

Distributed Storage Architecture: Zaxonlite Integration for Dynamic Policies, Replicated Reputation, and Incident Forensics

STATE	INTENDED STATUS
published	Published
CREATED	AUTHORS
2026-09-07	Sibuna Contributors <team@sibuna.local>
LAST UPDATED	
2026-10-03	
DISCUSSION	
Specifies and records the implementation of Sibuna's Edge surface: the embedded Zaxonlite store (replicated SQLite on paxos-zig) for cluster-wide dynamic policies, IP reputation, and forensic incident search, the read-copy-update engine slot that keeps the request path free of database access, the lock-free incident ring, and campaign clustering of attack payloads.	
LABELS	
storage, zaxonlite, paxos, sqlite, distributed	

Status of This Memo

This document is an internal Sibuna discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered SID documents are part of the permanent project record.

Contents

Abstract	1
Introduction	1
The SID Lifecycle	1
Tooling and Numbering Workflow	1
Creating a Draft	1
Promoting a Draft	1
Listing Discussions	1

Engineering Principles and Code Standards	1
Hard Structural Limits	1
Core TigerStyle Priorities	1
Elm-Style Diagnostic and Error Reporting	1
The Three Elements of an Elm-Style Error	1
Concrete Diagnostic Example	1
Implementation Architecture	1
Recoverable Errors vs Invariant Assertions	1
Compilation Targets and Verification	1
Conclusion	1
Decision	1
Request contract	1
Authentication and work	1
Policy and memory	1
Measurement gates	1
Release toolchain	1
Native platforms and release boundaries	1
Remaining limits	1
1. Abstract	9
2. Introduction	9
3. The SID Lifecycle	9
4. Tooling and Numbering Workflow	10
4.1. Creating a Draft	10
4.2. Promoting a Draft	10
4.3. Listing Discussions	10
5. Engineering Principles and Code Standards	10
5.1. Hard Structural Limits	10
5.2. Core TigerStyle Priorities	11
6. Elm-Style Diagnostic and Error Reporting	11
6.1. The Three Elements of an Elm-Style Error	11
6.2. Concrete Diagnostic Example	11
6.3. Implementation Architecture	12
6.4. Recoverable Errors vs Invariant Assertions	12
7. Compilation Targets and Verification	12
8. Conclusion	12
9. Decision	9
10. Request contract	10
11. Authentication and work	11
12. Policy and memory	11
13. Measurement gates	11
14. Release toolchain	12
15. Native platforms and release boundaries	12
16. Remaining limits	12
Decision summary	1
Data model	1
Evaluation order	1
Built-in policy	1
Policy file	1
Dynamic policies	1

Verification	1
Milestones	1
Review corrections (2026-09-07)	1
17. Decision summary	9
18. Data model	9
19. Evaluation order	10
20. Built-in policy	10
21. Policy file	11
22. Dynamic policies	12
23. Verification	12
24. Milestones	12
25. Review corrections (2026-09-07)	12
Context and problem statement	1
Engine choice: automata and tokenizers, not regular expressions	1
Design	1
Byte-class pre-scan	1
26. Context and problem statement	9
27. Engine choice: automata and tokenizers, not regular expressions	9
28. Design	10
28.1. Byte-class pre-scan	10
Context and motivation	1
Zaxonlite foundations	1
Build and configuration	1
Schema	1
Hot-path isolation	1
Read-copy-update engine slot	1
Incident ring	1
Storage thread tick	1
Campaign clustering	1
Forensics	1
Product boundary	1
Delivery record	1
Verification	1
Open items	1
Review corrections (2026-09-07)	1
29. Context and motivation	9
30. Zaxonlite foundations	10
31. Build and configuration	10
32. Schema	10
33. Hot-path isolation	11
33.1. Read-copy-update engine slot	11
33.2. Incident ring	11
33.3. Storage thread tick	11
34. Campaign clustering	12
35. Forensics	12
36. Product boundary	12
37. Delivery record	12
38. Verification	12
39. Open items	13

