars units) with an algorithmic monetary governor.
GlobalDEX	A multi-network cross-chain exchange built on Atomence and IntentVM.
Vexcelon	App-specific zones that run custom execution environments and settle to Vextaris L1.

Solution	What it demonstrates
VexFi	A DeFi suite (lending, synthetics, perpetuals) on the protocol roadmap. See VexFi.

Tooling & intelligence

- **VexVisor** — on-chain binary-upgrade governance with checksums and timelocks. See VexVisor.
- **VexAI** — AI-powered chain diagnostics and a knowledge assistant.
- **SDK, CLI & Docs** — a JavaScript SDK, a full command-line client, and the RPC reference.
- **Veridia** — an autonomous ecosystem agent active across social platforms.

What makes Vextaris different

1. **IntentVM** *(patent-protected)* — declare what you want; the chain figures out how.
2. **VIDA** — sovereign digital identity built into the protocol, not bolted on.
3. **Atomence** *(patent-protected)* — escrowed cross-chain swaps without wrapped tokens: registered market makers, attester-verified settlement, and a guaranteed refund if the market maker does not settle.
4. **VIP programs** — composable templates that replace smart contracts for the vast majority of use cases.
5. **Native standards** — tokens, NFTs, DEX, staking, governance, and identity as protocol operations, not deployed bytecode.

Vextaris L1 · live across seven validators on four continents · explore it at vexscan.io.

Vextaris Testnet Incentive Program

> **Status:** Program structure finalized. The reward pool is **48.54M VXS** (3% of total supply) with allocations planned across all three tracks. Specific per-track amounts will be published before mainnet.
>
> **Why no dollar amounts?** We believe in transparency. VXS does not have a market price yet, and we refuse to anchor rewards to speculative valuations. Concrete figures will be published before mainnet, once there is real price discovery. Early participants who join based on the technology — not a dollar promise — are exactly the community members we want.
>
> **All rewards vest.** 12-24 month linear vesting with no immediate unlocks at TGE. Milestone-based release where possible — if a project goes inactive, unvested tokens return to the pool.

Overview

The Vextaris Testnet Incentive Program rewards three types of participants who help build, secure, and grow the Vextaris ecosystem during the testnet phase:

Track	Who	What We're Looking For
Validators	Node operators	Reliable infrastructure, community building, uptime
Builders	Developers & teams	Applications, tools, and services deployed on Vextaris
Community	Delegators & active users	Staking participation, governance engagement

A dedicated pool of **48.54M VXS** (3% of total supply) funds all three tracks plus the **Intent-a-thon** builder event and, once it opens, a **bug bounty** programme (not open yet; see below). All rewards vest 12-24 months linear to align long-term incentives.

Track 1: Validator Rewards

Goal: 20-25 validators across multiple continents, stress-testing Vexcel attestation, Dragonfly Stream, and VexVisor upgrades at scale.

Qualification

- Run a Vextaris validator node for **minimum 30 consecutive days**
- Maintain **B grade (75+) reputation score** throughout
- Keep VexVisor auto-upgrade enabled (accept 90%+ of upgrades within 24 hours)
- Stake the minimum 1,000 VXS (available via testnet faucet)

Reward Factors

Factor	How It Works
Base allocation	Equal share across all qualifying validators each month
Reputation grade	B = 1.0x, B+ = 1.1x, A = 1.25x, A+ = 1.5x multiplier
Tenure streak	30d = 1.0x, 60d = 1.1x, 90d = 1.2x, 180d = 1.4x multiplier
Community builder	5+ delegators = +10%, 15+ = +25%, 30+ = +40% bonus

What This Rewards

- **Consistency over intensity** — 90 days at A grade beats 7 days of perfection
- **Community building** — validators who recruit delegators and grow their pool earn more
- **Upgrade compliance** — testnet requires everyone on the same binary; auto-upgrade is mandatory

Getting Started

1. Follow the Validator Setup Guide
2. Fund your validator via the testnet faucet
3. Stake and begin producing blocks
4. Your reputation score builds automatically — check it on VexScan

---

Track 2: Builder Grants

Goal: Seed the Vextaris application ecosystem with original, innovative projects across every industry.

Our Philosophy: Innovation Over Conversion

We're not looking for Ethereum dApps with a logo swap. We're looking for builders who see what Vextaris makes possible and think: *"I couldn't build this anywhere else."*

Vextaris is a general-purpose L1 blockchain — there is room for everyone. Finance, gaming, logistics, social, healthcare, education, creative arts. If you believe in yourself and you believe in building on a chain that was designed from the ground up to be different, there is a place for you here.

Who will build the next pump.fun — but with anti-rug protections baked into the protocol? Who will create the go-to DEX or marketplace on Vextaris? Who will be the first to use IntentVM for supply chain attestation, or VexBridge to unify assets across five chains with one signature? These are open questions, and the answers will come from builders, not from us.

We reward innovation, not imitation.

What You Can Build With

Vextaris ships production-ready infrastructure that most L1s only put on a roadmap:

Primitive	What It Enables
VSC-7 tokens	Launch tokens with built-in metadata, reputation scoring, and automatic bridge compatibility
VSC-21 NFTs + VSC-55 multi-token	Collections with on-chain metadata and cross-chain provenance
VexBridge	Vaultless cross-chain bridge. Assets from Solana, Ethereum, BSC and more become native tokens on Vextaris — not wrapped IOUs
VexiDEX AMM	On-chain liquidity pools and swap routing, ready to build on top of
IntentVM	1 intent, 1 signature, N operations across multiple chains. Natural language and structured input

Primitive	What It Enables
VSC-88 Governance	On-chain proposals, voting, multi-sig accounts — build DAOs and community-governed projects
VSC-LAUNCH	Presale and escrow with anti-rug protections built into the protocol layer
Vexcel DAG	Attestation layer for provable data — receipts, certifications, audit trails
Dragonfly Stream	Mempoolless transaction pipeline — zero MEV, zero front-running, by design

Grant Categories

We're open to any project that builds genuine value on Vextaris. Here are some areas we're particularly excited about:

Category	Ideas That Excite Us
DeFi & Launchpads	Token launchpads with VSC-LAUNCH anti-rug protections. DEX aggregators. Lending and borrowing. Yield strategies. Payment rails. The DeFi stack is wide open.
Gaming & Entertainment	On-chain games with VSC-21 NFT assets. Gaming economies with VSC-55 multi-token inventories. Prediction markets.
Real-World Applications	Supply chain tracking with Vexcel attestations. Credential verification. Document notarization. IoT data logging. Logistics management.
Social & Community	On-chain social platforms. Creator economies. Tipping and micropayments. DAO tooling for communities.
Cross-Chain & Intent	Applications that showcase IntentVM's "1 intent, N chains" capability. Multi-chain portfolio management. Bridge-powered remittance.
Infrastructure & Tools	Alternative explorers. Analytics dashboards. SDKs in new languages (Python, Go, TypeScript). Testing frameworks. Wallet integrations. API wrappers.
Something We Haven't Thought Of	The best projects are often the ones nobody predicted. If you have an idea that doesn't fit a category, that might be exactly what we're looking for.

How It Works

1. **Build first, ask later** — You don't need permission to start building on Vextaris testnet. Get VXS from the faucet, use the SDK and CLI, and deploy
2. **Show, don't tell** — Deploy a working prototype on testnet. Transactions on-chain are worth more than a slide deck
3. **Apply for a grant** — Once you have something working, submit a brief proposal: what you built, what traction looks like, and where you're headed
4. **Grow with us** — Top projects get ongoing support, promotion, and deeper integration with the Vextaris ecosystem

Evaluation Criteria

- **Does it work?** Deployed and functional on testnet. Code over promises
- **Is it original?** We reward projects that leverage what makes Vextaris unique — not ports of existing projects from other chains
- **Does it attract users?** Real on-chain activity signals genuine value
- **Is the builder committed?** Continued development, responsiveness, and vision for the future
- **Does it strengthen the ecosystem?** Projects that make Vextaris better for everyone else earn extra consideration

Grant Tiers

Tier	For	What You Get
Seed	Solo developers, early prototypes, experiments	VXS allocation + testnet support
Growth	Small teams, working products with real users	Larger VXS allocation + milestone-based funding + ecosystem promotion
Ecosystem	Established teams, high-impact applications	Largest allocations + partnership-level engagement + co-marketing

Exact VXS amounts per tier will be published when the reward pool is finalized. What matters now is building.

Track 3: Community Rewards

Goal: Reward non-technical participants who contribute to network health through staking and governance.

Delegator Rewards

- **Minimum:** Delegate 500 VXS to any validator for 14+ consecutive days
- **Reward:** Proportional to `amount_delegated × days_active`
- **Cap:** No single address earns more than 2% of the delegator pool (prevents whale domination)
- **Bonus:** Delegating to validators outside the top 5 by stake earns a 1.2x multiplier (encourages decentralization)

Governance Participation

- Vote on VSC-88 proposals when available
- Participate in governance discussions
- No direct VXS reward for voting (prevents vote farming) — but governance activity factors into future consideration for builder grants and validator reputation

---

Intent-a-thon (Builder Event)

Status: Ongoing (exact dates TBA)

The Intent-a-thon is a time-limited builder event focused on IntentVM and VIP (Vextaris Intent Programs). Build applications that showcase what natural-language intents and declarative on-chain programs can do.

What We're Looking For

- **IntentVM integrations** -- Applications that use "swap 100 VXS for ETH" style natural-language intents
- **VIP programs** -- Declarative templates that compose multiple operations into reusable on-chain programs
- **Cross-chain intent flows** -- Multi-chain operations powered by VexBridge + IntentVM

- **Novel use cases** -- Supply chain, gaming, social, DeFi, identity, or anything we haven't imagined

Prize Tiers

Tier	For	Criteria
Grand Prize	Best overall project	Innovation, execution, real-world utility
Runner Up	Strong submissions	Working prototype with clear vision
Honorable Mention	Promising early-stage	Good concept, needs more development

Exact VXS amounts per tier will be published when the event launches. All prizes vest 12 months linear.

How to Participate

1. Build on Vextaris testnet using the IntentVM Guide and VIP Guide
2. Deploy a working prototype before the submission deadline
3. Submit via the Intent-a-thon portal (link TBA)
4. Present to judges during demo day

Bug Bounty Program

> **The Vextaris bug bounty has not launched.** There is no active programme, no reward values have been set, and nothing on this page is an offer of payment.

Security reports are still welcome. They are triaged and recorded, and if a programme opens, earlier reports are revisited for eligibility, with credit to whoever reported first. How to report, and what is already known, is on Security: known issues.

Intended shape, once it opens

Published for transparency. It is not a commitment, and no rewards are payable today.

Severity	What would qualify
Critical	Consensus break, fund theft, validator compromise
High	State corruption, denial of service, bridge exploit
Medium	RPC crashes, incorrect balances, explorer data errors
Low	UI bugs, documentation errors, minor inconsistencies

- **Responsible disclosure required** -- report privately before publishing
- **No self-inflicted damage** -- do not exploit vulnerabilities on testnet to cause disruption

---

Anti-Gaming Rules

We want genuine participants, not farmers.

- **Snapshots:** Qualification measured via random-interval snapshots averaged over the period — not cumulative totals
- **Sybil resistance:** Validator rewards tied to unique pubkeys with real stake. Delegator rewards require minimum amounts and duration
- **Self-delegation detection:** Validators delegating to themselves from multiple addresses does not count toward the community builder bonus
- **Upgrade compliance:** Validators who repeatedly reject VexVisor upgrades or run outdated binaries forfeit rewards for that period
- **Good faith clause:** The Vexidus Foundation reserves the right to disqualify participants who demonstrably game the system. We'd rather have 15 genuine validators than 25 with 10 gaming it

---

Timeline

Phase	When	What Happens
Program announced	Now	Structure and rules published. Start earning immediately.
Reward pool confirmed	Done	48.54M VXS (3% of supply) allocated. Per-track amounts published before mainnet.
Builder grants open	Ongoing	Apply anytime. First grants awarded on rolling basis
Intent-a-thon	Ongoing (dates TBA)	Time-limited builder event for IntentVM and VIP projects
Bug bounty	Not yet open	No programme and no reward values yet. Reports are welcome and are revisited if a programme opens
Mainnet snapshot	Before mainnet launch	Final qualification snapshot taken
Rewards distributed	At mainnet launch	VXS distributed with 12-24 month linear vesting

---

FAQ

Q: How do I know my rewards are being tracked? Your validator reputation, blocks produced, delegation history, and on-chain activity are all recorded in Vextaris state. Nothing to submit — we read it from the chain. Check your status at vexscan.io.

Q: What if I start validating late? The program rewards tenure, so earlier is better. But a validator who joins on day 60 and runs flawlessly for 120 days still earns well. Quality matters more than when you started.

Q: Can I run multiple validators? Each validator must have a unique public key and genuine infrastructure. Running 5 validators on the same machine with the same IP defeats the purpose. We're testing geographic distribution and real-world network conditions.

Q: What about testnet VXS — does it carry over? No. Testnet VXS has zero monetary value. Mainnet rewards are a separate allocation distributed at launch based on testnet participation records.

Q: I'm a developer — where do I start?

1. Read the docs
2. Get testnet VXS from the faucet
3. Use the CLI or SDK to deploy
4. Submit a builder grant application when you have something working

---

Contact

- **Validator support:** docs.vextaris.io
- **Builder grants:** Apply via the form on vextaris.com (coming soon)
- **General questions:** Community channels (coming soon)

VexFi — Financial Infrastructure

VexFi is the protocol-level financial infrastructure layer of the Vextaris ecosystem. It unifies payments, trading, staking, custody, and natural-language portfolio management into a single interface, built entirely on Vextaris L1 primitives — no wrapped assets, no off-chain custody, no T+2 settlement.

VexFi is not a separate protocol. It is a purpose-built application layer that exposes the full power of IntentVM, VexiDEX, VSC-88 multi-sig, Dragonfly Stream, and the VNS name service through a clean API and SDK — making complex financial operations accessible with a single call.

What VexFi Provides

Capability	How It Works
Payments	Send VXS or any VSC token to a Vextaris address or `.vex` name. VNS resolution is automatic.
Trading	Spot swaps via VexiDEX AMM pools with TWAP-based price feeds.
Cross-chain settlement	Swap out to Solana, EVM chains and Bitcoin via Atomence escrow swaps: you set the destination asset and the minimum you receive, a bonded market maker pays you, and if it does not settle within 1 hour you reclaim your escrow plus 10% of its bond. 0.30% fee (0.15% market maker, 0.15% protocol treasury).
Staking and delegation	Stake VXS, delegate to validators, track rewards, and compound — all in one operation.
Multi-sig custody vaults	Create M-of-N vaults backed by VSC-88. Treasury management, team wallets, and enterprise custody.
Natural language intents	Express financial operations in plain language via IntentVM. One intent, one signature, atomic execution.
Synthetic assets	VSC-71 synthetic equities and perpetuals (planned) — oracle-priced, collateral-backed, self-custodial.

Capability	How It Works
Portfolio management	Rebalance, DCA, conditional orders, and allocation targets expressed as intents.

SDK Overview

The VexFi SDK provides a typed interface for all financial operations. Install via npm:

```
npm install @vexidus/vexfi-sdk
```

Core Methods

```
import { VexFi } from '@vexidus/vexfi-sdk';

const vexfi = new VexFi({ rpc: 'https://api.vexfi.com' });

// Send payment — resolves .vex names automatically
await vexfi.pay({
  to: 'alice.vex',          // or a Vx address
  amount: '100',
  token: 'VXS',
});

// Get portfolio snapshot
const portfolio = await vexfi.portfolio({ address: 'Vx...' });

// Get a swap quote
const quote = await vexfi.quote({
  from: 'VXS',
  to: 'USDC',
  amount: '500',
});

// Execute a swap
await vexfi.swap({
  from: 'VXS',
  to: 'USDC',
  amount: '500',
  maxSlippage: 0.5,         // percent
});

// Settle cross-chain (Atomence)
await vexfi.settle({
  chain: 'solana',
  amount: '1000',
  token: 'VXS',
});

// Stake VXS
await vexfi.stake({
  validator: 'Vx...',
  amount: '10000',
});

// Delegate to a validator
await vexfi.delegate({
  validator: 'Vx...',
```

```
  amount: '5000',
});

// Create a multi-sig custody vault
await vexfi.createVault({
  signers: ['Vx...', 'Vx...', 'Vx...'],
  threshold: 2,              // M-of-N
  label: 'Team Treasury',
});

// Submit a natural language intent
await vexfi.intent({
  text: 'Rebalance my portfolio to 60% staked VXS and 40% USDC',
});
```

REST API

Base URL: `https://api.vexfi.com`

Endpoint	Method	Description
`/v1/pay`	POST	Send a payment (VNS-aware)
`/v1/portfolio`	GET	Portfolio snapshot for an address
`/v1/market`	GET	Market data — prices, volume, liquidity
`/v1/quote`	GET	Swap quote with price impact
`/v1/swap`	POST	Execute a spot swap
`/v1/stake`	POST	Stake or delegate VXS
`/v1/custody`	POST/GET	Create or query multi-sig vaults
`/v1/intent`	POST	Submit a natural language intent
`/v1/prices`	GET	Current and historical token prices

Example: Get a swap quote

```
curl "https://api.vexfi.com/v1/quote?from=VXS&to=USDC&amount=500"
```

```
{
  "from": "VXS",
  "to": "USDC",
  "inputAmount": "500",
  "outputAmount": "142.87",
  "priceImpact": "0.12%",
  "fee": "0.0003",
  "route": ["VXS/USDC"]
}
```

Real-Time Price Feed

VexFi exposes a WebSocket endpoint for real-time price updates:

```
wss://api.vexfi.com/v1/ws
```

```
const ws = new WebSocket('wss://api.vexfi.com/v1/ws');

ws.send(JSON.stringify({
  type: 'subscribe',
  pairs: ['VXS/USDC', 'VXS/USDT', 'BTC/VXS'],
}));

ws.onmessage = (event) => {
  const { pair, price, volume24h, timestamp } = JSON.parse(event.data);
  console.log(`${pair}: $${price}`);
};
```

Price data is TWAP-sourced from VexiDEX pools with external oracle integration planned for real-world asset feeds (equities, commodities, forex).

Synthetics and Perpetuals (Planned)

VexFi will surface VSC-71 synthetic assets — tokenized equities, indices, and perpetual contracts — collateral-backed and oracle-priced, with self-custodial margin. Intent examples:

```
"Buy 10 AAPL with USDC"
"Open 5x long ETH perp with 100 USDC"
"DCA $200 weekly into my target allocation"
```

These are expressed as structured intents and executed atomically on-chain. No brokerage account. No counterparty custody. Instant settlement.

