LP-103: Omni-Chain Explorer Single-Binary Block Explorer with E2E PQ Encrypted Streaming

Zach Kelling

Lux Industries

March 2026

LP-103: Omni-Chain Explorer
Single-Binary Block Explorer with E2E PQ Encrypted Streaming

Zach Kelling
Lux Architecture Team

March 2026

Abstract

We present the Lux Explorer, a single-binary omni-chain block explorer that replaces the traditional multi-service EVM explorer architecture with a unified Go implementation. The explorer indexes an arbitrary number of heterogeneous blockchains concurrently—including EVM, DAG, linear, DEX, FHE/threshold, post-quantum, and zero-knowledge chain types—with per-chain SQLite storage isolation and end-to-end post-quantum encrypted streaming backups to S3 via the Replicate protocol. We demonstrate that a single statically-linked binary implements the `/v1/explorer/` API while reducing operational complexity from six containers and three languages (Elixir, Rust, JavaScript) to one, eliminating the need for managed PostgreSQL, Redis, or any external services. The system scales from a single development chain to 200+ concurrent chains for white-label platform deployments.

Contents

1 Introduction

Block explorers are critical infrastructure for any blockchain network, providing transaction transparency, contract verification, and developer tooling. The dominant open-source explorer, the traditional multi-service EVM explorer stack, requires multiple services, PostgreSQL, Redis, and typically six Docker containers for a production deployment. This operational complexity is a significant barrier for new chains and white-label deployments.

We present a clean-room reimplementation that achieves full EVM explorer coverage while fundamentally simplifying the architecture:

- **Single binary**: One statically-linked Go executable replaces all prior explorer services
- **Embedded storage**: SQLite WAL mode with per-chain isolation (no PostgreSQL)
- **Embedded frontend**: Next.js static export served via `go:embed`
- **PQ encrypted backups**: Continuous WAL streaming to S3 with ML-KEM-768 + X25519
- **Omni-chain**: Native support for 9 chain types and 100+ external chains
- **White-label**: Zero branding in code, runtime-configurable

2 Architecture

2.1 Single Binary Design

The explorer binary contains three major subsystems running concurrently:

1. **Indexer goroutines**: One per chain, each writing to its own SQLite database
2. **API server**: HTTP server (Base) reading from all chain databases, serving `/v1/explorer/{chain}/*` endpoints
3. **Replication**: Per-chain WAL streaming to S3 with PQ encryption

The frontend (Next.js static export) is embedded in the binary via Go's `embed` directive and served at the root path.

2.2 Per-Chain Storage Isolation

Each chain receives an independent directory containing a SQLite database (WAL mode) and a BadgerDB key-value store:

```
{data_dir}/{chain_slug}/
  query/indexer.db     # SQLite: blocks, txs, tokens, contracts
  kv/                  # BadgerDB: hash->data, height->block
```

Properties:

- Zero write contention between chains (each is sole writer to its SQLite)
- 8 concurrent readers per chain (API layer, WAL mode)
- Independent backup and restore per chain
- Drop a chain by deleting its directory

2.3 Dual-Layer Storage

Definition 2.1 (Unified Store)**.** *The storage layer combines two complementary backends:*

- ***KV Layer*** *(BadgerDB): $O(1)$ lookups by hash, height, or composite key. Prefix-isolated namespaces:* `blk:`, `vtx:`, `edg:`, `tx:`, `st:`, `meta:`, `idx:`.
- ***Query Layer*** *(SQLite): Complex queries with SQL, joins, aggregations, full-text search. WAL mode for concurrent reads.*

The indexer performs dual-writes to both layers. Reads prefer the KV layer for direct lookups and fall back to the Query layer for filtered/sorted/aggregated queries.

3 Chain Support

3.1 Native Chain Types

The explorer natively indexes nine Lux chain types:

Chain	Consensus	Indexed Data
C-Chain (EVM)	Linear	Blocks, transactions (type 0–3), internal traces, tokens (ERC-20/721/1155), smart contracts, DeFi protocols, NFT marketplaces
P-Chain	Linear	Validators, delegators, staking rewards, subnet management
X-Chain	DAG	UTXOs, atomic swaps, multi-asset transfers, vertices, edges
D-Chain (DEX)	EVM	CLOB orderbook, perpetuals, AMM pools, liquidations
A-Chain (AI)	DAG	Attestations, model registration, inference results
B-Chain (Bridge)	DAG	Cross-chain transfers, proof validation, MPC custody
Q-Chain (PQ)	DAG	Post-quantum finality proofs, lattice signatures
M-Chain (MPC)	DAG	Threshold signing ceremonies (CGGMP21 / FROST / Corona-gen), key shares, bridge custody — per LP-134
F-Chain (FHE)	DAG	TFHE keygen, encrypted-EVM compute, decryption proofs — per LP-134
Z-Chain (ZK)	DAG	Zero-knowledge proofs, shielded transfers, nullifier tracking
K-Chain (KMS)	DAG	PQ key material, certificate lifecycle, HSM bridge operations

Table 1: Native chain indexing capabilities

3.2 External Chain Support

For bridge, MPC, and threshold operations, the explorer indexes 100+ external chains:

- **EVM**: Ethereum, Polygon, Arbitrum, Optimism, Base, BSC, Avalanche, +30 more
- **Solana**: Mainnet, Devnet (Metaplex, Magic Eden, Raydium, Jupiter protocols)
- **Bitcoin**: Ordinals, Runes, BRC-20, Stamps, Atomicals
- **Cosmos**: Hub, Osmosis, Injective, dYdX, Celestia, Sei
- **Move**: Aptos, Sui
- **Other**: NEAR, Tron, TON, Substrate/Polkadot

4 E2E Post-Quantum Encrypted Streaming Backups

4.1 Threat Model

Traditional database backups (pg_dump, WAL-G) produce plaintext or GPG-encrypted archives. GPG relies on RSA or classical ECDH, both vulnerable to quantum computers running Shor's algorithm. A harvest-now-decrypt-later adversary could capture today's encrypted backups and decrypt them when quantum computers become available.