40. Review corrections (2026-09-07)	13
Abstract	1
Introduction and Adversary Model	1
Setting	1
Axioms	1
Cost calculus	1
Part A: The Hashcash Tier	1
Part B: Proof of Sequential Work	1
Why sequential work	1
The construction	1
Theorems	1
Why not the alternatives	1
Implementation and calibration	1
Part C: Stateless Challenges and Symmetric Tokens	1
Pseudorandom functions and message authentication	1
Stateless challenge identifiers	1
Session tokens	1
Part D: Rate Limiting by the Generic Cell Rate Algorithm	1
Part E: Robin Hood Hashing for the Spent Set	1
Part F: Linear-Time Inspection	1
Multi-pattern matching	1
The semantic layer	1
Vectorised matchers	1
Part G: Read-Copy-Update Engine Slots	1
Part H: Load-Adaptive Difficulty	1
Part I: Campaign Clustering by Feature Hashing	1
Verification and Benchmark Gates	1
Security Considerations	1
Open Problems	1
References	1
41. Abstract	9
42. Introduction and Adversary Model	10
42.1. Setting	10
42.2. Axioms	10
42.3. Cost calculus	11
43. Part A: The Hashcash Tier	11
44. Part B: Proof of Sequential Work	12
44.1. Why sequential work	12
44.2. The construction	12
44.3. Theorems	13
44.4. Why not the alternatives	14
44.5. Implementation and calibration	15
45. Part C: Stateless Challenges and Symmetric Tokens	15
45.1. Pseudorandom functions and message authentication	15
45.2. Stateless challenge identifiers	16
45.3. Session tokens	17
46. Part D: Rate Limiting by the Generic Cell Rate Algorithm	17
47. Part E: Robin Hood Hashing for the Spent Set	18
48. Part F: Linear-Time Inspection	19

48.1. Multi-pattern matching	19
48.2. The semantic layer	20
48.3. Vectorised matchers	20
49. Part G: Read-Copy-Update Engine Slots	21
50. Part H: Load-Adaptive Difficulty	21
51. Part I: Campaign Clustering by Feature Hashing	22
52. Verification and Benchmark Gates	22
53. Security Considerations	23
54. Open Problems	23
55. References	24
Abstract	1
Introduction and Motivation	1
Operator Workflows	1
Design Principles	1
Don't make me think	1
Fast judgement: System 1	1
Careful judgement: System 2	1
Elegance	1
Principle audit by page	1
Terminology and Scope	1
Goals and Non-Goals	1
Goals	1
Non-Goals	1
Architecture Overview	1
Modules and directories	1
Process model and the isolation contract	1
Startup from the command line	1
The Serve Kernel	1
Authentication and Authorization	1
The Telemetry Pipeline	1
The sampler	1
The incident tap	1
The challenge funnel	1
Traffic sampling and top-k rankings	1
Data-plane changes this record requests	1
Cluster aggregation	1
The Real-Time Protocol	1
Data Model	1
GeoIP	1
Live earth globe on the landing page	1
Cluster Management	1
The Interface Module	1
Pages	1
Wireframes	1
The Build Pipeline	1
Security Considerations	1
Performance Budget	1
Delivery Plan	1
Verification	1

Resolved Design Decisions	1
Operational Contracts and Trade-offs	1
Retained observations	1
Rule observations	1
Management consistency	1
Browser and peer state	1
Acceptance and Open Questions	1
Query and authentication observation bounds	1
Launch scope (2026-09-11)	1
Post-launch completion (2026-09-11)	1
Interface budget review (2026-09-11)	1
References	1
56. Abstract	9
57. Introduction and Motivation	10
58. Operator Workflows	10
59. Design Principles	11
59.1. Don't make me think	11
59.2. Fast judgement: System 1	12
59.3. Careful judgement: System 2	13
59.4. Elegance	13
59.5. Principle audit by page	13
60. Terminology and Scope	14
61. Goals and Non-Goals	15
61.1. Goals	15
61.2. Non-Goals	15
62. Architecture Overview	16
62.1. Modules and directories	16
62.2. Process model and the isolation contract	17
62.3. Startup from the command line	18
63. The Serve Kernel	19
64. Authentication and Authorization	20
65. The Telemetry Pipeline	21
65.1. The sampler	21
65.2. The incident tap	22
65.3. The challenge funnel	23
65.4. Traffic sampling and top-k rankings	23
65.5. Data-plane changes this record requests	24
65.6. Cluster aggregation	24
66. The Real-Time Protocol	24
67. Data Model	26
68. GeoIP	28
68.1. Live earth globe on the landing page	29
69. Cluster Management	30
70. The Interface Module	30
70.1. Pages	31
71. Wireframes	33
72. The Build Pipeline	43
73. Security Considerations	44
74. Performance Budget	45