Key Links

Resource	URL
VexFi App	vexfi.com
API	api.vexfi.com
Wallet (VexSpark)	wallet.vexspark.com
IntentVM Guide	IntentVM Architecture
DEX (VexiDEX)	vexidex.com

Vexalus — Supply Chain Intelligence

Vexalus is a verifiable supply chain platform built on Vextaris L1 primitives. It provides tamper-evident shipment tracking, multi-party custody handoffs, document attestation, invoice tokenization, and a six-layer **Accountability Web** that makes lying about what shipped economically irrational.

Every Vexalus operation maps directly to a native Vextaris primitive. There is no new trust model to audit — the security guarantees are inherited from the underlying chain.

Surface	Where
Marketing site	vexalus.com
REST API	api.vexalus.com
ROI simulator	vexalus.com/simulation
Contact	vexalus.com/contact

How Existing Primitives Map to Supply Chain Operations

Supply Chain Concept	Vextaris Primitive	What It Provides
Shipment identity	VSC-21 NFT	Unique, non-fungible token representing a physical shipment — ownership = custodianship
Checkpoint events	VSC-55 multi-token mint	Scoped tokens (temperature, location, condition, environmental events) minted at each checkpoint — immutable log
Custody transfer	VSC-88 multi-sig proposal + execution	M-of-N approval for handoff between carrier, customs, and recipient

Supply Chain Concept	Vextaris Primitive	What It Provides
Document attestation	BLAKE3 hash anchored on-chain	Bill of lading, certificates of origin, inspection reports — hash stored, content off-chain
Invoice tokenization	VSC-7 fungible token	Tradeable receivable backed by on-chain shipment proof
Independent inspection	VIDA `custom:supply_chain_inspector` credential	Tier 4 issuer credential — inspectors sign evidence packages, revocation makes collusion irrational
Compliance audit	Explorer RPC queries	Full immutable history of every checkpoint, handoff, and attestation

---

The Accountability Web

The Accountability Web is six mutually-reinforcing verification layers, plus an independent inspector signature (Layer 7), that combine into a single dispute evidence package. A reviewer with the package can detect any post-hoc tampering with any layer, and the cost of faking all layers exceeds the value of the goods.

Layer	What it proves	Mechanism
1. Unit scanning	Every unit was actually scanned at pack-out	Per-unit barcode / RFID scan logged to the manifest
2. Manifest hash	The unit list hasn't changed since pack-out	BLAKE3 hash of the canonical manifest anchored on-chain
3. Weight continuity	The shipment's weight didn't change between checkpoints	Per-checkpoint weight observations with PASS / minor / major severity bands and a configurable tolerance
4. Seal chain	The container wasn't opened in transit	Sequence of seal attestations from packing through every handoff to unpacking
5. Photo / video evidence	Specific moments (packing, sealing, loading) were captured	Off-chain encrypted media with on-chain BLAKE3 anchor; client-side decrypt by authorized recipients

Layer	What it proves	Mechanism
6. Custody handoffs	Every party that held the shipment is recorded	Custody chain entries with from / to / timestamp / location
7. Independent inspector	A neutral third party verified the package	VIDA-credentialed inspector signs the dispute rollup hash (Ed25519); revocation invalidates past signatures

A **dispute evidence package** combines all seven layers into a single rollup BLAKE3 hash anchored on-chain. A verifier:

1. Re-computes the per-layer hashes from the bundle's payloads
2. Re-computes the rollup hash
3. Checks it against the on-chain anchor
4. Verifies the inspector's signature against their current VIDA credential

Any mismatch at any step exposes tampering. The bundle becomes legally defensible evidence in a fraction of the 60-120 days a traditional dispute resolution would take.

Environmental Envelope

The environmental envelope aggregates IoT checkpoint events (temperature, humidity, radiation, pressure, vibration, shock, atmospheric pressure, security events, power events) into a per-vertical PASS / FAIL verdict.

Six standard profiles ship with thresholds backed by public standards:

Profile	Backed by	Highlights
`pharma_cold_chain`	USP <659>, WHO 2-8°C vaccine cold chain	Tight 2-8°C band, 0.5 °C·h excursion budget, 30-min sustained cap, zero refrigeration outages
`pharma_frozen`	Frozen biologics	–25 to –15°C band, any thaw is fail
`electronics`	IEC 60068	Wider band, 30g shock tolerance
`hazmat`	DOT 49 CFR class 2/3	Pressurized container thresholds, zero tamper events

Profile	Backed by	Highlights
`perishable_food`	IATA Perishable Cargo Regulations	0-7°C with 3 °C·h budget, 85-95% humidity
`general`	Wide bands	Tamper + power are the active checks

The aggregator uses linear-interpolation area-under-curve integration for numeric dimensions -- so a 1°C overshoot for 10 minutes counts as approximately 0.17 °C·h against the budget, not a flat fail. Severity ladder: `pass` (within band) → `minor` (excursion under budget) → `major` (budget exhausted or sustained-time exceeded) → `fail` (auto-fail subtype fired, e.g. tamper / refrigeration outage / no observations for a required dimension).

Verdicts are anchored on-chain so an auditor can confirm the verdict that was claimed at the time. This eliminates the "we'll get to it later" pattern that turns into "we lost the records" months after the fact.

Store-and-Forward IoT

Real supply chains include ocean cargo, rural distribution, and (eventually) lunar manifests -- environments where the IoT device cannot reach the network reliably. The store-and-forward agent:

- Persists every checkpoint to a local SQLite queue **before** any network attempt
- Drains the queue when the link returns, with per-event exponential backoff (5s base × 2^retry, capped at 1 hour)
- Detects connectivity via an optional probe so failed batches don't hammer a downed upstream
- Recovers in-flight events on process restart so a crashed gateway doesn't lose telemetry
- Uses UUID-based idempotency so a reconnect-after-blip is safe to retry without double-submitting

The API endpoint accepts a client-supplied UUID per event and dedups against a server-side mapping table. Reconnects are safe by construction.

Three-Tier Privacy

Vexalus operations encrypt shipment data into three tiers:

Tier	Audience	What they see
`public`	Anyone	Existence of the shipment, anchor hashes
`counterparty`	Direct parties	Routing, status, custody handoffs -- enough to do their job
`owner`	Shipping party only	Full manifest (suppliers, unit costs, volumes), commercial terms

Encryption is AES-256-GCM with per-recipient X25519 key wraps. The recipient's secret key never leaves their device -- decryption is client-side. A Disney can ship a movie tie-in merchandise drop without the carrier seeing what's inside; a Walmart can move private-label inventory without exposing supplier negotiation terms.

SDK Overview

```
npm install @vexidus/vexalus-sdk
```

```typescript
import { Vexalus } from '@vexidus/vexalus-sdk';

const vexalus = new Vexalus({ rpcEndpoint: 'https://api.vexalus.com' });

// Create a new shipment (mints a VSC-21 NFT)
const shipment = await vexalus.shipments.create({
  id: 'SHP-2026-00441',
  origin: 'Shanghai, CN',
  destination: 'Rotterdam, NL',
  cargo: 'Electronics — 240 units',
});

// Add a checkpoint (mints VSC-55 tokens for this event)
await vexalus.checkpoints.add({
  shipmentId: 'SHP-2026-00441',
  type: 'TemperatureReading',
  value_celsius: 18,
});

// Aggregate an environmental envelope verdict
const verdict = await vexalus.environmental.aggregate({
  shipment,
  profileName: 'pharma_cold_chain',
});

// Compile a dispute evidence package (Accountability Web)
const evidence = await vexalus.manifest.compileDisputeEvidence({
  shipment,
  manifest,
  weightObservations,
  sealAttestations,
  evidenceAttachments,
  custodyHandoffs,
});
// → rollupHash binds all six layers; anchor it on-chain for tamper-evident dispute defense
```

REST API

Base URL: `https://api.vexalus.com`

Endpoint	Purpose
`/v1/shipment`	Create / retrieve shipments
`/v1/checkpoint`	Add / list checkpoints
`/v1/custody`	Propose / approve custody handoffs
`/v1/document`	Attest / verify document hashes
`/v1/finance`	Tokenize / query invoices
`/v1/audit`	Full audit trail for a shipment
`/v1/party`	Counterparty identity registry
`/v1/dc`	Distribution-center entities (Role 4)
`/v1/stores`	Retail-store entities (Role 5)
`/v1/inventory`	Per-DC inventory snapshots + on-chain attestation
`/v1/network-events`	Cross-dock + allocation events
`/v1/shipments/encrypted`	Privacy envelope prepare / anchor / verify
`/v1/manifest`	Accountability Web — manifest anchor, weight check, seal-chain verify, dispute evidence compile
`/v1/environmental`	Environmental envelope aggregate / anchor / verify
`/v1/contact`	Public contact form intake

Swagger UI: `https://api.vexalus.com/v1/docs` OpenAPI JSON: `https://api.vexalus.com/v1/openapi.json`

Five-Role Data Model

Role	Entity	What they do
1. Manufacturer / Producer	Originating party	Mint shipments, attach packing manifests
2. Carrier / Transport	Custodian during transit	Record checkpoints, propose handoffs
3. Buyer / Receiver	Destination party	Approve handoffs, scan-and-verify on receipt
4. Distribution Center	DC NFT entity	Cross-dock events, inventory snapshots, fan-out to children
5. Retail Store	Store NFT entity	Allocation decisions, store-level inventory

A parent shipment can fan out into typed children allocated to specific stores, with per-SKU quantity validation that the fan-out preserves the total.

ROI Simulation Engine

The simulation engine at vexalus.com/simulation is a white-box ROI calculator. Each savings wedge is a pure function with sanity caps preventing compounding-fractions overestimation. Eight wedges (dispute elimination, compliance automation, working capital acceleration, safety stock reduction, manual reconciliation elimination, insurance premium reduction, distribution routing optimization, counterfeiting / shrinkage reduction) sum to a total annual savings figure traceable line-by-line.

Canonical Disney and Walmart profiles tie out to industry-published baseline numbers (Walmart dispute costs hit the $2-4B published range, sanity-capped). The prospect's analyst can plug their own numbers in and rebuild the calculation from first principles. See the Disney / Walmart enterprise case study via the Vexalus contact form for the full narrative.

IoT Edge Agent

Vexalus includes a lightweight IoT agent for edge device checkpoint submission. The agent runs on any Linux device (including Raspberry Pi) and persists every event to a local SQLite queue before any network attempt -- so a process crash or unexpected reboot does not drop telemetry.

```
# Install and configure the agent
npm install -g @vexidus/vexalus-iot

vexalus-agent init --device-id IOT-2026-001 --keypair ./device-key.json

# Start streaming checkpoints
vexalus-agent start \
  --shipment SHP-2026-00441 \
  --interval 60 \
  --sensors temperature,humidity,gps
```

The agent submits signed checkpoint batches at the configured interval. If connectivity is lost, events queue locally and drain on reconnect with exponential backoff. The agent process can pause / resume / flush-soon programmatically, and stuck `forwarding` events recover on restart.

Cost Model

Vexalus operations use standard Vextaris gas pricing (~10 nanoVXS per unit). A complete shipment lifecycle — creation, four checkpoints, two custody handoffs, two document attestations, an environmental envelope verdict, a manifest anchor, and an invoice tokenization — costs approximately **$0.016 at current VXS testnet pricing**.

Operation	Approximate Cost
Create shipment (VSC-21 mint)	~$0.0002
Checkpoint (VSC-55 mint)	~$0.0002
Custody handoff (VSC-88 multisig)	~$0.0004
Document attestation (hash anchor)	~$0.0001
Manifest anchor (Accountability Web Layer 2)	~$0.0001
Environmental envelope verdict anchor	~$0.0001
Dispute evidence rollup anchor	~$0.0001
Invoice tokenization (VSC-7 mint)	~$0.0002

A Disney / Walmart-scale dispute at industry-baseline cost is in the $10K-50K range and takes 60-120 days. The equivalent Vexalus dispute defense produces the evidence package as a byproduct of normal operation at less than 2 cents per shipment.

Multi-Zone Deployment

The Vexalus data model includes a `zone` field on all entities (default: `"earth"`). This field is future-proofed for multi-zone deployments — enabling the same shipment tracking infrastructure to operate across logistically distinct domains without changes to the core protocol.

Current Status

Capability	Status
Five-role data model (Manufacturer / Carrier / Buyer / DC / Store)	Live
VSC-21 shipment NFTs + VSC-55 checkpoint events	Live
VSC-88 multi-sig custody handoffs	Live
BLAKE3 document attestation	Live
VSC-7 invoice tokenization	Live
Privacy envelope (AES-256-GCM + X25519 + three-tier disclosure)	Live
Distribution Center + Store + Inventory entities	Live
Cross-dock + Allocation events	Live
Accountability Web (7 verification layers + dispute compiler)	Live
VIDA inspector credential type (Layer 7)	Live
Environmental envelope aggregator (6 vertical profiles)	Live
Store-and-forward IoT agent with durable queue + retry	Live

Capability	Status
ROI simulation engine	Live (vexalus.com/simulation)
Public marketing site + contact form	Live (vexalus.com)
Integrated dashboard UI (shipment map + timeline overlay)	Live
ERP / WMS integration adapter templates	Planned
RFID / NFC integration guide	Planned

Key Links

Resource	URL
Vexalus marketing site	vexalus.com
API	api.vexalus.com
ROI simulator	vexalus.com/simulation
Contact	vexalus.com/contact
API documentation	api.vexalus.com/v1/docs
VIDA inspector framework	VIDA Identity Protocol
Token Standards	VexForge Token Overview
Governance / Multi-sig	VSC-88 Multi-Sig Wallets

VNS — Vextaris Name Service

VNS provides human-readable `.vex` names for Vextaris addresses. Names are permanent, require no annual renewal, and are backed by VSC-21 NFTs — ownership of the NFT is the name registration. Transfer the NFT, transfer the name. No separate registry update is needed.

VNS is an identity extension of VSA v2 (Vextaris Smart Accounts). It adds a human-readable layer to an account system that VSA already manages — and because VNS names compose natively with IntentVM, the same name works everywhere: payments, intents, and VexFi.

How It Works

1. **Register a name** — Submit a `vex_registerName` transaction. The name is validated for length and availability, the registration fee is collected, and a VSC-21 NFT is minted to your address.
2. **NFT = registration** — The NFT owner is the name's resolution target. Resolution reads the NFT's current owner from chain state — no off-chain registry.
3. **Transfer = re-route** — Transferring the NFT immediately redirects resolution to the new owner. No admin call required.
4. **Permanent** — No renewal fees. The name is yours until you transfer or burn the NFT.

```
Register "alice.vex"
  → Validates: name available, length ≥ 3 chars
  → Collects fee (100 VXS for "alice")
  → Mints VSC-21 NFT in global VNS collection
  → Stores resolution: vns:alice → owner address
  → Resolves: vex_resolveName("alice.vex") → Vx...
```

Pricing

Name Length	Registration Fee
3–4 characters	1,000 VXS

Name Length	Registration Fee
5–7 characters	100 VXS
8+ characters	10 VXS

Fees are collected once at registration. There is no recurring charge.

RPC Methods

VNS exposes four dedicated RPC methods on any Vextaris node.

`vex_registerName`

Register a new `.vex` name. The calling address must hold enough VXS to cover the registration fee.

```
{
  "method": "vex_registerName",
  "params": {
    "name": "alice",
    "owner": "Vx1abc..."
  }
}
```

Response:

```
{
  "name": "alice.vex",
  "nftId": "VSC21:0x...",
  "fee": "100000000000",
  "txHash": "Vxh..."
}
```

`vex_resolveName`

Resolve a `.vex` name to its current owner address.

```
{
  "method": "vex_resolveName",
  "params": ["alice.vex"]
}
```

Response:

```
{
  "name": "alice.vex",
  "address": "Vx1abc...",
  "nftId": "VSC21:0x..."
}
```

`vex_reverseResolve`

Look up the primary `.vex` name for a given address (if any).

```
{
  "method": "vex_reverseResolve",
  "params": ["Vx1abc..."]
}
```

Response:

```
{
  "address": "Vx1abc...",
  "name": "alice.vex"
}
```

`vex_getNameInfo`

Get full metadata for a registered name: owner, NFT ID, registration block, and fee paid.

```
{
  "method": "vex_getNameInfo",
  "params": ["alice.vex"]
}
```

Response:

```
{
  "name": "alice.vex",
  "owner": "Vx1abc...",
  "nftId": "VSC21:0x...",
  "registeredAtBlock": 1041200,
  "feePaid": "100000000000"
}
```

Ecosystem Integration

VNS names are resolved automatically across the Vextaris ecosystem:

- **VexFi** `pay()` — Pass a `.vex` name directly as the payment recipient. Resolution is handled internally before bundle construction.
- **IntentVM** — Natural language intents accept `.vex` names: `"send 50 VXS to alice.vex and stake the rest"` is one intent, one signature.
- **VexSpark wallet** — The wallet displays `.vex` names wherever available, and allows searching by name in the send flow.
- **VexScan explorer** — Addresses with registered names display the `.vex` name in transaction history and account views.

Registering via CLI

```
# Register a name
vexidus vns register --name alice --from Vx1abc... --rpc
http://localhost:9933

# Resolve a name
vexidus vns resolve alice.vex --rpc http://localhost:9933

# Look up names for an address
vexidus vns reverse Vx1abc... --rpc http://localhost:9933
```

Key Links

Resource	URL
VNS App	vns.name
Wallet (name management)	wallet.vexspark.com/names
VSA v2 Smart Accounts	VSA Architecture
IntentVM	IntentVM Architecture
VexFi (uses VNS)	VexFi — Financial Infrastructure

Bushido Passport -- Universal Gamer Identity

Bushido Passport is a universal gamer identity protocol native to the Vextaris blockchain. Players carry a provable, portable identity -- including achievements, skill ratings, reputation, and portable items -- across any game or platform that integrates the SDK.

Why Bushido Passport?

For gamers:

- One identity across all games -- your rank, achievements, and items follow you
- Soulbound achievement NFTs that prove what you've earned (no buying achievements)
- Quantum-resistant: your identity is still yours in 2040

For game developers:

- One-call SDK integration: `bp.attest()` handles everything
- No need to build your own identity system
- Players arrive with a pre-built reputation -- instant personalization
- Earn royalties on portable item resales

Core Concepts

Passport Account

Every wallet can create one Bushido Passport -- an on-chain identity with:

- **Samurai Name** -- unique display name across all games
- **Mon** -- profile image/crest
- **Skill Ratings** -- four ELO-style ratings that evolve across games
- **Samurai Rank** -- calculated from skill averages
- **Achievements** -- soulbound VSC-21 NFTs from specific games

- **Items** -- portable game assets with configurable tradability
- **Honor Points** -- lifetime achievement score

Skill System

Four skill categories (baseline 1000, max 5000):

Category	What It Measures
Combat Prowess	Action, PvP, reflexes
Strategic Wisdom	Planning, puzzle-solving, tactics
Social Honor	Teamwork, community contribution
Exploration Spirit	Discovery, completionism

Games submit skill deltas via attestations. Your rank evolves automatically:

Rank	Skill Average	Japanese
Ashigaru	< 1200	足軽
Gokenin	1200+	御家人
Hatamoto	1400+	旗本
Daimyo	1700+	大名
Shogun	2000+	将軍

Achievements (Soulbound NFTs)

Games mint achievements as VSC-21 soulbound tokens. Each achievement records:

- Which game awarded it
- What type of achievement
- Skill impact (how much each rating changed)
- Rarity (what percentage of players earned it)

- Timestamp

Achievements are **non-transferable** -- you can't buy someone else's Dragon Slayer badge.