4.2 Cryptographic Construction

We employ a hybrid post-quantum encryption scheme using the `age` file encryption format [2] with the ML-KEM-768 extension:

Definition 4.1 (Hybrid PQ Encryption)**.** *For each WAL segment, the encryption proceeds as follows:*

1. *Generate ephemeral keypair* $(ek_{ml}, dk_{ml}) \leftarrow$ *ML-KEM-768.KeyGen*()
2. *Generate ephemeral keypair* $(ek_x, dk_x) \leftarrow$ *X25519.KeyGen*()
3. *Encapsulate:* $(c_{ml}, ss_{ml}) \leftarrow$ *ML-KEM-768.Encaps*$(pk_{recipient})$
4. *Key agree:* $ss_x \leftarrow$ *X25519*$(dk_x, pk_{recipient})$
5. *Derive file key:* $k \leftarrow$ *HKDF-SHA256*$(ss_{ml} \| ss_x, info)$
6. *Encrypt payload: STREAM construction with ChaCha20-Poly1305, 64 KB chunks*
7. *Zero ephemeral keys:* $dk_{ml} \leftarrow 0, dk_x \leftarrow 0$

Proposition 4.1 (Security)**.** *The hybrid scheme is IND-CCA2 secure if either ML-KEM-768 or X25519 is secure. Formally, an adversary must break* both *algorithms to recover plaintext. This provides defense-in-depth: classical security from X25519 and quantum resistance from ML-KEM-768.*

4.3 Streaming Protocol

1. SQLite writes produce WAL frames on disk
2. Replicate monitors the WAL file via `fsnotify`
3. New frames are packaged into immutable LTX (Log Transaction) segments
4. Each LTX segment is encrypted with a fresh ephemeral keypair
5. Encrypted segments are uploaded to S3 with per-chain path isolation
6. Replicate manages checkpointing to ensure all data is streamed before flush

Invariant 4.1 (Completeness)**.** *No WAL frame is checkpointed (flushed to main SQLite file) before it has been successfully streamed to S3. This is enforced by disabling SQLite auto-checkpoint (`_auto_checkpoint=0`) when Replicate is active.*

4.4 Point-in-Time Restore

Each LTX segment is identified by a monotonic transaction ID (TXID). Restore proceeds by:

1. Downloading LTX segments from S3 up to the target TXID
2. Decrypting each segment with the recipient's private key
3. Replaying LTX frames onto a fresh SQLite database
4. Resulting database is byte-identical to the original at that TXID

5 Comparison with Prior Art

Dimension	Traditional	Explorer (this work)
Runtime	Elixir BEAM + 4 Rust services	Single Go binary
Database	PostgreSQL (managed)	SQLite WAL (embedded, per-chain)
Cache	Redis	BadgerDB (embedded)
Containers	6	1
Languages	Elixir + Rust + JavaScript	Go (+ optional Rust staticlib)
Backup encryption	None / GPG (RSA)	E2E PQ (ML-KEM-768 + X25519)
Restore granularity	Full dump	Point-in-time (any TXID)
Multi-chain	EVM only	9 native types + 100+ external
GraphQL	Graph Node + subgraphs	G-Chain native + embedded schema
Dev setup	Docker Compose + PostgreSQL + Redis	`go run ./cmd/explorer`
Min memory	4–8 GB	256 MB
White-label	Env vars + code forks	Runtime flags only

Table 2: Comparison with traditional multi-service explorer architecture

6 Scalability

Each chain runs as an independent goroutine with its own storage. Resource usage scales linearly:

Chains	RAM	CPU	Use Case
1	256 MB	1 core	Single-chain dev
15	1 GB	4 cores	Full Lux native ecosystem
30	3 GB	8 cores	Lux + major bridge chains
100+	10 GB	16 cores	Exchange / aggregator
200+	20 GB	32 cores	White-label platform

Table 3: Scaling characteristics

7 Implementation

The implementation comprises approximately 20,000 lines of Go across 22 packages, with 431+ unit tests achieving full EVM explorer coverage. The Elixir test factory (1,931 lines) has been

ported to Go for regression testing.

Key packages:

- `evm/`: Full EVM indexer (2,330 lines) with all EVM explorer features
- `explorer/`: API layer with 30 explorer endpoints + 35 tests
- `storage/`: Dual-layer SQLite + BadgerDB with build-tag backends
- `{x,a,b,q,t,z,k}chain/`: Native DAG chain adapters
- `multichain/`: 100+ external chain parallel indexer
- `rustffi/`: Optional CGO bindings to Rust smart-contract-verifier

8 Related Work

LP-102 [3] defines the encrypted SQLite replication standard used by the backup system. The G-Chain (GraphVM) provides native cross-chain GraphQL queries as described in the Lux node architecture. The ZAP wire protocol [4] is used for high-performance node-to-indexer communication.

References

[1] "Open-source EVM block explorers: a survey of multi-service architectures," 2024.

[2] F. Valsorda, "age: a simple, modern and secure file encryption tool," https://age-encryption.org

[3] Lux Architecture Team, "LP-102: Encrypted SQLite Replication Standard (age + Replicate)," 2026.

[4] Lux Architecture Team, "ZAP: Zero-Copy Application Protocol," 2025.