75. Delivery Plan	46
76. Verification	47
77. Resolved Design Decisions	48
78. Operational Contracts and Trade-offs	49
78.1. Retained observations	49
78.2. Rule observations	49
78.3. Management consistency	50
78.4. Browser and peer state	50
79. Acceptance and Open Questions	51
79.1. Query and authentication observation bounds	51
79.2. Launch scope (2026-09-11)	51
79.3. Post-launch completion (2026-09-11)	52
79.4. Interface budget review (2026-09-11)	53
80. References	53
Abstract	1
Introduction and Motivation	1
Terminology and Scope	1
Problem Statement and Architectural Challenges	1
The Gap in the Current Codebase	1
The Asymmetric Challenges of AI Bot Identification	1
System Invariants and Design Principles	1
Detailed Design	1
1. The Multi-Tier Bot Classification Engine	1
1.1 Provider and Bot Taxonomy	1
1.2 Comprehensive AI Bot Signatures Database	1
1.3 Sub-Microsecond Radix CIDR Verification	1
1.4 Behavioral and Structural Heuristics	1
2. High-Efficiency Streaming Analytics: Algorithms, Mathematics, and Storage Decoupling	1
2.1 The Analytics Dilemma: Why Traditional OLAP Fails Edge WAFs	1
2.2 Axiomatic Foundations of Edge Analytics	1
2.3 Contention-Free Cache-Line Striped Accumulation	1
2.4 Bounded Stochastic Sampling and Poisson Ring Dynamics	1
2.5 Space-Saving Heavy-Hitters Algorithm for Path Scraping	1
2.6 Micro-RRD Pre-Aggregated Rollups and Zaxonlite Storage Decoupling	1
2.7 Long-Term Telemetry Egress Strategy	1
3. Operator Console Dashboard Interface	1
3.1 Operator Screen Wireframes	1
3.2 Visual Components	1
3.3 Granular Tabular Components	1
3.4 Actionable Policy Integration	1
4. Elm-Style Operator Error and Diagnostic Reporting	1
Verification and Testing Matrix	1
Security Considerations	1
Rollout and Operational Plan	1
References	1
81. Abstract	9
82. Introduction and Motivation	9
83. Terminology and Scope	10

84. Problem Statement and Architectural Challenges	11
84.1. The Gap in the Current Codebase	11
84.2. The Asymmetric Challenges of AI Bot Identification	12
85. System Invariants and Design Principles	12
86. Detailed Design	13
86.1. 1. The Multi-Tier Bot Classification Engine	13
86.1.1. 1.1 Provider and Bot Taxonomy	13
86.1.2. 1.2 Comprehensive AI Bot Signatures Database	14
86.1.3. 1.3 Sub-Microsecond Radix CIDR Verification	15
86.1.4. 1.4 Behavioral and Structural Heuristics	16
86.2. 2. High-Efficiency Streaming Analytics: Algorithms, Mathematics, and Storage Decoupling	16
86.2.1. 2.1 The Analytics Dilemma: Why Traditional OLAP Fails Edge WAFs	16
86.2.2. 2.2 Axiomatic Foundations of Edge Analytics	17
86.2.3. 2.3 Contention-Free Cache-Line Striped Accumulation	17
86.2.4. 2.4 Bounded Stochastic Sampling and Poisson Ring Dynamics	18
86.2.5. 2.5 Space-Saving Heavy-Hitters Algorithm for Path Scraping	19
86.2.6. 2.6 Micro-RRD Pre-Aggregated Rollups and Zaxonlite Storage Decoupling . . .	20
86.2.7. 2.7 Long-Term Telemetry Egress Strategy	21
86.3. 3. Operator Console Dashboard Interface	22
86.3.1. 3.1 Operator Screen Wireframes	22
86.3.2. 3.2 Visual Components	24
86.3.3. 3.3 Granular Tabular Components	25
86.3.4. 3.4 Actionable Policy Integration	25
86.4. 4. Elm-Style Operator Error and Diagnostic Reporting	26
87. Verification and Testing Matrix	26
88. Security Considerations	26
89. Rollout and Operational Plan	27
90. References	27
Abstract	1
Introduction	1
Terminology and Scope	1
Problem Statement	1
Goals and Non-Goals	1
Goals	1
Non-Goals	1
State of the Art	1
Protocol Rules	1
Detailed Design	1
The decoder	1
Admission: decoding into the head buffer	1
Forwarding	1
Cost	1
Verification and Testing	1
Security Considerations	1
Rollout and Operational Plan	1
Implementation Record (2026-10-02)	1
References	1
91. Abstract	9