Portable Items

Games can mint items to player passports with configurable portability:

Level	Can Port to Other Games	Can Sell on Marketplace
Locked	No	No
Portable	Yes	No
Tradeable	Yes	Yes (game earns royalty)
Unrestricted	Yes	Yes (no royalty)

Games earn 0-15% royalty on every resale of items they created.

SDK Integration

For Game Developers (Node.js)

```
import { BushidoPassport } from '@bushido-passport/sdk'

const bp = new BushidoPassport({
  rpcUrl: 'https://testnet.vextaris.io',
  gameId: 'your_game',
  gameWallet: gameKeypair,
})

// Award achievement + update skills -- ONE call
await bp.attest(playerWallet, {
  type: 'dragon_slayer',
  name: 'Dragon Slayer',
  combat: +150,
  wisdom: +50,
})
```

For Any Game Engine (REST)

```
curl -X POST https://testnet.vextaris.io \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "vex_submitAttestation",
    "params": ["<game_wallet>", "<player>", "your_game",
               "dragon_slayer", "Dragon Slayer", null, null,
               150, 50, 0, 0],
    "id": 1
  }'
```

Works from Unity, Unreal, Godot -- any engine that can make HTTP requests.

Read Player Data

```
import { BushidoPassportReader } from '@bushido-passport/sdk'

const reader = new BushidoPassportReader('https://testnet.vextaris.io')
const passport = await reader.getPassport(walletAddress)
const achievements = await reader.getAchievements(walletAddress)
const items = await reader.getItems(walletAddress)
```

Quantum-Resistant Identity

Vextaris accounts can hold a post-quantum key: a CRYSTALS-Dilithium3 key, alongside or instead of Ed25519, is verified on-chain today. A passport held by an account that signs with that key does not depend on the classical signatures quantum computers are expected to break.

When quantum computers break the classical signatures most chains rely on (ECDSA on Ethereum, Ed25519 on Solana, and Ed25519 on any Vextaris account that has not added a post-quantum key), a passport on an account that signs with Dilithium3 stays verifiable. **Add the key, and your rank is still yours in 2040.**

Operations

Disc	Operation	Description
71	CreatePassport	Create passport with samurai name + mon
72	UpdatePassportProfile	Change name or mon
73	TransferPassport	Wallet migration / recovery
74-77	Game Registry	Register, update, deactivate, verify games
78	SubmitAttestation	Award achievement + update skills (one call)
79	SetClanMembership	On-chain clan proof
80-82	Admin	Freeze, unfreeze, revoke achievement
83-86	Items	Mint, transfer, update portability, burn
87-89	Marketplace	List, buy, cancel

RPC Methods

```
vex_getPassport(address)              -- Full passport data
vex_getPassportByName(name)           -- Lookup by samurai name
vex_getAchievements(address, game?)   -- Achievement NFTs
vex_getItems(address, game?)          -- Portable items
vex_getGameInfo(gameId)               -- Game details
vex_getMarketplaceListings(game?)     -- Active item listings
vex_getPassportStats()                -- Network-wide stats
```

Transaction Cost

~$0.0003 per attestation. Games can award thousands of achievements per day without meaningful cost. This makes the system viable for indie developers, not just large studios.

Vextaris Testnet Reference

> Updated: November 2026 | Running on a fresh genesis (relaunched July 2026) | RPC: https://testnet.vextaris.io

EVERY TRANSACTION MUST BE SIGNED

Since 31 October 2026 every validator refuses unsigned transactions. Sign client-side with the SDK, the CLI (`--key <path>`) or a wallet such as VexSpark, and submit with `vex_submitBundle`. Write methods that build a transaction on the node from an address you name (e.g. `vex_transfer`, `vex_bridgeToken`, `vex_updateTokenMetadata`) return `refused: unsigned bundle … sign it client-side and submit with vex_submitBundle`. Read methods are unaffected. See Testnet Security Model.

Network

Property	Value
Chain ID (Testnet)	1618032 (`0x18b070`)
Block Time	Adaptive (~2s normal, 500ms-12s range)
Native Token	VXS (9 decimals)
RPC Port	9933
P2P Port	9945 (seed node; default is 9944)
Public IP	51.255.80.34

System Accounts

Name	Address (32-byte hex)	VXS Balance	%
Public Distribution	`0x...0001`	614,840,000 VXS	38%

Name	Address (32-byte hex)	VXS Balance	%
Ecosystem & DeFi	`0x...0002`	242,700,000 VXS	15%
Foundation Treasury	`0x...0003`	161,800,000 VXS	10%
Team & Founders	`0x...0004`	161,800,000 VXS	10%
Marketing & Community	`0x...0005`	113,260,000 VXS	7%
SAFT Round	`0x...0006`	48,540,000 VXS	3%
Testnet Rewards	`0x...0007`	48,540,000 VXS	3%
Staking & Strategic	`0x...0008`	161,800,000 VXS	10%
Block Proposers (validators)	Rotated via leader selection	Earn time-weighted rewards (85%) + fees (80%)	4% over 10yr
System (protocol ops)	`0xfffffff

Category	Examples	Notes
Bridge Canonical (13+ tokens)	ETH, BTC, SOL, POL, MONAD, etc.	Testnet only. Inbound bridge deposits are paused since 31 Oct 2026. Planned for mainnet: no unbacked bridged tokens -- major assets are reached by swapping through Atomence

Use `vex_listTokens` to get the full current list with mint addresses:

```
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_listTokens","params":[50],"id":1}'
```

Quick RPC Examples

```
# Get VXS balance
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getBalance","params":
["0x0000000000000000000000000000000000000000000000000000000000000001","VX
S"],"id":1}'

# Get token balance by symbol
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getBalance","params":
["0x0000000000000000000000000000000000000000000000000000000000000001","US
DC"],"id":1}'

# List all tokens
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_listTokens","params":[100],"id":1}'

# Submit a signed transaction (sign client-side with the SDK, CLI or a
wallet first)
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_submitBundle","params":["
<signed_bundle_borsh_hex>"],"id":1}'

# Node version
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_version","params":[],"id":1}'

# Consensus status
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_consensusStatus","params":
[],"id":1}'
```

Active Pools

Use `vex_listPools` to get current pools:

```
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_listPools","params":
{"limit":10},"id":1}'
```

Token Metadata

Metadata updates are signed transactions from the token creator. With the CLI:

```
# Update token metadata (creator only, token must be mutable)
vexidus --rpc https://testnet.vextaris.io --key creator.key \
  token update-metadata --mint <MINT> --sender <CREATOR_ADDRESS> \
  --image https://example.com/logo.png --description "Token description"

# Lock token metadata permanently (creator only, one-way)
vexidus --rpc https://testnet.vextaris.io --key creator.key \
  token lock-metadata --mint <MINT> --sender <CREATOR_ADDRESS>
```

```
# Get token reputation score (read-only)
curl -s -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_getTokenReputation","params":
["0x...mint"],"id":1}'
```

Validators (7-Node Testnet)

The testnet runs 7 validators across four continents. The live validator set and their keys are queryable via the explorer at vexscan.io and the `vex_listValidators` RPC method.

Leader rotation is active with stake-weighted selection -- all 7 validators take turns proposing blocks across 4 continents. Block rewards are time-weighted (85% to proposer, 15% Foundation) with a density multiplier (0.5x empty to 2.0x heavy). Transaction fees split 80% proposer / 20% Foundation.

Notes

- **Token mint addresses** are deterministic. Genesis tokens (VSC-8, bridge canonical) are stable across restarts. Custom tokens include a timestamp component and change on data wipe.
- **All balances are raw units** (multiply by 10^decimals). Divide by 10^decimals for human-readable values. Token decimals vary: VXS (9), USDC/USDT (6), BTC (8), ETH (18). Use `vex_listTokens` to get each token's decimal count.
- **VXS address is all zeros** (`0x00...00`). This is the native token, not a custom token.
- **Signed transactions**: Every transaction must be signed (unsigned transactions are refused). Wallets sign bundles with Ed25519 and submit via `vex_submitBundle`. Current CLI, SDK and VexSpark releases sign with the **canonical** scheme by default; legacy signatures are refused from block 33,000,000, so update older client builds before then. The first send from a receive-only account requires **pubkey revelation** so the network can verify the signing key. The native wallet VexSpark (wallet.vexspark.com) handles this automatically.
- **Block rewards**: Time-weighted from 13M VXS/yr (Year 1-2). 85% to proposer (density-weighted), 15% to Foundation Treasury.
- **Gas fees**: 10 nanoVXS per gas unit (~0.00021 VXS per transfer, ~$0.0002 at $1/VXS).
- **Faucet**: https://vexswap.xyz -- max 100 VXS per address per hour.
- **Transaction pipeline verified**: Signed sends, gas deduction, leader rotation, millions of txs processed with 0 crashes. 32+ VexVisor upgrades deployed with zero manual intervention.
- **WebSocket subscriptions**: Real-time push notifications via `vex_subscribe_newHeads`, `vex_subscribe_newTransactions`, and `vex_subscribe_logs`. See RPC Reference.
- **Leader skip rotation**: Offline validators (3+ consecutive misses) are automatically skipped in leader rotation, maintaining ~2s block time even with validator downtime.
- **Genesis**: The current testnet has run on a fresh genesis since its July 2026 relaunch on BFT consensus.

Vextaris Load Test Results

> Latest: July 2026 — relaunched on BFT consensus (fresh genesis). A sustained-TPS benchmark on the new consensus is in progress. Network: 7-validator testnet across 4 continents Protocol: BFT consensus with deferred execution, Dragonfly Stream (mempoolless direct delivery)

Throughput

Vextaris provides instant, deterministic BFT finality (a committed block is final — no confirmations) and a high-throughput native execution engine. A sustained-TPS benchmark on the current consensus is in progress; we do not publish throughput numbers we have not re-verified.

Metric	Value
Finality	**Deterministic BFT** — a committed block is final, typically well under two seconds
Block time	**Sub-second to ~1.5s** (7 validators, 4 continents)
RPC rejection	**0%** (all bundles accepted)

Network Configuration

Parameter	Value
Validators	7 (across 4 continents)
Block time	sub-second to ~1.5s (consensus-set under BFT)
Max tx/block	50,000 (configurable via `--max-txs-per-block`)
LeaderBuffer capacity	100,000
Gas price	10 nanoVXS/gas

Parameter	Value
Leader rotation	Enabled (pure stake-weighted, leader skip recovery)
P2P transport	Dragonfly Stream (sealed direct-to-leader, PQ envelopes)
Wire encoding	Borsh binary with JSON fallback (dual-format)
Storage	ElasticState 3-tier (Hot/Warm/Cold) + RocksDB bloom filters + 512MB cache

Validator Nodes

> The network relaunched on BFT (fresh genesis, July 2026) with 7 validators across 4 continents. The table below reflects the prior 5-validator configuration.

Node	Location	Stake	Notes
Gravelines	OVH, France	10,000 VXS	Seed + validator
Remote 1	Contabo, Germany	10,000 VXS	
Remote 2	Contabo, Germany	10,000 VXS	Top producer
SG2	OVH, Singapore	10,000 VXS	
Mumbai	Contabo, India	10,000 VXS	

Test Methodology (June 2026)

> The measured results in the following sections (methodology, results summary, block-by-block, and engine execution speed) were measured on the prior immediate-execution consensus; superseded by the relaunch — re-benchmark pending.

Audit-grade Rust load tester (`tools/loadtest-rs/`) with pre-signed Ed25519 bundles. Each wallet sends exactly 1 bundle (nonce 0) — zero nonce contention by design. Wallet count auto-calculated from target TPS.

Transaction type: VXS transfers (10 per bundle, 1 nanoVXS each). Lightest operation type for pure throughput measurement.

Submission: Single RPC endpoint (`localhost:9933`), 128 concurrent async workers, token-bucket rate limiter at 100 bundles/sec. Bundles forwarded to leader via Dragonfly Stream direct routing.

Verification: Balance-delta checking (pre-test vs post-test wallet balances) + node execution logs as ground truth.

Results Summary (June 2026)

Metric	Value
Bundles submitted	2,712 (100% RPC acceptance)
Bundles in blocks	2,714 (block-level count)
Block span	9 blocks (~27s)
Block-level TPS	1,005 ops/sec
Submit rate	90.4 bundles/sec (904 ops/sec)
Peak block	#1760860: 1,318 bundles (4,393 ops/sec)
Latency p50 / p99	0.3ms / 552ms

Block-by-Block Breakdown

Block	Bundles	Ops	Ops/sec
#1760860	1,318	13,180	4,393
#1760861	301	3,010	1,003
#1760862	285	2,850	950

Block	Bundles	Ops	Ops/sec
#1760863	281	2,810	937
#1760864	448	4,480	1,493
#1760865	81	810	270

Engine Execution Speed

From node execution logs (precise `Instant::elapsed()` timing):

Block	Proposed	Executed	Time	Speed
#1759061	1,336	1,329	16ms	**83,062 bundles/sec**
#1759155	266	266	3ms	**88,667 bundles/sec**
#1759156	281	281	4ms	**70,250 bundles/sec**
#1759428	641	641	15ms	**42,733 bundles/sec**

The execution engine processes bundles at 42,000–89,000 bundles/sec depending on block size. With 10 operations per bundle, this translates to **420,000–890,000 operations/second** of raw processing speed.

Throughput Ceiling

Under BFT, block cadence is consensus-set (round timeout + 2f+1) rather than driven by a fixed producer interval, so prior throughput ceilings derived from that interval no longer apply and have been retired. A sustained-TPS benchmark on the current consensus is in progress; we do not publish throughput numbers we have not re-verified.

The execution and storage stack that supports high throughput remains in place: bundle-batched native execution, RocksDB with bloom filters and write buffering, Borsh binary wire encoding, and ElasticState tiered storage.

What Limits Throughput

In load tests on the prior consensus, the execution engine was never the bottleneck — the limits were client-side. Those characteristics of the submission path still hold:

1. **Single-RPC submission:** All bundles submitted to one node (Gravelines). Distributed submission across 7 RPCs would increase throughput proportionally.
2. **HTTP round-trip:** 128 concurrent workers limited by TCP connection pool and serialization overhead.
3. **Dragonfly forwarding:** ~75% of bundles delivered to leader on first attempt. Remaining arrive via bootnode relay.

What is NOT a bottleneck

- **Execution speed:** the engine sat idle the vast majority of the time during load tests
- **RocksDB writes:** WriteBatch commits in under 1ms. Bloom filters, 512MB cache, periodic flush (every 100 blocks) all deployed.
- **Block capacity:** Peak observed is 1,318 bundles (2.6% of 50K limit)
- **Network gossip:** Block propagation uses Borsh binary encoding (5-10x smaller than JSON). Across 4 continents in under 500ms.
- **State reads:** ElasticState hot tier provides sub-microsecond reads for active accounts

---

Historical Results (March 5, 2026)

> (Measured on the prior immediate-execution consensus; superseded by the relaunch — re-benchmark pending.)

Earlier Python-based load test with 500 wallets and 11 operation types:

Metric	Value
Submitted	12,361 (100% RPC acceptance)
On-chain success	92% (sampled)

Metric	Value
Peak block	2,856 transactions
Submission TPS	67.9
Inclusion TPS	144.5
Operation types	11 (transfers, swaps, intents, liquidity)

This test was limited by Python client throughput and nonce collisions. The Rust load tester (March 29) eliminated both bottlenecks.

Current Testnet State

Metric	Value
Network	Relaunched on BFT consensus (fresh genesis, July 2026)
Validators	7 (4 continents)
Finality	Deterministic BFT — a committed block is final
Block time	sub-second to ~1.5s

Transaction Cost

Operation	Gas	Cost (at $1/VXS)
VXS Transfer	~21,000	~$0.00021
Token Create	~50,000	~$0.0005
Swap	~55,000	~$0.00055
Bridge Deposit	~100,000	~$0.001

Operation	Gas	Cost (at $1/VXS)
Name Registration	~60,000	~$0.0006 + name fee

Gas price: 10 nanoVXS per gas unit. Fees split: 80% to block proposer, 20% to Foundation Treasury.

Key Takeaways

- **Deterministic BFT finality** — a committed block is final, typically well under two seconds, no confirmations needed
- **Sub-second to ~1.5s blocks** across 7 validators on 4 continents
- **High-throughput native execution engine** — a sustained-TPS benchmark on the current consensus is in progress; we do not publish throughput numbers we have not re-verified
- **~$0.0002 per transaction** — Solana-competitive fees
- **Borsh binary wire protocol** — 5-10x smaller block propagation across all validators
- **ElasticState active** — 3-tier intelligent storage with sub-microsecond hot reads

Vextaris Keypair and Recovery Reference

Keypair Generation (Server-Side)

RPC Method: `vex_generateKeypair`

Generates an Ed25519 keypair:

```
curl -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_generateKeypair","params":
[],"id":1}'
```

Returns:

```
{
  "address": "Vx0...",
  "public_key": "<64 hex chars>",
  "secret_key": "<64 hex chars>",
  "key_type": "Ed25519"
}
```

How It Works

1. An Ed25519 signing key (32 bytes) and verifying key (32 bytes) are generated
2. Address derived: `SHA256(public_key)` -> 32 bytes -> base58 with `Vx0` prefix
3. EVM-compatible address: last 20 bytes of the 32-byte address -> `0x` hex

Dual Key Types

Vextaris accounts support two signature schemes, giving every account a clear path to post-quantum security:

Type	Pubkey Size	Signature Size	Role
Ed25519	32 bytes	64 bytes	Fast, compatible classical signatures used for everyday signing
Dilithium3 (NIST Level 3)	1,952 bytes	3,293 bytes	Post-quantum signatures for long-term quantum hardening

Signature Verification

Current Pipeline

1. **Bundle signing**: The client signs the bundle's signing hash with its Ed25519 secret key. The **canonical** signing hash covers every field that determines what the transaction does -- sender, operations, gas limits, validity window, nonce, revealed public key and gas sponsor.
2. **Submission**: The signed bundle is submitted with `vex_submitBundle`. Unsigned transactions are refused by every validator, including those built on the node by convenience write methods such as `vex_transfer`.
3. **Verification**: During execution, the network:
   - Checks the account's registered signing keys for one authorized to sign transactions
   - Accepts either an Ed25519 (64-byte) or Dilithium3 (3,293-byte) signature
   - **Pubkey revelation**: The first signed send from a receive-only account includes the sender's public key. The network confirms the key hashes to the account address and registers it for future sends.

Signature Schemes

Scheme	Status
Canonical	Accepted since block 18,700,000. Default in current releases of the CLI, Rust SDK, `@vexidus/sdk` and VexSpark.
Legacy	Accepted until block 32,999,999. **Refused from block 33,000,000.** Update any older client build before then.

The CLI accepts `--signing-scheme canonical|legacy` (or `VEXIDUS_SIGNING_SCHEME`) to choose explicitly; the default is canonical.

Pre-Signed Bundle Submission

```
curl -X POST https://testnet.vextaris.io -H "Content-Type:
application/json" \
  -d '{"jsonrpc":"2.0","method":"vex_submitBundle","params":["
<borsh_hex>"],"id":1}'
```

Guardian Recovery (VSA v2)

Status: Fully Implemented

Guardian recovery is live as a set of native account operations.

Setup

An account owner configures recovery by specifying:

- **Guardians** -- up to 10 trusted Vx0 addresses
- **Threshold** -- the M-of-N guardian approvals required to recover
- **Timelock** -- a delay before recovery can finalize (recommended: 48-72 hours)

Recovery Flow

1. **User loses all keys**
2. **Guardians initiate**: M-of-N guardians submit `InitiateRecovery` with a new public key
3. **Timelock starts**: The pending recovery is recorded on the account
4. **Owner can cancel**: If they regain access during the timelock, submit `CancelRecovery`
5. **After timelock**: Anyone submits `FinalizeRecovery` -- replaces all account keys with the recovery key

Operations

Operation	Who Signs	What Happens
`SetRecovery`	Account owner	Configures guardians, threshold, timelock

Operation	Who Signs	What Happens
`InitiateRecovery`	Guardian	Starts recovery with new pubkey, records guardian approval
`CancelRecovery`	Account owner	Cancels pending recovery
`FinalizeRecovery`	Anyone	After timelock, replaces keys. Requires threshold guardian approvals.

Recovery State

Each account's recovery configuration tracks its guardian set, the approval threshold, the timelock duration, and any in-progress recovery. A pending recovery records the proposed new public key and key type, when it was initiated, and which guardians have approved so far.

Key Management (Also Live)

Operation	Purpose
`AddKey`	Add an Ed25519 or Dilithium3 key to an account
`RemoveKey`	Remove a key (must keep at least one)
`RotateKey`	Atomic swap of one key for another

Quantum Migration Path

- **Phase 1 (Now)**: Ed25519 only. Wallet SDK compatible (Phantom, MetaMask via bridge).
- **Phase 2 (~2028)**: Users add Dilithium3 key alongside Ed25519 via `AddKey`. Either key signs. Address unchanged.
- **Phase 3 (~2032+)**: High-value ops require Dilithium3. V0-only accounts get warnings.

The infrastructure is ready -- post-quantum key generation works and the verification pipeline handles both Ed25519 and Dilithium3, so the migration requires no protocol upgrade.

Security Best Practices

Key Storage

- Store wallet keys with `chmod 600`
- Never log or print secret keys in production
- Store backups encrypted on offline media
- For web wallets, use the browser's `SubtleCrypto` API to encrypt keys at rest

Nonce Management

- Nonces are sequential: transaction with nonce=5 only executes after nonce=4
- Always fetch the current nonce immediately before signing
- If a transaction fails, the nonce is not consumed -- retry with the same nonce

Testnet vs Mainnet

Behavior	Testnet	Mainnet
Chain ID	1618032	1618033
`vex_generateKeypair`	Available	Disabled
Transaction signature enforcement	**Enforced** (since 31 October 2026) — see Testnet Security Model	**Enforced**

Never reuse testnet keys on mainnet. Generate fresh keypairs for production.

Network Defense Model

This page documents the complete threat mitigation posture of the Vextaris network across all attack categories. It is a living document -- updated as defenses are deployed and new threats are identified.

Last updated: November 2026 | **Network stage**: Public beta testnet

---

Defense Philosophy

Vextaris takes a **structural elimination** approach to network security. Rather than adding mitigations after vulnerabilities are discovered, the architecture is designed to make entire categories of attack structurally impossible.

Key principles:

- **Eliminate preconditions** -- remove the conditions that enable attacks, not just detect the attacks themselves
- **Defense in depth** -- every layer (P2P, consensus, RPC, economics) has independent protections
- **Graceful degradation** -- under attack, the network slows down but never crashes or forks
- **Economic deterrence** -- attacks that can't be structurally prevented are made economically irrational

---

MEV Elimination (~95% of Known Vectors)

Maximal Extractable Value (MEV) is the largest systemic risk to L1 blockchains. On Ethereum, MEV extraction exceeds $600M annually. Vextaris eliminates the preconditions for MEV through four independent mechanisms.

Attack Surface Removed

Attack	How Traditional Chains Are Vulnerable	Vextaris Defense	Status
Front-running	Bots observe pending txs in gossip mempool	**No mempool.** Dragonfly Stream delivers txs directly to leader via sealed P2P pipeline. No public pool to observe.	Deployed
Sandwich attacks	Attacker wraps victim tx with front-run + back-run	**No mempool + atomic execution.** Can't see victim tx; IntentVM executes multi-step operations atomically with no inter-step window.	Deployed
Back-running	Bots position txs after profitable opportunities	**Deterministic ordering.** Content-based transaction ordering removes all proposer ordering discretion.	Deployed
Proposer reordering	Block proposer shuffles txs for profit	**Content-determined order.** Ordering is a pure function of transaction content -- proposer cannot influence it.	Deployed
Liquidation sniping	Bots race to liquidate positions they observe in mempool	**No mempool.** Liquidation txs flow directly to leader; no broadcast for bots to intercept.	Deployed
JIT liquidity	LPs add/remove liquidity around observed swaps	**No mempool + deterministic ordering.** Can't observe target swap or control position relative to it.	Deployed
Proposer censorship	Leader drops txs from competitors	**Attestation commitments.** Validators commit received tx hashes in attestation blocks. Censorship is cryptographically detectable.	Detectable
Quantum tx forgery	Future quantum computers forge transaction signatures	**Post-quantum seals.** Every forwarded tx carries a post-quantum cryptographic seal proving temporal provenance.	Deployed

How It Compares

Chain	Mempool	Proposer Ordering	Atomic Multi-Op	MEV Surface
Ethereum	Public gossip	Proposer-controlled (PBS mitigates redistribution)	No	~100% exposed
Ethereum + Flashbots	Private relay (partial)	MEV-Boost auction	No	~60-70% redistributed, not eliminated
Solana	No gossip (leader-based)	Leader FIFO (no deterministic sort)	No	~40-50% (leader advantage)

Chain	Mempool	Proposer Ordering	Atomic Multi-Op	MEV Surface
Vextaris	**No mempool (Dragonfly)**	**Deterministic content-based ordering**	**Yes (IntentVM)**	**~5% remaining**

Most chains attempt to **redistribute** MEV (Flashbots, MEV-Share, PBS). Vextaris **eliminates the preconditions** for MEV to exist. The attack surface isn't mitigated -- it's structurally absent.

DDoS & Spam Protection

Transaction Layer

Defense	Mechanism	Threshold	Status
Signature enforcement	Every state-changing transaction must carry a valid signature from a key registered to the sending account. Unsigned transactions are refused at the RPC and by every validator; the reserved system account cannot be used as a sender from outside.	All transactions	Deployed
Canonical signing	The canonical signature covers every field that determines what a transaction does. Accepted since block 18,700,000; legacy signatures are refused from block 33,000,000.	Per transaction	Deployed
Gas fees	Every transaction costs gas (10 nanoVXS/gas). Spamming 1M transactions costs ~21 VXS.	~$0.0002/tx at $1/VXS	Deployed
LeaderBuffer cap	Hard limit of 100,000 pending transactions. When full, new submissions receive `BufferFull` rejection -- graceful degradation, not crash.	100K transactions	Deployed
Per-sender limit	Each sender address can have at most 128 pending transactions in the buffer. Prevents single-account spam.	128 per address	Deployed
Transaction dedup	Duplicate transactions are silently dropped (hash-based seen set). Replay of the same transaction has zero network cost.	Automatic	Deployed
Bundle TTL	Transactions expire after 120 seconds. Expired bundles are rejected on submission and skipped on drain. Prevents stale transaction buildup.	120s default	Deployed

Defense	Mechanism	Threshold	Status
Nonce sequencing	Each account has a monotonic nonce. Out-of-order or replayed nonces are rejected before execution.	Per-account	Deployed
GBI pre-flight validation	Balance, gas, and nonce checks run before buffering. Transactions that can't possibly succeed are rejected immediately with user-friendly error messages (e.g., "Insufficient USDC balance for swap").	Before buffer	Deployed
GBI failure throttling	Per-sender consecutive failure tracking. After 10 consecutive rejected submissions, sender is throttled for 60 seconds. Resets on successful submission.	10 failures → 60s cooldown	Deployed
GBI failed gas charging	Failed bundles still pay 21K base gas × gas_price. Makes zero-cost spam economically unviable.	Per-bundle	Deployed
Token creation rate limits	Per-account rate limiting on expensive operations (token creation, presale creation). Prevents resource exhaustion through state bloat.	Per-epoch per-account	Deployed
Dynamic gas pricing	Gas price auto-adjusts during sustained congestion, making spam progressively more expensive.	Governance-tunable	Planned

P2P Network Layer

Defense	Mechanism	Threshold	Status
PeerGuard reputation	Every peer has a score (0-100). Malicious behavior reduces score; score 0 = banned. Scores recover slowly through valid blocks.	Ban at 0, start at 100	Deployed
Replay detection	200-hash ring buffer per peer. Replaying a previously-seen block immediately deducts 20 points. 5 replays = ban.	200 block hashes	Deployed
Rate limiting	Per-peer block rate capped at 30 blocks per 12-second interval. Validators get 3x allowance. Exceeding triggers score penalty.	30/12s (90/12s validators)	Deployed
Invalid signature = instant ban	Forged block signature deducts 100 points (instant ban from starting score of 100). No recovery path for forgers.	-100 points	Deployed
Non-leader block rejection	Blocks from non-elected leaders deduct 15 points. Persistent impersonation leads to ban within 6-7 attempts.	-15 per offense	Deployed
Ban escalation	First ban: 1 hour. Repeat offense: 24 hours. Persistent attackers face exponentially increasing lockout.	1h / 24h escalating	Deployed

Defense	Mechanism	Threshold	Status
Validator allowlisting	Authenticated validator P2P identity. On connection, validators prove Ed25519 signing key ownership via domain-separated signature over PeerId. Mutual authentication in one round-trip. Non-authenticated peers can be rejected from block gossip; bootnodes are always exempt. Self-healing on PeerId regeneration.	Authenticated peers	Deployed
QUIC transport	TLS 1.3 + UDP multiplexing with built-in connection-level flow control. Resistant to SYN floods and connection exhaustion.	Protocol-level	Deployed
Dragonfly O(1) bandwidth	Transactions are routed directly to the current block producer over authenticated peer links, with redundancy, rather than broadcast to N peers. Eliminates gossip amplification -- an attacker's spam gets zero network multiplication.	O(1) per tx	Deployed
GossipSub message size limit	GossipSub messages exceeding 10 MB are rejected before processing. Prevents single-message memory exhaustion through the gossip layer.	10 MB max	Deployed
Sync request validation	`GetBlocks` requests with absurd count values (>1,000) are rejected with a reputation penalty. Server-side cap of 10 blocks per request regardless of client-specified count.	count ≤ 1,000; serve ≤ 10	Deployed
Replay hash capacity	Ring buffer of recently-seen block hashes for replay protection. 1,000 hashes retained (32 KB memory). Replaying a previously-seen block costs -20 reputation points.	1,000 hashes	Deployed
Request/response size limits	P2P sync requests capped at 1 MB, responses at 64 MB. Prevents single-request memory exhaustion.	1 MB / 64 MB	Deployed
Sync timeout	Bulk sync requests time out after 60 seconds. Prevents slowloris-style connection holding.	60s	Deployed
RPC connection limit	JSON-RPC server accepts maximum 4,096 concurrent connections.	4,096	Deployed
Response pagination	All list-type RPC methods (tokens, validators, transactions, history) are paginated with hard caps (50-1,000 items). Prevents single-query data explosion.	50-1,000 per page	Deployed

Defense	Mechanism	Threshold	Status
Per-IP RPC rate limiting	Token bucket rate limiting on all RPC endpoints. Each IP gets an independent bucket (default 100 req/s sustained, 200 burst). Exceeded requests receive HTTP 429 with JSON-RPC error. IP extracted from `X-Real-IP`/`X-Forwarded-For` headers (proxy-aware). `--rpc-rate-limit` CLI flag (0 = unlimited).	100 req/s default	Deployed
Per-peer bandwidth tracking	Per-peer byte volume tracking in PeerGuard. Rolling 60-second window, default 100 MB/min per peer (validators get 3x). Tracks gossip, sync, and Dragonfly payloads. Score penalty on exceed, ban on repeat.	100 MB/min default	Deployed

Consensus Layer

Defense	Mechanism	Status
Staked validator set	Only validators with 1,000+ VXS staked can propose or attest blocks. No self-adding -- validator set loaded from on-chain state only.	Deployed
Leader rotation	Block proposer rotates every slot via deterministic, verifiable leader selection. No single validator can monopolize block production.	Deployed
Double-sign detection	Validators producing conflicting blocks at the same height are jailed with escalating penalties (1h, 24h, 7d, 30d). Evidence stored permanently.	Deployed
Attestation validation	Attestation blocks validated for correct signatures, uniqueness (one per validator per height), and zero transaction content. Duplicate attestations treated as double-sign violations.	Deployed
Jailing	Misbehaving validators removed from rotation. No slashing of stake (protects delegators) but missed rewards + reputation damage create strong economic disincentive.	Deployed
Upgrade grace period	100-block (~200s) grace window after upgrade halt prevents jailing validators for upgrading.	Deployed

---

Cryptographic Security

Layer	Algorithm	Purpose	Quantum Status
Transaction signing	Ed25519 (canonical signing hash)	Bundle authentication -- required on every transaction	Classical
Block signing	Ed25519	Block proposer verification	Classical
Transaction seals	Dilithium3 (NIST Level 3)	Temporal provenance proof for Dragonfly Stream	Post-quantum
Address derivation	Blake3	Deterministic address from public key	Quantum-resistant hash
State root	LtHash (homomorphic accumulator)	O(1) per-write state commitment, Blake3 finalization	Quantum-resistant hash
Transaction ordering	Blake3	Deterministic content-based sort	Quantum-resistant hash
Key rotation	AddKey / RemoveKey / RotateKey	VSA v2 account key management	Supports Ed25519 + Dilithium3

Vextaris is **quantum-hardened from day one**. Dilithium3 keys are auto-generated on every validator startup. When quantum migration becomes necessary, key rotation is already built into the account model -- no protocol upgrade required.

Cross-Chain Security (VexBridge & Atomence)

Cross-chain bridges are historically the highest-value attack target in blockchain (>$2B stolen industry-wide). Vextaris answers this with a policy -- **nothing comes to Vextaris that can't leave** -- and with layered protections on every cross-chain flow.

Defense	Mechanism	Status
Attester quorum	A cross-chain settlement is accepted only with signatures from a quorum of registered attesters, each running on separate infrastructure with its own key (on testnet all five are operated by the Vexidus team on five separate machines; independent outside operators are planned before mainnet). Each attester checks the payment on the destination chain and signs a statement naming the order, the recipient, the asset and	Atomence: activating at block 30,655,525

Defense	Mechanism	Status
	the amount paid. Quorum: at least half of the registered attesters for Ethereum and Solana, at least 70% for Bitcoin, never fewer than 3.	
Finality-only attestation	Attesters sign only final payments: Ethereum finalized checkpoint, Solana finalized commitment, Bitcoin at least 6 confirmations.	Activating at block 30,655,525
Amount re-check on-chain	The chain re-checks that the attested amount is at least the minimum the user named when creating the order.	Activating at block 30,655,525
One payment, one order	A destination-chain transaction can settle only one order; it cannot be reused.	Activating at block 30,655,525
Bonded market makers	Only registered market makers with a VXS bond (minimum 1,000 VXS) can match orders.	Activating at block 30,655,525
Settle deadline + guaranteed exit	A match starts a 1-hour settle window. If the market maker does not settle in time, the user reclaims the full escrow plus 10% of the market maker's bond. Funds cannot be locked by a market maker that walks away.	Activating at block 30,655,525
Attester set governance	Attesters are added or removed only by on-chain governance. On testnet, five attesters run on five separate machines; independent outside operators are planned before mainnet.	Deployed
Emergency pause	The Foundation can pause bridge operations in one transaction; resuming requires a governance vote. While paused, Atomence takes no new orders or matches, but orders already in flight can still settle or be refunded -- a pause never locks funds.	Deployed
Inbound deposits paused (testnet)	Inbound VexBridge deposits are paused on testnet while deposit attestation moves to the attester quorum.	Since 31 Oct 2026
Per-user rate limits	Rate limiting on bridge deposits per account to prevent rapid drain attacks.	Deployed
VSC-SAFE scoring	Heuristic safety scoring for foreign token contracts (5-factor analysis). Known scam contracts auto-blocked.	Deployed
Metadata sanitization	All cross-chain metadata (token names, URIs) sanitized for XSS, injection, null bytes, and dangerous patterns.	Deployed

Mainnet model (planned)

- Every token on Vextaris is either **native** (VXS and tokens created on Vextaris) or **issued by a named party obliged to redeem it 1:1** (a chartered stablecoin issuer).
- **No unbacked bridged or synthetic copies of foreign tokens.** Major assets (BTC, ETH, SOL, foreign USDC) are not held as wrapped tokens; users swap into a native asset through Atomence and back, with escrow and refund guarantees on every leg.
- **Stablecoins only from chartered issuers** -- no bridge-minted stablecoins.
- **VexBridge verifies, it does not mint:** its role is proving foreign-chain events through the attester quorum.
- Long-tail foreign tokens are not supported at mainnet.