92. Introduction	9
93. Terminology and Scope	10
94. Problem Statement	10
95. Goals and Non-Goals	10
95.1. Goals	10
95.2. Non-Goals	11
96. State of the Art	11
97. Protocol Rules	11
98. Detailed Design	12
98.1. The decoder	12
98.2. Admission: decoding into the head buffer	13
98.3. Forwarding	14
98.4. Cost	15
99. Verification and Testing	15
100. Security Considerations	15
101. Rollout and Operational Plan	16
102. Implementation Record (2026-10-02)	16
103. References	17

Revision note (2026-09-07)

Revised after the implementation review of 2026-09-07, at which point the storage layer existed only as a design. All three phases are now implemented in `apps/sibuna/src/persistent.zig` against the official `insanai/zaxonlite` release (v0.7.0 since 2026-09-14). Zaxonlite 0.7.0 fixed the cluster trim regression that wedged members under sustained replicated load (upstream issue 10) and cut its journal, TRIM, identity, and wire formats without migration, so data directories from earlier builds are recreated rather than upgraded. This record describes the code as built and lists what remains unexercised.

For Sibuna 0.2.0, the same Zaxonlite and Paxos 0.7.0 library sources are included in `vendor/` with explicit Zig 0.17 compatibility changes. Original archives, package hashes and source file digests are recorded in `vendor/provenance.json`; MIT licenses remain with the sources. The library-only build graph excludes the unused terminal UI, preserves the SQLite and OpenSSL settings, and changes neither durable formats nor database schema. Compatibility is qualified through Sibuna's replay and cluster tests, rather than an edited package cache.

29. Context and motivation

A single Sibuna instance loads policy from a file and keeps its challenge, rate-limit, and ban state in local memory. Operating a fleet of edge nodes adds three requirements: policy changes that take effect everywhere without restarts, threat intelligence that propagates from the node that observed an attack to every other node within milliseconds, and durable, searchable forensics for the payloads that were blocked. Existing designs delegate this to external PostgreSQL and Redis deployments or to a proprietary edge platform. Sibuna embeds **Zaxonlite**, a replicated SQLite built on the `paxos-zig` Multi-Paxos library, and keeps all of it out of the request path.

30. Zaxonlite foundations

1. **Replicate the bytes, not the SQL.** A transaction executes once on the leader; the SQLite write-ahead-log page images are what consensus replicates, so nondeterministic SQL is safe and replicas converge to byte-identical files.
2. **The journal is the truth.** Every write is framed, checksummed, and fsynced before consensus proceeds; the SQLite file is a materialised cache rebuildable from the durable anchor plus the journal suffix.
3. **Embedded.** Zaxonlite compiles into the Sibuna binary from build.zig.zon (<https://github.com/insanai/zaxonlite/archive/refs/tags/v0.7.0.tar.gz>, hash pinned). The single-node Node needs no transport; the cluster Embedded facade adds a TCP listener, peers, and either mTLS or the loopback development PSK.

31. Build and configuration

Option	Effect
-Dstorage=true (default)	Links SQLite (with FTS5 and sqlite-vec), paxos-zig, and Zaxonlite; the binary links libc.
-Dstorage=false	Pure in-memory daemon; --data-dir is refused with StorageDisabled.
-Dcluster=true	Enables the Embedded facade and links OpenSSL 3 for Zaxonlite's mutual TLS.
--data-dir <path>	Opens (or creates) the node data directory and starts the storage thread.
--cluster-node <id>	This member's id; non-zero switches from Node to Embedded.
--cluster-listen <host:port>	This member's cluster endpoint.
--cluster-peer id@host:port[/role]	A peer, repeatable up to 8; role defaults to data-voter.
--cluster-secret-file <path>	Shared PSK for the loopback development transport (allow_psk_only_loopback).
--cluster-tls-cert/key/ca <path>	Per-node certificate, key, and cluster CA for production TCP.
--storage-poll-ms <ms>	Storage thread cadence (default 500).