Network Resilience

These properties have been proven under sustained load testing (55,000+ transactions, 899K+ blocks, peak 2,856 tx/block):

Property	Evidence
Zero crashes under load	Sustained load testing with 11 operation types across 7 validators on 4 continents. 1,210 bundles executed in under one second at peak. Zero node crashes, zero manual restarts.
Graceful buffer saturation	At buffer capacity, new transactions receive `BufferFull` rejection. Existing transactions continue processing normally. No cascading failure.
Geographic fairness	Vexcel attestation DAG ensures all validators produce blocks at a roughly even rate regardless of geographic latency. High-latency nodes (Singapore, Mumbai) are never disadvantaged.
Self-healing	No manual intervention required during any load test. Network recovers automatically when load decreases.
Sub-millisecond RPC	RPC p50 latency 0.5ms under sustained load. RocksDB reads offloaded to blocking thread pool to prevent tokio worker starvation.
Crash-safe state	All block writes committed atomically via RocksDB WriteBatch. No partial state on crash.

Threat Model Summary

Threat Category	Risk Level	Mitigation Status
MEV extraction	Minimal	~95% eliminated (Dragonfly + deterministic ordering + IntentVM). Remaining: proposer censorship (detectable, enforcement planned).
Transaction forgery	Low	Signatures required on every transaction (unsigned refused network-wide) + canonical signing hash covering every transaction field.
Transaction spam	Low	Gas fees + GBI pre-flight validation + failure throttling + LeaderBuffer cap + per-sender limits + TTL + dedup + failed-bundle gas charging. Dynamic gas pricing planned.
P2P network flood	Low	PeerGuard reputation + rate limiting + per-peer bandwidth tracking + QUIC flow control + ban escalation. All deployed.
RPC abuse	Low	Connection limits + pagination + size limits + per-IP token bucket rate limiting (100 req/s default) deployed.
Consensus manipulation	Low	Staked validator set + leader rotation + double-sign detection + jailing + tombstone mechanism + correlation penalty + three-tier key separation + attestation validation.
Cross-chain exploitation	Low	Attester quorum + finality-only attestation + one-payment-one-order + bonded market makers + settle deadline with guaranteed refund + emergency pause that never locks funds. Mainnet plan: no unbacked bridged tokens.
Sybil attacks	Low	1,000 VXS minimum stake for validators. P2P allowlisting available.
Quantum threats	Minimal	Dilithium3 PQ seals deployed. Ed25519 + Dilithium3 dual-key accounts with key rotation.
Eclipse attacks	Low	Explicit bootnode peering + validator allowlisting. Full mesh peering planned.
State bloat	Low	ElasticState 3-tier storage (Hot/Warm/Cold) + per-CF compression + pruning modes + creation rate limits.

Continuous Hardening

The defense model is continuously strengthened as the network matures. Ongoing investment areas include dynamic congestion-based gas pricing, fail-fast signature pre-validation at the RPC

edge, censorship-pattern governance enforcement, full post-quantum seal enforcement, and independent third-party security review. Each capability is rolled into the model as it ships.

Comparison: Vextaris vs. Industry

Feature	Ethereum	Solana	Vextaris
Mempool	Public gossip	No gossip (leader receives directly)	No mempool (Dragonfly sealed pipeline)
MEV protection	PBS/Flashbots (redistribution)	Jito (redistribution)	Structural elimination (~95%)
Tx ordering	Proposer-controlled	Leader FIFO	Deterministic content-based ordering
Atomic multi-op	No (separate txs)	No (separate txs)	Yes (IntentVM bundles)
Quantum readiness	None	None	Dilithium3 PQ seals + dual-key accounts
Cross-chain model	Lock-and-mint (honeypot)	Lock-and-mint (honeypot)	Escrowed swaps verified by an attester quorum; no unbacked bridged tokens (planned for mainnet)
Peer reputation	Basic scoring	Stake-weighted	PeerGuard (7-factor scoring + ban escalation)
Spam deterrence	EIP-1559 dynamic fees	Priority fees	Gas fees + buffer cap + per-sender limits

---

Risk Management & Recovery

Vextaris implements a forward-only recovery model -- no chain rollbacks, no forks. All corrections happen through governance proposals with validator consensus.

Governance-Based Recovery Tools

Tool	Mechanism	Governance Required
Account Freeze	Block all outbound operations from a compromised account	`AccountFreeze` proposal (67% vote)
Token Freeze	Suspend all transfers of a specific token	`TokenFreeze` proposal (67% vote)
State Correction	Adjust account balances to remediate exploits	`StateCorrection` proposal (67% vote + 100-block delay)
Emergency Mint	Mint tokens to cover verified losses	`EmergencyMint` proposal (67% vote + 100-block delay)

Frozen accounts can still receive tokens, vote on governance proposals, and claim staking rewards. Only outbound operations (transfers, swaps, bridge exits) are blocked. This prevents collateral damage to innocent counterparties while containing threats.

Automated Protections

- **Pool Drain Circuit Breaker** -- Any single swap extracting >50% of a pool's output reserve is automatically rejected. Always-on, no governance needed.
- **Bridge Emergency Pause** -- Single transaction to halt all bridge deposits and new Atomence orders and matches. Orders already in flight can still settle or be refunded. Governance vote required to resume (asymmetric design: fast pause, slow resume).
- **Per-User Bridge Rate Limiting** -- Maximum 100K VXS equivalent per user per day, preventing circuit breaker manipulation.
- **Validator Uptime Tracking** -- Block production tracked on-chain for reputation scoring. Validators who miss blocks lose reputation, affecting delegator trust.

State Preservation

Full state export/import tooling for state-preserving genesis restarts:

- `vexidus state export` -- Deterministic JSON snapshot of all state domains with SHA256 checksum
- `vexidus state import` -- Load exported state into fresh chain at block 0
- `vexidus state verify` -- Cross-validate export against live state

Round-trip verified: all accounts, tokens, validators, pools, and stablecoins preserved through the export-import cycle.

Responsible Disclosure

If you discover a security vulnerability in Vextaris, please report it responsibly:

- **Email**: security@vexidus.com
- **GitHub**: Security advisories (SDK and tooling)

We take all reports seriously and will respond within 48 hours.

Security Audit Report

Updated: November 2026 **Scope**: Full codebase (15+ Rust crates, 624 dependencies) + VexAI Python service **Status**: Internal audit complete. Pre-flight validation (GBI), tombstone mechanism, correlation penalty, and three-tier key separation deployed. External professional audit planned post-funding.

Summary

We ran 9 industry-standard security analysis tools against the entire Vextaris codebase. The results confirm **zero critical or high-severity vulnerabilities in our application code**.

Tool	Purpose	Critical	High	Medium	Low
cargo-audit	Dependency CVE scan	0	0	0	0
cargo-deny	License & policy	0	0	0	0
cargo-geiger	Unsafe code detection	0	0	0	0
gitleaks	Secret scanning (full git history)	0	0	0	0
semgrep	Static analysis (SAST)	0	0	0	0
clippy (pedantic)	Rust linting (max strictness)	0	0	0	0
bandit	Python SAST	0	0	0	0
pip-audit	Python dependency CVE scan	0	0	0	0
Dependency pruning	Attack surface reduction	—	—	—	—

All identified dependency CVEs have been patched. All unused dependencies have been removed.

Key Findings

Zero Unsafe Rust

Vextaris is written entirely in safe Rust. All application crates contain **zero `unsafe` blocks**. The only `unsafe` code in the build is within well-audited, widely-used infrastructure dependencies (tokio, rocksdb, ring/rustls, libp2p).

Tool: `cargo-geiger` — scans the entire dependency tree for unsafe code surface area.

Zero SAST Findings in Production Code

Static analysis with `semgrep` (Rust and security-audit rulesets) found **zero vulnerabilities** across all production Rust code. The only low-severity findings were in non-production test utilities and were reviewed and cleared.

Zero Secret Leaks

`gitleaks` scanned the entire git history and found **no leaked credentials, API keys, SSH keys, or private keys**. Nine detections were all confirmed false positives (well-known public token contract addresses and deterministic testnet keypairs with no real-world value).

All Dependency CVEs Patched

- **Rust**: `cargo-audit` checked 624 crate dependencies against the RustSec Advisory Database. One high-severity CVE was found (`bytes` integer overflow) and immediately patched. Remaining advisories are informational (unmaintained upstream crates with no impact on our usage).
- **Python**: `pip-audit` found 10 CVEs across 6 Python packages. All were upgraded to patched versions on the same day. Re-scan confirmed zero remaining vulnerabilities.

Dependency Pruning

We analyzed every dependency declaration against actual usage in source code and removed 8 unused dependencies to reduce attack surface. Staged dependencies for planned features (ZK-STARK verification, hybrid key exchange) were intentionally retained.

Post-Quantum Ready

Vextaris includes Dilithium3 (NIST post-quantum standard) alongside Ed25519 for signature verification. Accounts can hold both key types, and key rotation is built in — positioning Vextaris ahead of the quantum migration timeline.

Production Hardening

Vextaris enforces strict signature and consensus verification by default. Since October 2026 every validator refuses unsigned transactions; clients sign locally and submit with `vex_submitBundle`. Operators configure their nodes using the flags documented in the Validator Setup Guide.

Tooling

All security tools remain installed on our infrastructure for ongoing monitoring:

Tool	Version	Purpose
cargo-audit	0.22.1	Rust dependency CVE scanning
cargo-deny	0.19.0	License policy enforcement
cargo-geiger	0.13.0	Unsafe code surface detection
gitleaks	8.21.2	Git history secret scanning
semgrep	1.152.0	Static application security testing
clippy	rustup	Rust linting (pedantic + nursery)
bandit	1.9.3	Python SAST
pip-audit	2.9.0	Python dependency CVE scanning

Ongoing Security Practices

Current

- Automated dependency scanning before every release
- Output filtering and guardrails on all public-facing AI services
- Rate limiting on all API endpoints
- Ed25519 signature verification on all transaction bundles -- unsigned transactions are refused by every validator (since October 2026)
- Canonical transaction signing: the signature covers every field that determines what a transaction does (legacy signatures refused from block 33,000,000)
- Cross-chain settlements accepted only with signatures from an attester quorum (Atomence, activating at block 30,655,525)
- Strict-by-default verification of signatures and consensus

Planned

- **CI/CD Integration**: Automated `cargo audit` + `semgrep` on every push
- **Fuzz Testing**: `cargo-fuzz` targets for critical execution paths
- **External Audit**: Professional audit by a top-tier blockchain security firm (Trail of Bits, Zellic, OtterSec, or Halborn)
- **Bug Bounty Program**: Public bounty program via Immunefi or HackerOne
- **Formal Verification**: Consensus finality and bridge logic

---

Test Suite

The Vextaris codebase maintains a comprehensive test suite:

- **354 tests** across all crates
- All tests passing with zero failures
- `cargo check` clean compilation
- Tests cover: state machine operations, token standards, staking, governance, multi-sig, bridge logic, signature verification, address derivation, hash formats, and more

---

Internal Code Audits

In addition to automated tooling, the team performs targeted internal code audits of critical subsystems -- including the state-root accumulator and all state-write paths -- to confirm every state change is tracked consistently and identically across validators. Findings from these reviews are addressed and deployed before they can affect mainnet.

Responsible Disclosure

If you discover a security vulnerability in Vextaris, please report it responsibly:

- **Email**: security@vexidus.com
- **GitHub**: Security advisories (SDK and tooling)

We take all reports seriously and will respond within 48 hours. Please do not disclose vulnerabilities publicly until we've had time to investigate and patch.

Glossary

Quick reference for Vextaris-specific terms, standards, and concepts.

Addresses & Identity

Term	Definition
Vx address	Native Vextaris address format. Full 32-byte encoding with type prefix (~54 chars). Lossless round-trip.
Vx0	Wallet/EOA address prefix (e.g., `Vx0ZbxZq...aQ3TB`)
Vx1	Fungible token mint address prefix
Vx2	NFT collection address prefix
Vx3	Contract/program address prefix
Vx4	Multi-token collection (VSC-55) address prefix
Vx5	Multi-sig account (VSC-88) address prefix
Vx6	Staking pool address prefix
Vx7	DAO (VSC-99) address prefix
Vx8	AMM pool address prefix
Vx9	VNS name address prefix
VxP	VIP program address prefix
0x address	EVM-compatible hex format. Used by MetaMask and `eth_*` RPC methods. Secondary format.
Vxh	Native hash display format. `Vxh` + base58(BLAKE3 32 bytes). ~48 chars. Used for block and transaction hashes.
VNS	Vextaris Name Service. Permanent `.vex` names (e.g., `foundation.vex`). NFT-backed, no renewal fees.

Consensus & Network

Term	Definition
HyperSync	Vextaris consensus protocol. Leader-based BFT with weighted leader rotation and adaptive block timing (~2s).
Vexcel	Attestation DAG layer. Non-leader validators produce attestation blocks referenced via `parent_blocks` in the block header. Fork detection without rollback.
Leader rotation	Stake-weighted deterministic leader selection. Higher stake = more frequent block proposals.
Block	Contains a header + zero or more bundles. Produced by the elected leader, finalized by the network.
Bundle	Atomic multi-operation transaction. One signature covers N operations. All succeed or all revert.
Canonical signature	The current transaction signature scheme. The signature covers every field that determines what the transaction does (sender, operations, gas, validity window, nonce, revealed key, gas sponsor). Accepted since block 18,700,000; the default in current CLI, SDK and VexSpark releases.
Legacy signature	The original transaction signature scheme. Refused from block 33,000,000 -- older client builds must be updated before then.
Signed submission	Every state-changing transaction must be signed client-side (SDK, CLI `--key`, or a wallet) and submitted with `vex_submitBundle`. Validators refuse unsigned transactions.
Operation	A single instruction within a bundle (e.g., Transfer, Swap, MintNft, Stake). 103 discriminants defined.
Dragonfly Stream	Mempoolless transaction pipeline. Transactions flow directly to the leader via P2P (Phase 3). No gossip pool.
Epoch	A period of block production used for validator rotation, reputation scoring, and reward calculations.
Finalization	A block is finalized when the leader produces it and the network accepts it. Single-slot finality.

Execution & Intents

Term	Definition
IntentVM	Intent Virtual Machine (U.S. Patent App. 19/571,463). Declarative execution: 1 intent → 1 signature → N operations.
Intent	A natural-language or structured instruction (e.g., "swap 10 USDC for VXS"). Parsed into a Goal, then decomposed into operations.
Goal	The parsed intent type: Transfer, Swap, Acquire, Liquidate, Rebalance, Stake, Program.
VIP	Vextaris Intent Programs. Declarative on-chain program templates. No compiler, no bytecode, no VM. Steps reference prior step outputs.
Anchor	Universal data attestation. `vex_anchor` stores a hash on-chain in a namespace:key structure. Used for document verification, supply chain proofs.

Token Standards (VexForge)

Standard	Full Name	Purpose
VSC-7	Fungible Token	Standard fungible tokens (like ERC-20)
VSC-8	Stablecoin	Regulated stablecoins with freeze/blacklist, 6 security layers
VSC-20	Staking Pool	Programmable staking pools with time-based rewards, lock periods, per-user caps
VSC-21	NFT	Non-fungible tokens (like ERC-721) with on-chain metadata
VSC-55	Multi-Token	Batch token collections (like ERC-1155) — fungible + non-fungible in one collection
VSC-88	Governance + Multi-Sig	Protocol governance proposals + M-of-N multi-sig accounts
VSC-99	DAO	On-chain DAOs with treasury, proposals, voting, council, snapshot modes
VSC-LAUNCH	Presale	Anti-rug presale escrow with LP lock and refund protection
VSC-REP	Reputation	Two-score validator reputation: production score + public reputation. Used for delegation decisions and display.

Standard	Full Name	Purpose
VSC-SAFE	Safety	Bridge contract heuristic scoring + NFT metadata sanitization (XSS/injection blocking)

Staking & Validators

Term	Definition
Validator	A node that participates in consensus by producing and voting on blocks. Requires minimum 1,000 VXS stake.
Delegation	Staking VXS to a validator without running a node. Delegators earn proportional rewards minus commission.
Self-stake	VXS staked by the validator operator themselves.
Commission	Percentage of block rewards kept by the validator before distributing to delegators. Default: 10% (1000 bps).
Jailing	Temporary removal from consensus for misbehavior (double-signing, excessive missed blocks). Escalating penalties.
Production score	0-100 score based on uptime, block production, and tenure. Used for reputation display and delegation decisions.
Streak	Consecutive blocks produced without a miss. Tracked for reputation display.
Unbonding period	21-day waiting period after unstaking before VXS can be withdrawn.

Bridge & Cross-Chain

Term	Definition
VexBridge	Cross-chain verification layer (U.S. Patent App. 63/987,929). Proves foreign-chain events through the attester quorum. Inbound deposits are paused on testnet; for mainnet its planned role is verification, not minting.

Term	Definition
Atomence	Cross-chain escrow swaps (U.S. Patent App. 63/998,160). The user escrows funds on Vextaris and names the asset and minimum amount to receive on the destination chain; a registered market maker pays on that chain and settles with attester signatures.
Attester	An independent signer that checks a payment on a foreign chain (only once final) and signs a statement naming the order, recipient, asset and amount paid. A settlement needs a quorum of registered attesters; the attester set changes only by on-chain governance.
Attester quorum	The number of attester signatures a settlement needs: at least half of the registered attesters for Ethereum and Solana, at least 70% for Bitcoin, never fewer than 3.
Market maker (MM)	A registered Atomence participant who fills orders by paying on the destination chain. Requires a VXS bond of at least 1,000 VXS (tiers: 1K / 10K / 100K VXS). Earns 0.15% of the settled amount.
Settle window	The 1-hour deadline that starts when a market maker matches an order. If the MM does not settle in time, the user reclaims the full escrow plus 10% of the MM's bond.
Minimum receive	The minimum amount of the destination asset an Atomence order requires. The chain checks the attested payment is at least this amount.
"Nothing comes to Vextaris that can't leave"	Adopted mainnet principle: every token is either native or issued by a named party obliged to redeem it 1:1. No unbacked bridged copies of foreign tokens (planned for mainnet).
Bridge mapping	Registry entry linking a source-chain token to its Vextaris representation (testnet).

Atomence market-maker registration, the settle window, minimum receive and attester-signed settlement apply from block 30,655,525.

Infrastructure

Term	Definition
VexVisor	Binary upgrade governance system. On-chain upgrade proposals trigger coordinated binary swaps at a target block height.
ElasticState	3-tier storage engine: Hot (DashMap in-memory), Warm (SSD RocksDB), Cold (ZSTD compressed).
RocksDB	Embedded key-value database used for all on-chain state (accounts, tokens, validators, pools, NFTs).

Term	Definition
LtHash	Lattice-based homomorphic hash accumulator for state root computation. O(1) per update.
Seed node	Non-validating node that relays blocks and peer connections. Does not participate in consensus.

Economics

Term	Definition
VXS	Native token of Vextaris. 9 decimals. 1.618 billion total supply (golden ratio).
nanoVXS	Smallest unit. 1 VXS = 1,000,000,000 nanoVXS.
Gas price	Fixed at 10 nanoVXS per gas unit. Typical transfer costs ~$0.0002.
Block reward	Time-weighted emission. Split: 85% validator (density-weighted), 15% Foundation.
Foundation Treasury	System address `Vx0Y6upiBWdopdA8JHWGvzG8bPeprVQGV1bE8iuQ1LuV7CuPFMs1j` (EVM form `0x...03`). Receives 15% of block rewards, 20% of transaction fees, and the protocol share of Atomence fees.
Density multiplier	Block reward multiplier based on transaction count: 0.5x (empty) to 2.0x (full). Incentivizes validators to include transactions.

Ecosystem Products

Product	Description
VexScan	Block explorer at vexscan.io
VexSpark	Native wallet at wallet.vexspark.com
VexForge	Token studio at vexforge.xyz
VexiDEX	DEX interface at vexidex.com
VexSwap	Testnet faucet at vexswap.xyz

Product	Description
VexFi	Financial infrastructure at vexfi.com
Vexalus	Supply chain platform at vexalus.com
DaimyoDAO	DAO launcher at daimyodao.com
VexAI	AI assistant at vexai.xyz
Bushido Passport	Universal gamer identity — soulbound profiles, cross-game achievements

Vextaris Blockchain Technical Whitepaper

Next-Generation Layer 1 Infrastructure Version: 3.0 **Date:** July 2026 (updated November 2026)
Status: Beta Testnet Live | Mainnet Target 2026

Executive Summary

Vextaris is a Layer 1 blockchain built from the ground up to solve real problems: quantum vulnerability, cross-chain fragmentation, developer complexity, and bridge security. It runs a live testnet across 7 validators on 4 continents, relaunched in July 2026 on BFT consensus (fresh genesis). The consensus core is built on Informal Systems' Malachite — a production Rust implementation of the Tendermint BFT algorithm — with Vextaris's deferred-execution, post-quantum signing, and direct-to-leader delivery layered on top.

Vextaris is built on **eight pillars** that work together as a unified system:

1. **IntentVM** (U.S. Patent App. 19/571,463) -- 1 intent, 1 signature, N operations. Declarative execution.
2. **HyperSync Consensus + Vexcel** -- Leader rotation with weighted selection, adaptive attestation DAG for global latency fairness, jailing, and reputation scoring.
3. **VexBridge** (U.S. Patent App. 63/987,929) -- Cross-chain verification: proves foreign-chain events through an attester quorum. Under the adopted mainnet principle -- *nothing comes to Vextaris that can't leave* -- its planned role is verification, not minting.
4. **VSA v2** -- Smart accounts with Ed25519 + Dilithium3 dual-signature quantum hardening. Includes VNS (`.vex` name service) and VIP (Vextaris Intent Programs) as identity and programmability extensions.
5. **VexForge** -- Ten token standards (VSC-7, VSC-8, VSC-21, VSC-55, VSC-88, VSC-LAUNCH, VSC-REP, VSC-20, VSC-99, VSC-SAFE) with protocol-level anti-rug protection, on-chain governance, and native DAOs.
6. **VexiDEX** -- Native on-chain AMM with constant-product pools and LP token management.
7. **ElasticState** -- Intelligent 3-tier storage engine (Hot/Warm/Cold) with LtHash homomorphic state root accumulator. State grows without degrading performance.

8. **Atomence** (U.S. Patent App. 63/998,160) -- Cross-chain escrow swaps: registered market makers pay on the destination chain, settlement is proven by an attester quorum, and every order has a guaranteed refund path.

Current Status: Solo founder-built from concept to live testnet in 16 months. 15+ Rust crates, 124 operation discriminants (disc 0-123), 110+ RPC endpoints, 354 passing tests, 16+ deployed frontends. On-chain upgrades delivered via VexVisor. Live testnet across 7 validators on 4 continents, relaunched on BFT consensus (fresh genesis, July 2026).

Table of Contents

1. The Problem
2. The Eight Pillars
3. Architecture Overview
4. Performance and Benchmarks
5. Security Model
6. Token Economics
7. Ecosystem and Applications
8. Roadmap
9. Competitive Analysis
10. Team and Development

---

1. The Problem

Scalability vs. Decentralization

Ethereum processes ~15 TPS with 12-minute finality. Bitcoin manages ~7 TPS. "Fast" L1s like Solana achieve 65K TPS but face periodic outages and centralization concerns. No chain has proven it can sustain high throughput while remaining reliable and decentralized.

Quantum Vulnerability

NIST estimates quantum computers could break current blockchain cryptography by 2030-2035. Over $3 trillion in cryptocurrency assets rely on ECDSA or Ed25519 alone. No major L1 has implemented post-quantum cryptography at the protocol level alongside classical signatures.

Bridge Security Crisis

$1.1B+ lost in bridge hacks in 2022 alone (Wormhole $320M, Ronin $625M, Nomad $190M). Every major hack targeted the **vault** -- the smart contract holding locked tokens. The vault-based bridge model is fundamentally broken.

Developer and User Complexity

Users must understand gas optimization, transaction construction, and chain-specific tooling. "Swap 100 USDC for ETH" requires multiple transactions, multiple signatures, and MEV awareness on most chains.

---

2. The Eight Pillars

Pillar 1: IntentVM (U.S. Patent App. 19/571,463)

The Problem: Traditional blockchains require users to specify exact execution steps. A simple "swap then stake" requires 2 transactions, 2 signatures, and 2 consensus rounds.

The Solution: IntentVM accepts a single intent -- expressed as structured JSON or natural language -- and decomposes it into the optimal sequence of native operations, executed atomically in one block slot.

```
User: "swap 100 VXS for USDC and stake the rest"

IntentVM decomposes:
  1. Swap (VXS -> USDC via VexiDEX)
  2. Stake (remaining VXS to validator)

Result: 1 intent, 1 signature, 2 operations, 1 block slot
```

Key Advantage -- 1 Intent, 1 Signature, N Operations:

On Solana or Ethereum, "swap then stake" = 2 transactions, 2 signatures, 2 consensus rounds. On Vextaris, it is 1 bundle, 1 signature, 1 round. This gives Vextaris 3-5x effective throughput per user action compared to per-transaction chains.

Multi-Input Atomic Swaps: IntentVM processes multi-input operations atomically. "Swap 10 USDC and 10 USDT for VXS" executes both conversions in one block slot with unified slippage protection -- on other chains this requires 2 separate transactions with slippage risk between them.

Provenance: IntentVM was first built and proven as an on-chain program on Solana, with real transaction proofs from a working prototype. It was then rebuilt as a core protocol feature on Vextaris -- not a theoretical design, but a port of working code.

Supported Goals:

- **Swap** -- Token exchange via VexiDEX pools with auto-routing through intermediate tokens
- **Stake** -- Delegate VXS to validators
- **Provide Liquidity** -- Add to AMM pools
- **Transfer** -- Send tokens to any address or VNS name
- **Bridge** -- Cross-chain operations, verified through the attester quorum
- **Composite** -- Atomic multi-step (e.g. swap + stake + provide liquidity in one intent)
- **Rebalance** -- Portfolio rebalancing to target allocations (e.g. "50% VXS, 30% USDC, 20% ETH")
- **DCA / Recurring** -- Scheduled repeated execution (e.g. "buy 10 VXS every day for 7 days")
- **Conditional / Limit Orders** -- Price-triggered execution (e.g. "swap USDC for VXS if price below 0.15")
- **Synthetic Assets (VSC-71)** -- Tokenized stocks, commodities, and perpetual contracts (licensee interface)
- **Custom** -- VNS registration, future LLM-parsed intents

Intelligent Routing: When no direct liquidity pool exists between two tokens, IntentVM automatically discovers and executes multi-hop routes through intermediate pools, selecting the path with deepest liquidity. A 25% price impact guard protects users from excessive slippage on large orders.

Natural Language in 7 Languages: Users can express intents in English, Spanish, Portuguese, French, German, Italian, or Japanese. The NL parser normalizes non-English input and extracts inline constraints (slippage, deadline, max fee) from any intent.

RPC Endpoints:

- `vex_submitIntent` -- Submit natural language or structured JSON
- `vex_simulateIntent` -- Full dry-run with per-pool quotes, price impact, balance checks, and warnings
- `vex_previewIntent` -- Human-readable wallet confirmation (summary, fee, price impact, route)
- `vex_getExecutionPlan` -- Preview execution steps and gas estimate before submission
- `vex_getIntentStatus` -- Query result by hash
- `vex_getPrice` -- Spot price query with direct and auto-routed pricing, TWAP (time-weighted average price) for oracle-grade reference data

Traditional Flow vs. IntentVM:

Action	Traditional (Ethereum/Solana)	Vextaris IntentVM
Swap + Stake	2 transactions, 2 signatures, 2 blocks, MEV-exposed between steps	1 intent, 1 signature, 1 block, atomic
Swap + Provide LP	2-3 transactions, 2-3 signatures, MEV-exposed between steps	"Swap 100 USDC for VXS and add liquidity to VXS/USDC" -- 1 intent, 1 block
Weekly DCA into 3 tokens	3 recurring approvals, 3 DEX interactions per week, manual rebalancing	"DCA $200 weekly split 50/30/20 into VXS, ETH, USDC" -- 1 intent, automated
Buy tokenized stock	Off-chain broker, T+2 settlement, custody risk	"Buy 10 AAPL with USDC" -- 1 intent, instant settlement, self-custodial

IntentVM Licensing -- Ecosystem Flywheel:

IntentVM is architected as a three-layer system designed for wide adoption: a **patented core engine** (intent parsing, goal decomposition, constraint enforcement), a **ChainAdapter trait** (pluggable interface for any blockchain's token registry, DEX routing, and address format), and **chain-native execution** (atomic operations on the target chain).

This architecture enables a licensing flywheel: wallets, DEXs, and chains can license the patented IntentVM core to offer natural-language intent execution on their own platforms. Each licensee deepens the ecosystem while Vextaris retains the **deepest integration and performance moat** -- IntentVM on Vextaris executes natively within the consensus layer (zero overhead, atomic

guarantees), while licensed deployments operate as an application-layer abstraction. The commercial model: **license widely, but the best experience is always on Vextaris native.**

The IntentVM SDK is available to licensees with adapter implementations for Solana (SPL tokens), EVM chains (ERC-20), and Vextaris (native VSC tokens). Each adapter maps the same natural language to chain-specific operations, enabling "write once, execute anywhere" intent semantics.

VexFi -- Protocol-Level Financial Infrastructure:

VexFi is the financial infrastructure layer of the Vextaris ecosystem. It unifies payments, trading, staking, custody, and portfolio management into a single interface -- built entirely on Vextaris L1 primitives. VNS-aware payments, VexiDEX spot trading, VSC-88 multi-sig custody vaults, and natural-language intents all surface through the VexFi SDK and REST API.

The intent architecture already supports complex financial operations expressed in plain language:

```
"Rebalance my portfolio to 60% staking / 40% liquidity, max 0.7%
slippage"
  → Decomposes into: portfolio query → swap → stake, executed atomically

"DCA $200 weekly into my target allocation"
  → Recurring intent: scheduled rebalancing, no recurring approvals
needed

"Send 100 VXS to alice.vex"
  → VNS resolution + payment, one intent, one signature
```

`vex_getPrice` provides the oracle foundation -- TWAP pricing across VexiDEX pools, with external oracle integration planned for real-world asset pricing (stocks, commodities, forex). This is the data layer that makes portfolio intelligence, limit orders, and synthetic asset pricing possible.

VexFi also surfaces VSC-71 synthetic assets -- tokenized equities, indices, and perpetual contracts -- as they become available. Oracle-priced, collateral-backed, self-custodial. Where TradFi requires a brokerage account with custody risk and T+2 settlement, VexFi delivers the same capabilities with instant settlement and zero counterparty risk.

Payments, trading, staking, and custody vaults are live. Synthetics (VSC-71) and advanced margin are in design phase. See vexfi.com.

Pillar 2: HyperSync Consensus + Vexcel

Architecture: Weighted leader rotation with Byzantine fault tolerance, enhanced by the Vexcel (Vextaris Accelerated Consensus Layer) adaptive attestation DAG. Validators take turns proposing blocks based on stake weight and performance score.

How It Works:

1. Leader selection uses deterministic, verifiable seeded randomness derived from on-chain block and validator-set data
2. Weight = `stake * performance_score` (0.5-1.0 range)
3. Leader proposes a block containing thousands of transactions
4. Block is gossiped to all peers via GossipSub (QUIC transport, TLS 1.3)
5. Validators verify proposer signature (Ed25519) and execute transactions
6. Block finalized after gossip consensus

Vexcel -- Adaptive Attestation DAG:

When the leader is fast, the chain is purely linear -- zero overhead. When the leader is slow (geographic latency, network congestion), non-leader validators produce lightweight **attestation blocks** -- signed, zero-transaction blocks that prove liveness and state agreement without entering the canonical chain. The next leader references attestation hashes in its block header, forming a DAG of liveness proofs.

This solves the geographic fairness problem that plagues traditional BFT chains: before Vexcel, the closest validator to the network center produced 68% of blocks. After Vexcel, all validators produce within a tight band (ideal: even split) regardless of geographic location.

Attestation blocks replace four separate mechanisms (failover, voting, heartbeat, liveness proof) with one unified concept. No rollbacks, no fork-choice confusion, no special cases.

Block Production:

- Block time sub-second to ~1.5s (consensus-set under BFT: round timeout + 2f+1)
- Deterministic BFT finality — a committed block is final, no confirmations
- Thousands of transactions per block

Validator Economics:

- Minimum stake: 1,000 VXS
- Block rewards: Time-weighted (13M VXS/yr Year 1-2). 85% Validator Quality Pool (density-weighted 0.5x-2.0x), 15% Foundation Treasury
- Transaction fees: 80% to block proposer, 20% Foundation Treasury (no burn)
- Commission: 0-50%, configurable per validator

No Slashing -- Jailing Instead: Vextaris does not slash validator stake. Misbehaving validators are jailed via double-sign detection with escalating penalties (1h → 24h → 7d → 30d). This protects delegators from losing funds while still incentivizing uptime. Reputation scoring (7-factor, 100-point scale) affects leader selection weight and future reward multipliers.

Reputation Factors:

Factor	Weight	Measures
Uptime	25 pts	Block production consistency
Commission fairness	15 pts	Reasonable rates, stability
Self-stake	15 pts	Skin in the game
Governance participation	15 pts	Upgrade voting activity
Tenure	10 pts	Time active on network
Delegator trust	10 pts	Stake attracted from others
Reliability	10 pts	Historical jail count, missed blocks

Pillar 3: VexBridge -- Cross-Chain Verification

The Problem: Nearly every major bridge loss came from the same design: a vault holds foreign tokens and a contract mints unbacked copies on another chain. Whoever breaks the vault or fakes the proof takes everything.

The Principle (adopted): "Nothing comes to Vextaris that can't leave." Every token on Vextaris is either **native** (VXS and tokens created on Vextaris) or **issued by a named party obliged to redeem it 1:1** (a chartered stablecoin issuer; later possibly a chartered custodian).

The Role of VexBridge: VexBridge's job is to prove that events happened on a foreign chain. Foreign-chain proof is moving to a quorum of registered **attesters**, each running on separate infrastructure with its own key (testnet: five machines operated by the Vexidus team; independent outside operators planned before mainnet): attesters sign only final events (Ethereum finalized checkpoint, Solana finalized commitment, Bitcoin at least 6 confirmations), and the attester set is changed only by on-chain governance. Atomence settlement uses this quorum from block 30,655,525.

Mainnet model (planned):

- No unbacked bridged or synthetic copies of foreign tokens.
- Major assets (BTC, ETH, SOL, foreign USDC) are not held on Vextaris as wrapped tokens. Users **swap** into a native asset through Atomence (Pillar 8) and back, with escrow and refund guarantees on every leg.
- Stablecoins only from chartered issuers -- no bridge-minted stablecoins.
- VexBridge's role is **verification**, not minting.
- Long-tail foreign tokens are not supported at mainnet.

Testnet status: Inbound VexBridge deposits are **paused** (since 31 October 2026) while attestation moves to the attester quorum. Five attesters run on five separate machines on testnet today; independent outside operators are planned before mainnet. The earlier testnet design -- vaultless burn-and-mint with stateless/limbo resolution and three-tier verification -- is superseded for mainnet by the principle and model described here.

Pillar 4: VSA v2 -- Quantum-Hardened Smart Accounts

Dual-Signature Architecture: Every Vextaris account uses Ed25519 (classical, Solana-compatible) alongside Dilithium3 (NIST-approved post-quantum lattice-based signatures). This provides quantum resistance TODAY while maintaining compatibility with existing Ed25519 wallets.

Address Derivation (Same Key, Three Views):

```
Ed25519 Keypair -> pubkey (32 bytes)
  +---> Vx0:    base58(0x00 + pubkey) -> "Vx0abc..."  (native wallet, 32-
byte lossless)
  +---> 0x:     keccak256(pubkey)[12..32] -> "0x71C7..." (EVM/MetaMask,
20-byte)
  +---> base58: raw pubkey -> "7xKXtg..."              (Solana/Phantom)
       All resolve to the same internal 32-byte account.
```

Vextaris uses type-prefixed addresses for different account types: Vx0 (wallet), Vx1 (token), Vx2 (NFT), Vx3 (contract), Vx5 (multi-sig), Vx7 (DAO), Vx8 (pool), Vx9 (VNS), VxP (VIP program). Each encodes the full 32-byte public key -- lossless, approximately 54 characters.

Key Management:

- AddKey, RemoveKey, RotateKey -- hot-swap signing keys without changing your address
- Guardian recovery -- 3-of-5 trusted guardians can recover access (7-day delay, cancelable)
- No seed phrase required for recovery (protocol-level, not a service)

Key management operations (disc 6-12) are defined in the protocol and live on testnet. Guardian recovery with timelock is functional end-to-end.

Pubkey Revelation: Accounts created by receiving transfers get a placeholder public key. On first send, the wallet includes the real Ed25519 public key alongside the signed bundle. The protocol verifies the public key matches the address, registers the key, and proceeds -- seamless to the user.

Why Not ML-KEM? Dilithium3 is a signature scheme; ML-KEM is a key encapsulation scheme. Blockchains need signatures. Vextaris uses Ed25519 for speed and compatibility, Dilithium3 for quantum hardening, and a dedicated post-quantum block-signature field reserved for future dual-sig enforcement.

VNS -- Vextaris Name Service (Identity Extension):

VNS provides human-readable `.vex` names as an extension of the VSA identity system. Each `.vex` name IS a VSC-21 NFT -- ownership of the NFT IS name resolution. Transfer the NFT, transfer the name. No separate registry update needed.