32. Schema

```
CREATE TABLE sibuna_meta (key TEXT PRIMARY KEY, value TEXT NOT NULL);
CREATE TABLE policies (
    id TEXT PRIMARY KEY, name TEXT NOT NULL, priority INTEGER NOT NULL DEFAULT
100,
    path_pattern TEXT, ua_pattern TEXT, action TEXT NOT NULL, difficulty INTEGER,
algorithm TEXT,
    header_matchers TEXT, cidr_matchers TEXT,          -- JSON object / JSON array
    weight INTEGER NOT NULL DEFAULT 0, enabled INTEGER NOT NULL DEFAULT 1,
    created_at INTEGER NOT NULL, updated_at INTEGER NOT NULL);
CREATE TABLE ip_reputation (
    ip_or_cidr TEXT PRIMARY KEY, reputation_score INTEGER NOT NULL, banned_until
INTEGER,
    trigger_rule TEXT, hits INTEGER NOT NULL DEFAULT 1, last_seen INTEGER NOT
NULL);
```

```
CREATE TABLE security_incidents (
    id INTEGER PRIMARY KEY, -- node_id << 40 | sequence
    node_id INTEGER NOT NULL, client_ip TEXT NOT NULL, user_agent TEXT NOT NULL,
    method TEXT NOT NULL, path TEXT NOT NULL, violation_category TEXT NOT NULL,
    offending_payload TEXT NOT NULL, campaign_id INTEGER, recorded_at INTEGER NOT
NULL);
CREATE VIRTUAL TABLE incidents_fts USING fts5(path, offending_payload,
    content='security_incidents', content_rowid='id');
CREATE VIRTUAL TABLE incidents_vec USING vec0(item_id INTEGER PRIMARY KEY,
    embedding float[64] distance_metric=cosine, embedding_coarse bit[64]);
```

Incident ids are node-scoped (`node_id << 40 | sequence`) so members insert concurrently with no id coordination; the sequence is recovered from the table at startup.

33. Hot-path isolation

33.1. Read-copy-update engine slot

Workers never touch the database. AppState holds an atomic pointer to an EngineSlot (engine, cache-line-aligned reader count). A request pins the slot with `acquireEngine`: load the pointer, increment the slot's reader count, and re-load the pointer; if it changed, decrement and retry. The re-check closes the window in which a writer could have swapped and observed zero readers between the load and the increment. The storage thread rebuilds the *spare* slot (database policies followed by file/default rules, then reputation prefixes into the trie), publishes it with an atomic swap, and spins until the old slot's reader count reaches zero before it becomes the new spare. Two engine buffers, sized in the benchmark metadata, therefore serve an unbounded sequence of policy changes.

33.2. Incident ring

WAF denials and honeypot hits copy a fixed-size record (address, User-Agent, method, path, category, first 512 bytes of the payload, timestamp) into a Vyukov bounded MPSC queue of 512 slots; a full queue drops the newest record rather than blocking a response. The storage thread is the single consumer.

33.3. Storage thread tick

Every `--storage-poll-ms`, persist at most 32 records in one transaction, including incident, FTS, vector, and honeypot reputation writes. The nearest-campaign query executes inside that transaction and sees earlier records in the same batch. A per-issuer cursor in `sibuna_meta` guards every effect; the cursor advances atomically with the data. The exact pending SQL is retained after an uncertain reply, making retries idempotent without one network read per incident. Queue admission itself is not durable.

The thread then reads the trigger-maintained policy revision and rebuilds when it changes or a loaded reputation entry expires, even if incident persistence failed. Each slot owns its own arena until its readers drain. A committed ban propagates when a healthy member next reloads; quorum recovery and backlog can increase latency beyond the polling interval.