- Register `chris.vex` -> mints NFT in global VNS collection -> stored as `vns:chris` in RocksDB

- Resolve `chris.vex` -> find the NFT -> the owner's address IS the resolved address
- Pricing: 3-4 chars = 1,000 VXS, 5-7 chars = 100 VXS, 8+ chars = 10 VXS
- RPC: `vex_registerName`, `vex_resolveName`, `vex_reverseResolve`, `vex_getNameInfo`

VNS is a natural complement to VSA -- it adds a human-readable layer to the account system that VSA already manages. What makes Vextaris different is that VNS names compose atomically with IntentVM -- "send 10 VXS to chris.vex and stake the rest" is one intent, one signature.

VIP -- Vextaris Intent Programs (Programmability Extension):

VIP (Vextaris Intent Programs) provides declarative composition templates that replace smart contracts for common patterns. A VIP defines a sequence of native operations with parameters, safety constraints, and governance oversight. No compiler, no bytecode, no VM. VIPs are registered on-chain, scored for safety, and executable via IntentVM.

Pillar 5: VexForge -- Ten Token Standards

VexForge provides protocol-level token creation without smart contracts. All standards are live on testnet.

Standard	Purpose	Operations
VSC-7	Fungible tokens	Create, mint, transfer, burn (disc 1-4)
VSC-8	Regulated stablecoins	6 security layers, epoch rate limiting, multi-sig, guardian freeze (disc 50-57)
VSC-21	NFTs	Create collection, mint, transfer, burn (disc 19-22)
VSC-55	Multi-token (ERC-1155 equivalent)	Create, mint, batch mint/transfer, burn (disc 23-29)
VSC-88	Protocol governance + multi-sig	Parameter changes, treasury, voting, M-of-N multi-sig (disc 60-67)
VSC-LAUNCH	Presale with anti-rug	Create, contribute, finalize, refund (disc 31-36)
VSC-REP	Token reputation scoring	Deterministic 7-factor scoring, grade A+ to F

Standard	Purpose	Operations
VSC-20	Programmable staking pools	Treasury and mint rewards, lock periods, per-user caps (disc 90-96)
VSC-99	DAO program	Token-weighted governance, council actions, treasury management (disc 97-103)
VSC-SAFE	Bridge + NFT safety	Heuristic auto-scoring, governance reclassification, URI sanitization

Anti-Rug Protection (VSC-LAUNCH): Presale funds are held in protocol-level escrow. If the presale fails to meet its goal, contributors can claim refunds directly -- no trust in the creator required. On finalize, liquidity is automatically locked in VexiDEX.

Token Metadata: On-chain mutable metadata (name, symbol, image, social links) with a one-way `LockTokenMetadata` operation (disc 42). Once locked, metadata is permanently immutable -- useful for establishing trust in community tokens.

Pillar 6: VexiDEX -- Native On-Chain AMM

VexiDEX is a constant-product automated market maker built into the protocol (disc 38-41).

- **Pools:** Any VSC-7 token pair. 0.3% swap fee.
- **LP Tokens:** Minted on deposit, burned on withdrawal. Track proportional pool share.
- **Create Pool -> Add Liquidity -> Swap -> Remove Liquidity** -- all native operations.
- **RPC:** `vex_createPool`, `vex_addLiquidity`, `vex_removeLiquidity`, `vex_swap`, `vex_quoteSwap`, `vex_getPool`, `vex_listPools`

VexiDEX integrates with IntentVM -- a user can say "swap 100 VXS for USDC" and IntentVM routes through the optimal pool automatically.

Pillar 7: ElasticState -- Intelligent 3-Tier Storage Engine

The Problem: Every blockchain today treats all state equally. A validator that transacted 5 seconds ago and a wallet that has not moved in 3 years sit in the same data structure, consuming the same resources. As chains grow, state bloat becomes the dominant bottleneck.

The Solution: ElasticState automatically routes account data through three storage tiers based on activity, giving sub-microsecond access to hot data while compressing dormant accounts by up to 90%.

```
+-----------------------------------------------------+
|  HOT TIER -- DashMap (in-memory)                     |
|  Active accounts (last 7 days)                      |
|  Access: <1us | Capacity: 100K entries / 8 GB        |
+-----------------------------------------------------+
|  WARM TIER -- RocksDB (LZ4 compression)              |
|  Recent accounts (7-90 days inactive)               |
|  Access: ~100us | Fast SSD reads                    |
+-----------------------------------------------------+
|  COLD TIER -- RocksDB (ZSTD max compression)         |
|  Dormant accounts (90+ days inactive)               |
|  Access: ~1-10ms | 90% storage reduction            |
|  Includes Merkle proofs for light client verification|
+-----------------------------------------------------+
```

Key Design Decisions:

- **Feature-flagged** (`--elastic-state` CLI flag). Backward-compatible.
- **State root integrity preserved.** The main RocksDB default CF feeds the LtHash accumulator, which produces the state root. ElasticState is a read cache that cannot affect consensus.
- **No state rent.** Vextaris will never charge users to keep their accounts alive.
- **Merkle proofs for cold accounts.** Enables future light client verification.

Pillar 8: Atomence -- Cross-Chain Escrow Swaps (U.S. Patent App. 63/998,160)

The Problem: Moving value between chains usually means trusting a bridge vault or a wrapped token. If the bridge fails, the wrapped token is worth nothing.

The Solution: Atomence swaps value between Vextaris and other chains without wrapped tokens. The user's funds stay in escrow on Vextaris until an attester quorum confirms that a market maker has paid the user on the destination chain. If the market maker does not pay in time, the user gets everything back plus a penalty from the market maker's bond.

The rules below activate at block **30,655,525**.

Flow:

1. **Order:** The user creates an order on Vextaris, escrows the amount, and names the **destination asset** ("native" or the token contract/mint on the destination chain) and the **minimum amount** they must receive.
2. **Match:** A **registered market maker** matches the order. Matching starts a **1-hour settle window**.
3. **Pay:** The market maker pays the user on the destination chain.
4. **Attest:** Each registered attester checks the payment on the destination chain once it is final and signs a statement naming **this order, the recipient, the asset and the amount paid**.
5. **Settle:** With a quorum of attester signatures, the chain re-checks that the amount paid is at least the user's minimum and releases the escrow to the market maker, less fees.
6. **Or reclaim:** If the market maker does not settle within the window, the user reclaims the **full escrow plus 10% of the market maker's bond**.

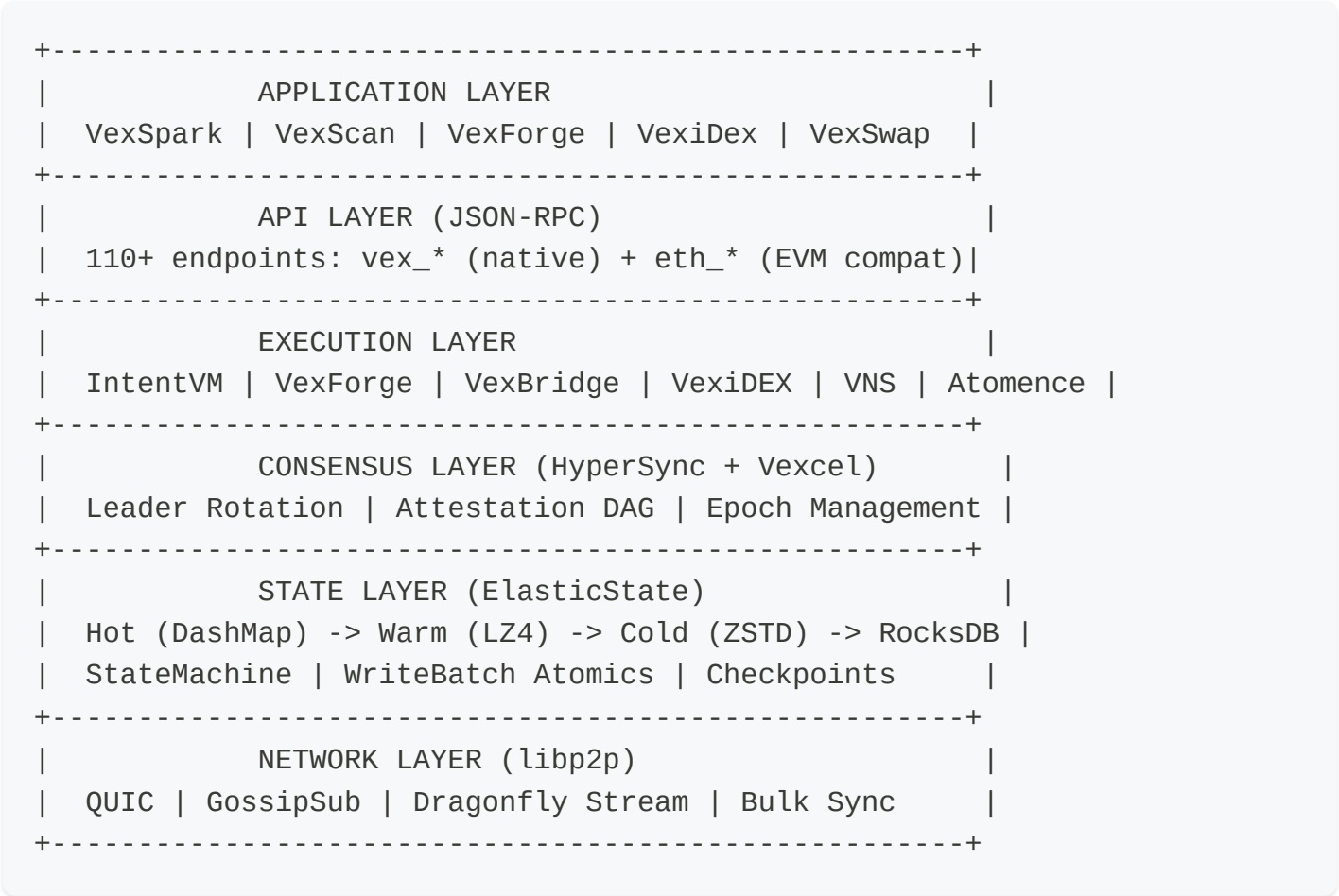
Key Properties:

- **Only registered market makers can match.** Registration requires a VXS bond of at least **1,000 VXS** (tiers: 1K / 10K / 100K VXS). A market maker whose bond falls below the minimum is deactivated.
- **Guaranteed exit:** funds can never be locked by a market maker that walks away.
- **Independent proof of payment:** quorum is at least half of the registered attesters for Ethereum and Solana, at least 70% for Bitcoin, and never fewer than 3 (with five attesters: 3-of-5, Bitcoin 4-of-5). Attesters sign only final payments.
- **One payment settles one order:** a destination-chain transaction cannot be reused.
- **Fees:** 0.30% of the settled amount -- 0.15% to the market maker, 0.15% to the protocol (Foundation) treasury. Nothing is burned.
- **Pause-safe:** when the bridge is paused, Atomence takes no new orders or matches; orders already in flight can still settle or be refunded.
- Orders created before activation without a destination asset and minimum amount cannot be matched afterwards; the user cancels them (if pending) or reclaims them (if matched, after the settle deadline).
- `vex_getAtomenceOrder` returns each order's `dest_asset`, `min_dest_amount` and `settle_deadline`.

---

3. Architecture Overview

System Layers



Implementation

- **Language:** Rust -- 15 crates, memory-safe, zero-cost abstractions
- **Database:** RocksDB with atomic WriteBatch commits (crash-safe, no partial state)
- **Storage:** ElasticState 3-tier engine -- Hot (DashMap, <1us), Warm (RocksDB LZ4), Cold (RocksDB ZSTD). Zero-copy checkpoints every 1K blocks.
- **Networking:** libp2p with QUIC transport (TLS 1.3, multiplexing over UDP)
- **State Root:** LtHash homomorphic accumulator -- O(1) per write, Blake3 finalization once per block (same approach as Solana SIMD-0215)
- **Transaction Model:** Account-based with bundle semantics (atomic multi-operation)

Transaction Lifecycle

Dragonfly Stream mode (direct delivery):

```
1. User submits bundle (signed Ed25519)
      |
2. RPC validates signature + format
      |
3. Post-quantum cryptographic seal created over the transaction
      |
4. Sealed bundle forwarded directly to leader + 2 pipeline backups
      |
5. Leader verifies seals, proposes block (deterministic content-based
ordering)
      |
6. Bundles executed and committed atomically as a single batch
      |
7. State root recomputed and persisted
      |
8. Block gossiped to peers -> verified -> finalized
      |
9. Explorer indexes updated
```

Dragonfly Stream is the sole transaction delivery mechanism. There is no gossip mempool -- transactions are sealed and forwarded directly via P2P request-response. PQ sealing is always on.

Wallet Compatibility

Wallet	Status	RPC Layer	Signature
VexSpark (native)	Live	`vex_*`	Ed25519
MetaMask	Live	`eth_*`	secp256k1/ECDSA
Phantom	Planned	`svm_*`	Ed25519 (same curve)

One Ed25519 keypair works on both Vextaris and Solana. The same key generates three address formats (Vx0, 0x, base58) that all resolve to the same internal account.

---

4. Performance and Benchmarks

Performance (July 2026, BFT Relaunch)

Vextaris provides instant, deterministic BFT finality (a committed block is final — no confirmations) and a high-throughput native execution engine. A sustained-TPS benchmark on the current consensus is in progress; we do not publish throughput numbers we have not re-verified.

Metric	Value
Finality	Deterministic BFT — a committed block is final, typically well under two seconds
Block time	sub-second to ~1.5s (7 validators, 4 continents)
RPC rejection rate	**0%** (zero rejections at submission)

Block propagation uses Borsh binary encoding (5-10x smaller than JSON) with automatic format negotiation for safe rolling upgrades.

Transaction Cost

Operation	Gas	Cost (at $1/VXS)
VXS Transfer	~21,000	~$0.00021
Token Create	~50,000	~$0.0005
Swap	~80,000	~$0.0008
Bridge Deposit	~100,000	~$0.001
Name Registration	~60,000	~$0.0006 + name fee

Gas price: 10 nanoVXS per gas unit. Fees split: 80% to block proposer, 20% to Foundation Treasury.

What We Claim vs. What Is Proven

We do not publish numbers we have not proven on the current consensus. The prior chain's audit-grade load tests (immediate-execution consensus) are retained on the Load Test Results page as clearly-labeled historical measurements, superseded by the relaunch. A re-benchmark on BFT is in progress; the qualitative finality and cost characteristics stated above are current.

Metric	Status
Finality	Deterministic BFT — a committed block is final, typically well under two seconds
Block time	sub-second to ~1.5s (7 validators)
Sustained TPS	Re-benchmark in progress — no number published until re-verified
Transaction cost	~$0.0002 per transfer (see below)

---

5. Security Model

Consensus Security

- **Byzantine Fault Tolerance:** Tolerates up to 1/3 malicious validators
- **Block Proposer Signatures:** Every block is Ed25519-signed by its proposer. Invalid signatures result in instant peer banning.
- **PeerGuard:** Per-peer scoring system. Replay detection (200-hash ring buffer), rate limiting, validator allowlisting. Score drops to 0 = permanent ban.
- **Validator Set from State:** No self-adding. Validators loaded from on-chain staking state only.

Transaction Authentication

- **Signatures required:** Every validator refuses unsigned transactions (since 31 October 2026). Clients sign locally -- SDK, CLI `--key`, or a wallet such as VexSpark -- and submit with `vex_submitBundle`. Convenience write methods that would build a transaction on the node are refused, and the reserved system account cannot be used as a sender from outside.
- **Canonical signing:** The canonical signature covers every field that determines what a transaction does. It is accepted since block 18,700,000 and is the default in current client releases; legacy signatures are refused from block 33,000,000.

MEV Elimination -- Dragonfly Stream

Dragonfly Stream eliminates MEV by removing the mempool entirely. Transactions are sealed with post-quantum Dilithium3 signatures and forwarded directly to the block leader -- no public broadcast, no pool to observe. Combined with deterministic content-based ordering and IntentVM

atomic execution, Vextaris has zero known MEV vectors without requiring BLS/DKG threshold encryption.

MEV Attack	Defense
Front-running	No mempool to observe -- direct delivery to leader
Sandwich attacks	IntentVM atomic execution -- no inter-step window
Proposer reordering	Deterministic blake3 sort -- content-determined order
Censorship	Attestation commitments prove tx existed -- provably detectable
Quantum tx forgery	Dilithium3 PQ seals -- quantum-safe temporal proofs

Quantum Resistance

Algorithm	Purpose	Security Level
Ed25519	Transaction signing, block proposing	Classical (Solana-compatible)
Dilithium3	PQ transaction seals (Dragonfly Stream), block PQ signatures	NIST Level 3 (quantum-safe)
Blake3	Hashing (state root, tx hash, address derivation, seal commitments)	256-bit (quantum-resistant)

Vextaris is quantum-**hardened** today: Dilithium3 is actively used for Dragonfly Stream transaction seals (auto-generated on every validator). When quantum computers threaten Ed25519, Vextaris can activate dual-signature enforcement without a hard fork -- a post-quantum block-signature field is already reserved on every block.

Cross-Chain Security

Major bridge losses (Wormhole $320M, Ronin $625M, Nomad $190M) came from vaults backing unbacked copies, or from proofs that could be faked. Vextaris addresses both:

Risk	Vextaris approach
Unbacked minted copies	Planned for mainnet: no unbacked bridged tokens. Every token is native or redeemable 1:1 by a named issuer.

Risk	Vextaris approach
Faked proof of a foreign event	A quorum of registered attesters, each on separate infrastructure, must sign; attester set changed only by governance.
Reorged or unfinalized payments	Attesters sign only finalized payments (Bitcoin: at least 6 confirmations).
Replayed proofs	One destination payment settles exactly one order.
Counterparty walks away	1-hour settle window, then full refund plus 10% of the market maker's bond.
Emergency	Pause stops new orders and matches; in-flight orders can still settle or be refunded.

No Slashing

Vextaris deliberately avoids slashing. Validators who double-sign are jailed with escalating penalties (1h → 24h → 7d → 30d), not economically punished. This protects delegators and removes the "slashing insurance" overhead that exists on other chains. Bad actors lose opportunity cost (missed rewards) and reputation, not principal. An upgrade grace period (100 blocks) prevents jailing during network upgrades.

---

6. Token Economics

Token Overview

Property	Value
Symbol	VXS
Total Supply	1,618,000,000 (Golden Ratio: 1.618B): 1,553.28M allocated at genesis, the rest issued as block rewards over 10 years
Decimals	9
Inflation	Declining, for 10 years only: ~0.84%/yr in years 1-2, ~0.41% in years 3-5, ~0.24% in years 6-10, **none from year 11**. VXS is never burned

Property	Value
Fee Model	80% to block proposer, 20% Foundation Treasury (no burn)

Why 1.618 Billion?

The golden ratio (1.618...) appears throughout nature and mathematics. It is memorable, distinctive, and provides ample supply for a global network while maintaining meaningful per-token value.

Genesis Allocations

96% of total supply (1,553.28M VXS) is allocated at genesis across 8 accounts:

Account	VXS	%	Status
Public Distribution (`0x...01`)	614,840,000	38%	Circulating (Pre-sale 20% + TGE 18%)
Ecosystem & DeFi (`0x...02`)	242,700,000	15%	Circulating (grants, LP incentives)
Foundation Treasury (`0x...03`)	161,800,000	10%	Circulating (operations)
Team & Founders (`0x...04`)	161,800,000	10%	Locked (4-year vest, 1-year cliff)
Marketing & Community (`0x...05`)	113,260,000	7%	Circulating (airdrops, campaigns)
SAFT Round (`0x...06`)	48,540,000	3%	Locked (12-month vest, 6-month cliff)
Testnet Rewards (`0x...07`)	48,540,000	3%	Circulating (beta testers)
Staking & Strategic (`0x...08`)	161,800,000	10%	Governance-controlled
Genesis Total	**1,553,280,000**	**96%**	**73% circulating at TGE**

Block Reward Schedule

Block rewards are **newly issued VXS**: ~65M over 10 years (~4% of the 1.618B total), on a declining schedule. Annual inflation is about **0.84%** in years 1-2, **0.41%** in years 3-5 and **0.24%** in years 6-10 (each measured against supply at the start of that period), and **zero from year 11**.

Period	Annual Total	Foundation (15%)	Validator Pool (85%)	Per-Block @2s (1.0x density)
Year 1-2	13M VXS/yr	1.95M	11.05M	~0.700 VXS
Year 3-5	6.5M VXS/yr	0.975M	5.525M	~0.350 VXS
Year 6-10	3.9M VXS/yr	0.585M	3.315M	~0.210 VXS
Year 11+	0	0	0	0

Per-block reward is time-weighted: `reward = annual x block_interval / seconds_per_year`. Transaction density multiplier scales the validator portion: 0.5x (empty) to 2.0x (1000+ txs). Foundation receives 15% of block rewards and 20% of transaction fees for sustainable development funding.

Transaction Fees

Solana-competitive pricing:

Parameter	Value
Gas price	10 nanoVXS per gas unit
Typical transfer cost	~~0.00021 VXS~~ ($0.0002 at $1/VXS)
Fee destination	80% to block proposer, 20% Foundation Treasury
Burn mechanism	**None** -- fees are not burned

Staking

Parameter	Value
Minimum stake	1,000 VXS
Unbonding period	21 days
Commission range	0-50%
Max validators	100 (configurable)

Parameter	Value
Epoch duration	1 hour (1,800 blocks)

---

7. Ecosystem and Applications

Live Frontends

Application	URL	Purpose
VexSpark	wallet.vexspark.com	Native Ed25519 wallet (send, receive, tokens, IntentVM) -- signs every transaction in the wallet
VexScan	vexscan.io	Block explorer (blocks, transactions, validators, tokens)
VexForge	vexforge.xyz	Token creation studio (templates, terminal, JSON editor)
VexiDex	vexidex.com	DEX interface (swap, pools, liquidity)
VexSwap	vexswap.xyz	Testnet faucet + IntentVM natural language demo
VexAI	vexai.xyz	AI knowledge assistant (LLaMA 3.1 + ChromaDB RAG)
Vexcel	vexcel.xyz	Vexcel adaptive attestation protocol product site
Atomence	atomence.com	Cross-chain escrow swaps (registered market makers, attester-verified settlement)
VexFi	vexfi.com	Protocol-level financial infrastructure (payments, trading, staking, vaults)
Vexalus	vexalus.com	Supply chain intelligence (shipments, attestation, audit trails)
VNS	vns.name	Vextaris Name Service (.vex domain registration)
DaimyoDAO	daimyodao.com	Native DAO launcher (VSC-99 governance)
VexVisor	vexvisor.com	Validator operations portal (staking, delegation, governance)
IntentVM Demo	intentvm.xyz	Natural language intent execution demo

Developer Tools

- **Rust SDK** (`vexidus-sdk`) -- WalletKeypair, BundleBuilder, IntentBuilder, ValidatorClient, DEX client
- **CLI** (`vexidus`) -- 18 command groups, 100+ subcommands
- **110+ RPC Endpoints** -- Full `vex_*` native API + `eth_*` EVM compatibility
- **VexVisor** -- Upgrade governance module: schedule upgrades via RPC, stake-weighted voting, automatic halt at upgrade height, binary download with SHA256 verification, auto-rollback on crash within 60 seconds. Two modes: seed-authority (auto-approve) and full (67% quorum). All validators run via VexVisor wrapper in production.

How the Pillars Work Together

The eight pillars are not isolated features -- they compose:

Example: "Swap 50 USDC for VXS and stake it with my validator, then send the rest to chris.vex"

1. **IntentVM** parses the natural language into a composite intent (swap + stake + transfer)
2. **VexiDEX** routes the USDC to VXS swap through the best pool
3. **HyperSync** staking operations delegate the VXS to the chosen validator
4. **VNS** resolves `chris.vex` to the recipient's address
5. **VSA v2** verifies the single client-side signature over the entire bundle
6. **HyperSync** includes the bundle in the next block (weighted leader selection)
7. **ElasticState** serves the user's account state from its storage tiers

Leaving Vextaris is a separate, escrowed step: an **Atomence** order pays the user out on another chain, settled only with attester signatures and refundable if the market maker does not settle in time.

All of this happens atomically -- one intent, one signature, one block slot. If any step fails, the entire operation rolls back.

---

8. Roadmap

Completed

Phase	Status	Key Deliverables
Core Architecture	Done	15 Rust crates, StateMachine, RocksDB, account model
Consensus	Done	HyperSync leader rotation, Vexcel attestation DAG, vote signing, epoch management
Token Standards	Done	VSC-7, VSC-21, VSC-55, VSC-LAUNCH, VSC-REP
Staking	Done	9 operations, weighted selection, jailing, commission, reputation
IntentVM	Done	Goal decomposition, atomic execution, NL parser (7 languages), auto-routing, price oracle (TWAP), portfolio intelligence (rebalance, DCA, conditional), VSC-71 synthetic interface, 6 RPC endpoints
VexBridge	Testnet	Three-tier architecture and burn/limbo model built on testnet; inbound deposits paused (Oct 2026) while attestation moves to the attester quorum. Mainnet role: verification, not minting (planned)
ElasticState	Done	3-tier storage (Hot/Warm/Cold), auto-promotion/demotion, checkpoints, feature-flagged
VNS	Done	NFT-backed `.vex` names (VSA identity extension), 4 RPC endpoints
VexiDEX	Done	Constant-product AMM, 7 RPC endpoints
P2P Security	Done	Block proposer signatures, PeerGuard, sync validation
Gas and Fees	Done	Solana-competitive pricing, 80% proposer / 20% Foundation
Explorer	Done	RocksDB indexing, 11 query endpoints
EVM Compat	Done	eth_* RPC translation (MetaMask works)
Frontends	Done	VexSpark, VexScan, VexForge, VexiDex, VexSwap, VexFi, Vexalus, VNS, DaimyoDAO, VexVisor, IntentVM Demo
Load Testing	Done	55K+ txs, 2,856 peak tx/block, 0 crashes, 11 op types proven
Multi-Validator BFT Network	Done	Leader rotation, Vexcel attestation DAG, 24h+ soak test

Phase	Status	Key Deliverables
VexVisor	Done	Upgrade governance, auto-halt, binary download, SHA256 verify, auto-rollback
Dragonfly Stream	Done (Phase 1-3.1)	Mempoolless direct delivery, PQ sealing (Dilithium3), mempool eliminated, direct-to-leader routing
VSC-88 Governance	Done	On-chain governance + multi-sig wallets, 12 RPC endpoints
VSC-8 Stablecoins	Done	18 protocol-native stablecoins, 6 security layers
VSC-SAFE	Done	Bridge contract safety scoring, NFT metadata hardening
VSC-20 Staking Pools	Done	Programmable staking pools, treasury/mint rewards, lock periods, per-user caps
VSC-99 DAOs	Done	Token-weighted governance, council actions, treasury management, global DAO index
Risk Management	Done	Account/token freeze, state correction, circuit breaker, emergency mint
State Preservation	Done	State export/import/verify CLI, round-trip proven (1,390 accounts)
Foundation/Validator Split	Done	15% Foundation / 85% Validator rewards, density-weighted multiplier
Automated Backups	Done	GPG-encrypted weekly backups, cold offload, offsite storage
Atomence Settlement	Done (Phase 1-4)	Cross-chain escrow swaps (U.S. Patent App. 63/998,160). Registered market makers, 1-hour settle window with guaranteed refund, attester-quorum settlement -- activating at block 30,655,525
Signature Enforcement	Done	Unsigned transactions refused network-wide (Oct 2026); canonical signing accepted since block 18,700,000
Attester Set	Done (testnet)	Five governance-registered attesters on five separate machines
Bushido Passport	Done	Universal gamer identity on Vextaris
VIP (Intent Programs)	Done	Declarative on-chain composition templates, safety scoring, governance-managed
Anchor (Data Attestation)	Done	Universal data attestation (BLAKE3-hashed, on-chain proof)

Phase	Status	Key Deliverables
Full-Length Vx Addresses	Done	Type-prefixed (Vx0-Vx9, VxP), 32-byte lossless, backward-compatible
Pending Nonce Tracker	Done	Concurrent RPC safety, 71 methods updated
GovernableParams Wiring	Done	On-chain parameter governance
Failover Timing Optimization	Done	5s base + 3s stagger failover, leader skip recovery (5 blocks), block time ~5s → ~2s
Tier 1 TPS Scaling	Done	50K bundles/block, 100ms/250ms adaptive floor, --benchmark-mode (100K/100ms)
RocksDB Optimization	Done	Periodic flush, bloom filters, 512MB cache, HashMap index, 64MB write buffers
ElasticState Activation	Done	3-tier storage (Hot/Warm/Cold) active on all validators
Borsh Wire Protocol	Done	Dual-format P2P encoding (reads both, writes Borsh), 5-10x smaller blocks
VIDA Identity Protocol	Done	Sovereign digital identity, 18 credential types, W3C VC export, DID resolver

Current Phase: Beta Testnet

- VexVisor E2E upgrade pipeline (auto-download, auto-swap across all validators)
- Validator rewards program (testnet reputation → mainnet VXS airdrops)
- Public GitHub releases
- 3rd-party security audit preparation

Next Phases

Phase	Focus
Beta Hardening	VexVisor E2E pipeline, SVM adapter (Phantom), monitoring dashboard, governance activation
Performance Tier 2-3	WriteBatch pipelining, transaction lanes, block sharding, state commitment deferral

Phase	Focus
Security Audits	External audit of core protocol, bridge, cryptography
Mainnet Prep	Dynamic gas, feature gate activation, key separation enforcement, legacy-signature cutoff (block 33,000,000), independent outside attester operators
Mainnet Launch	Genesis block, validator onboarding, exchange listings

---

9. Competitive Analysis

What Is Unique About Vextaris

No other L1 combines all eight:

Feature	Vextaris	Ethereum	Solana	Aptos	Sui
Intent-based execution	IntentVM	No	No	No	No
Quantum hardening	Ed25519 + Dilithium3	No	No	No	No
MEV elimination	Dragonfly Stream (no mempool)	Flashbots (partial)	Jito (partial)	No	No
No unbacked wrapped tokens	Native or 1:1-redeemable only (planned for mainnet)	Wrapped	Wrapped	Wrapped	Wrapped
3-tier state engine	ElasticState (Hot/Warm/Cold)	Single trie (300M+ accounts)	Flat store (500+ GB, rent)	Single tree	Single store
Protocol-level anti-rug	VSC-LAUNCH	No	No	No	No
Native AMM	VexiDEX	Uniswap (dApp)	Raydium (dApp)	No	Cetus (dApp)
No slashing	Jailing only	Slashing	Slashing	Slashing	No slash

Feature	Vextaris	Ethereum	Solana	Aptos	Sui
No state rent	ElasticState compresses, never evicts	Debated since 2017	Account rent required	No rent	Storage rebates
Cross-chain settlement	Atomence (escrowed, attester-verified, guaranteed refund)	No	No	No	No
Native DAOs	VSC-99 protocol-level	Snapshot (off-chain)	Realms (dApp)	No	No
Multi-wallet compat	Vx/MetaMask/Phantom	MetaMask	Phantom	Petra	Sui Wallet

Competitive Moats

1. **IntentVM (U.S. Patent App. 19/571,463)** -- Legal protection for intent-based execution with a proven licensing pathway. The three-layer architecture (patented core + ChainAdapter + native execution) means competitors can license IntentVM for their own chains, growing the ecosystem while Vextaris retains the deepest native integration. Every licensee validates the standard; the best performance is always on Vextaris.
2. **No Unbacked Bridged Tokens** -- "Nothing comes to Vextaris that can't leave." Planned for mainnet: assets are native or redeemable 1:1 by a named issuer, and cross-chain value moves through escrowed Atomence swaps verified by an attester quorum -- removing the vault that every major bridge hack targeted.
3. **Quantum-Ready Infrastructure** -- Ed25519 + Dilithium3 dual-signature is live in code. When quantum migration is urgent, Vextaris has a 5+ year head start.
4. **ElasticState -- No State Rent, Ever** -- Ethereum has debated state expiry since 2017 without resolution. Solana charges rent. Vextaris solves state bloat at the infrastructure level: inactive accounts are compressed, not evicted. Users never pay to exist.
5. **Dragonfly Stream -- Zero-MEV by Design** -- Most chains try to mitigate MEV with partial solutions (Flashbots, order-flow auctions, encrypted mempools). Vextaris eliminates the root cause by removing the mempool entirely. Transactions are PQ-sealed and forwarded directly to the block leader. No pool to observe = no MEV.
6. **Eight-Pillar Integration** -- Features that are separate dApps on other chains (DEX, presales, reputation, DAOs, cross-chain settlement) are protocol-level on Vextaris, enabling atomic composition via IntentVM.

---

10. Team and Development

Founder

CJ -- Founder and CEO

- Built Vextaris blockchain solo from concept to live testnet in 16 months
- Designed and implemented IntentVM (U.S. Patent App. 19/571,463, proven on Solana first)
- Founded Vexidus Corporation (Delaware C-Corp, est. 2019)
- MBA student (ongoing)

Development Structure

Entity	Role
Vexidus Labs Ltd (United Kingdom)	Engineering, infrastructure, operations
Vexidus Corporation (Delaware C-Corp)	IP, patents, investor capital, strategy
Vextaris Stiftung (Switzerland, planned)	Foundation governance, community treasury, grant programs

Codebase

- **Language:** Rust (15+ crates)
- **Tests:** 354
- **RPC Endpoints:** 110+
- **CLI Subcommands:** 100+
- **Deployed Frontends:** 16+
- **Lines of Rust:** 65,000+
- **On-Chain Upgrades:** 32+ via VexVisor (zero manual intervention)
- **SDK & tooling:** github.com/vexidus-labs/vexidus-sdk (the core node repository is private during the beta programme)

Open Source

- Core blockchain: Open source (post-beta)

- IntentVM: U.S. Patent App. 19/571,463 (proprietary until patent grant)
- SDK, CLI, frontends: Open source

Appendix A: Technical Specifications

Validator Requirements

Resource	Minimum	Recommended
CPU	4 cores	8+ cores
RAM	8 GB	16+ GB
Storage	100 GB SSD	500 GB NVMe
Network	100 Mbps	1 Gbps
Stake	1,000 VXS	5,000+ VXS

Network Parameters

Parameter	Value
Chain ID	1618032 (`0x18b070`)
Block time	sub-second to ~1.5s (consensus-set under BFT)
Max transactions/block	50,000 default, 100,000 benchmark (configurable via `--max-txs-per-block`)
Transaction delivery	Dragonfly Stream (direct, PQ-sealed)
Bundle TTL	120 seconds
P2P transport	QUIC (TLS 1.3 + multiplexing, UDP)
Gas price	10 nanoVXS/gas (configurable)
Auto-checkpoint interval	Every 1,000 blocks

Parameter	Value
Checkpoints retained	Last 5

ElasticState Parameters

Parameter	Default	CLI Flag
Hot cache entries	100,000	`--elastic-hot-entries`
Hot cache memory	8 GB	`--elastic-hot-memory-mb`
Hot to Warm threshold	7 days inactive	--
Warm to Cold threshold	90 days inactive	--
Demotion interval	Every 100 blocks (~20 min)	--
Warm compression	LZ4 (fast)	--
Cold compression	ZSTD (max, ~90% reduction)	--
Feature flag	Off (backward-compatible)	`--elastic-state`

State Root: LtHash Accumulator

Parameter	Value
Algorithm	LtHash (homomorphic lattice hash)
Accumulator size	1024 x u16 (2048 bytes)
Update cost	O(1) per state write
Finalization	Blake3 over the accumulator once per block
Migration	One-time full DB scan on first boot, persisted to storage
Crash safety	Included in the atomic write batch, rollback on failure

Parameter	Value
Verification	Optional cross-check against a full DB scan

Per-CF RocksDB Compression

Column Family	Compression	Rationale
Blocks, Explorer, NFTs	ZSTD	Immutable data, 40-60% better compression
Accounts, Validators, Attestations, Pools	LZ4	Fast reads, frequent updates

~35% total disk reduction on archive nodes compared to uniform compression.

Address Formats

Format	Example	Used By
Vx0 (native wallet)	`Vx0abc...xyz` (~54 chars)	VexSpark, native SDK
Vx1-Vx9, VxP (typed)	`Vx1abc...`, `Vx7abc...`, `VxPabc...`	Token, DAO, VIP, etc.
0x (EVM)	`0x71C7...`	MetaMask, eth_* RPC
System (32-byte hex)	`0x000...0001`	Genesis accounts, internal

Appendix B: Resources

Resource	URL
Wallet	wallet.vexspark.com
Explorer	vexscan.io
Token Studio	vexforge.xyz
DEX	vexidex.com

Resource	URL
Faucet	vexswap.xyz
Financial Infra	vexfi.com
Supply Chain	vexalus.com
Name Service	vns.name
DAO Launcher	daimyodao.com
Intent Demo	intentvm.xyz
GitHub	github.com/vexidus-labs/vexidus-sdk (SDK & tooling; core node repo private during beta)

Document Version: 3.0 **Last Updated:** July 2026 **Status:** Beta Testnet Live

Vextaris: Eight Pillars. One Chain. No Compromises.

Copyright 2026 Vexidus Corporation. All rights reserved.