Production counters expose persisted incidents, committed batches, dropped queue entries and failed write attempts. The queue holds 512 records plus one pending batch of at most 32. Incident issuer IDs are limited to 23 bits so `issuer << 40 | sequence` stays within SQLite's positive signed integer range; sequence exhaustion fails explicitly rather than wrapping.

34. Campaign clustering

Each payload is embedded as a 64-dimensional unit vector by the hashing trick over byte trigrams (digits folded, case folded, signed feature hashing, L2 normalisation). Before insert, a cosine nearest-neighbour query against `incidents_vec` returns the closest earlier incident; if its distance is at most 0.35 the new incident joins that incident's `campaign_id`, otherwise it starts a campaign of its own. Two SQL injection variants that differ only in target column and literal values land in one campaign; a cross-site scripting payload does not. No model is downloaded and nothing allocates on a request thread.

35. Forensics

`Persistent.searchIncidents(match, limit)` runs an FTS5 MATCH over path and payload joined to the incident rows, ordered by rank. There is no HTTP administration endpoint yet; operators query the data directory with `zaxon sql --data <dir>` or read `current.db` directly.

36. Product boundary

Gate provides admission, policy and local flood controls. Shield adds payload inspection. Distributed edge deployment adds replicated policy and reputation to either surface. No vendor feature-parity claim is made; coverage and limitations are defined by the code and tests.

37. Delivery record

Phase 1: Standalone embedded mode delivered

Node opened from `--data-dir`, schema migration, dynamic policy reload, reputation bans, incident persistence with FTS5, forensic search API.

Phase 2: Consensus replication delivered

Embedded facade behind `-Dcluster=true`, member list from `--cluster-*` flags, loopback PSK or mTLS transport, node-scoped incident ids, cluster-wide bans through `ip_reputation`. The external `benchmarks/distributed.py` harness exercises three real members, shared sessions, issuer-bound challenge verification, replicated bans, and one-member loss.

Phase 3: Payload clustering delivered

Feature-hashed trigram embeddings stored in `incidents_vec`, cosine nearest-neighbour campaign assignment at insert time.

38. Verification

The storage test (`apps/sibuna/src/persistent.zig`, run by `zig build test` when storage is enabled) opens a node in a temporary directory, verifies that a file policy survives the initial rebuild, inserts a `policies` row with header and CIDR matchers and a `banAddress` call, ticks once, and checks that the rebuilt engine applies the rule with its difficulty and algorithm and denies the banned address; it then pushes two SQL-injection variants and a honeypot hit through the hook, ticks, and checks the FTS search, the campaign assignment (two campaigns for three incidents), and the reputation upsert. Unit tests cover the ring under concurrent producers, the embedding,

peer-spec parsing, and SQL quoting. A manual run of the release daemon recorded WAF denials and a honeypot ban in `current.db` with 12 MB resident memory.

39. Open items

- Automated multi-node tests (leader failover, ban propagation latency).
- An authenticated HTTP administration API for policies and incident search.
- Retention and archival policy for `security_incidents`.

40. Review corrections (2026-09-07)

Startup retries idempotent initialization while a cluster leader settles; incident writes are not blindly retried because an ambiguous transport failure may already have committed them. INSERT/UPDATE/DELETE triggers maintain a revision for both dynamic tables, including multiple updates with the same timestamp. A failed rebuild does not acknowledge the revision. Expiry of a loaded reputation entry independently triggers a rebuild, so time-based bans disappear without another write. Dynamic rules precede generic file/default rules; capacity exhaustion fails the reload rather than silently dropping security rules. Each engine slot has its own arena, and all publication/pinning operations use sequential consistency.

--cluster-node binds challenge authentication to an issuer; token keys remain shared. Route challenge fetching and verification to that member. Rate limits and spent sets remain local, and spent state is not durable across restarts. Replication is off the request path, but snapshot pinning, incident copying/enqueue, local limits and production metrics are not free. The external load harness adds no server-side timing instrumentation. See `benchmarks/results/distributed-latest.json` for local results and their limits. A healthy HTTP endpoint alone does not prove quorum health. WAN partitions and sustained forensic-write saturation are separate test targets.

Response rule names are copied into a bounded buffer before the snapshot is released, so a slow origin does not hold the reader counter. Persistent rule strings remain owned by the corresponding slot arena until readers drain.