LVS — Developer Guide & API Documentation (EN)

Version 1.0 — Final, Production-Ready Developer Reference

---

1. Purpose of This Document

This Developer Guide defines the **official interfaces, APIs, message formats, and development standards** required to build applications, integrations, tools, or node implementations for the **LVS Autonomous Value Layer**.

It provides: - full module documentation, - API specifications, - message structures, - runtime behavior requirements, - development patterns, - compliance rules, - recommended SDK structure.

This document is intended for: - core LVS developers, - integration partners, - researchers, - external SDK builders.

---

2. Developer Requirements

To work with LVS, developers should understand: - distributed systems fundamentals, - asynchronous communication, - basic vector math, - LVS drift consensus model (DBC), - LVS state model (VU, TC, DC).

No cryptography, no blockchain, no identity systems required.

---

3. LVS Module Architecture

The LVS Node Runtime exposes the following modules:

1. **Transport Module** — communication engine
2. **Entropy Module** — entropy vector generation & normalization
3. **Drift Module** — applies DBC logic
4. **Shard Module** — manages partial state storage
5. **VaultGuard Module** — enforces invariants
6. **Runtime Scheduler** — manages cycles

Each module has its own API.

---

4. Transport API

The transport layer sends and receives **Entropy Packets (EP)** and **State Diff Messages (SDM)**.

4.1 sendEP(ep)

```
sendEP(ep: EntropyPacket) → void
```

Broadcasts an entropy packet to peers.

4.2 receiveEP()

```
receiveEP() → EntropyPacket | null
```

Returns the latest entropy packet or null.

4.3 sendSDM(sdm)

```
sendSDM(sdm: StateDiffMessage) → void
```

Sends a state diff to peers.

4.4 receiveSDM()

```
receiveSDM() → StateDiffMessage[]
```

Returns all newly received SDMs.

4.5 discoverPeers()

```
discoverPeers() → PeerList
```

Discovers and returns reachable peers.

---

5. Entropy Module API

Generates and normalizes entropy.

5.1 generateEntropy()

```
generateEntropy() → float[]
```

Returns a vector of random entropy values.

5.2 normalizeEntropy(E)

```
normalizeEntropy(E: float[]) → float[]
```

Ensures entropy stays within stable bounds.

5.3 getNodeLoad()

```
getNodeLoad() → float
```

Returns a load coefficient from 0.0 to 1.0.

---

6. Drift Module API

Applies Drift-Based Consensus.

6.1 driftFromEntropy(ep)

```
driftFromEntropy(ep: EntropyPacket) → DriftVector
```

Computes entropy-driven drift.

6.2 driftFromDiffs(sdmList)

```
driftFromDiffs(list: StateDiffMessage[]) → DriftVector
```

Computes peer-based drift.

6.3 applyDrift(D)

```
applyDrift(D: DriftVector) → void
```

Updates the local shard according to drift.

6.4 computeLocalDiff()

```
computeLocalDiff() → StateDiffMessage
```

Generates a diff after drift is applied.

---

7. Shard Module API

Manages distributed partial state.

7.1 loadShard()

```
loadShard() → Shard
```

Loads shard from memory or storage.

7.2 storeShard(shard)

```
storeShard(s: Shard) → void
```

Persists the updated shard.

7.3 mergeDiff(dv)

```
mergeDiff(dv: DiffVector) → void
```

Merges peer-generated diffs.

7.4 rebalanceShards()

```
rebalanceShards() → void
```

Ensures shard redundancy across nodes.

---

8. VaultGuard API

Enforces safety constraints.

8.1 validateDrift(D)

```
validateDrift(D: DriftVector) → boolean
```

Returns false if drift violates invariants.

8.2 correctDrift(D)

```
correctDrift(D: DriftVector) → DriftVector
```

Clamps drift to safe bounds.

8.3 detectAnomaly(dv)

```
detectAnomaly(dv: DiffVector) → boolean
```

Detects extreme or malicious diffs.

8.4 triggerRecovery()

```
triggerRecovery() → void
```

Activates recovery mode.

---

9. Runtime Scheduler API

Controls full node execution.

9.1 startNode()

```
startNode() → void
```

Initializes modules and enters active mode.

9.2 runCycle()

```
runCycle() → void
```

Executes one drift cycle.

9.3 enterLowPower()

```
enterLowPower() → void
```

Reduces cycle frequency.

9.4 shutdown()

```
shutdown() → void
```

Closes runtime safely.

---

10. Message Definitions

10.1 Entropy Packet (EP)

```
EntropyPacket = {
  entropy_vector: float[],
  timestamp: int,
  node_load: float
}
```

10.2 State Diff Message (SDM)

```
StateDiffMessage = {
  shard_id: int,
  diff_vector: float[],
  drift_weight: float,
  cycle_id: int
}
```

---

11. Node Integration Examples

11.1 JS/TS (Browser Node)

```
import { startNode } from "lvs-runtime";

startNode();
```

11.2 Go (Server Node)

```
node := lvs.NewNode()
node.Start()
```

11.3 Rust (High-performance Node)

```
let mut node = lvs::Node::new();
node.run();
```

---

12. Compliance Rules

An implementation is LVS-compliant only if it: - supports EP & SDM formats, - implements drift correctly, - enforces VaultGuard invariants, - manages shards, - tolerates packet loss, - supports redundancy.

Non-compliant nodes must not connect to the main network.

---

13. Recommended SDK Structure

```
/transport
/entropy
/drift
/shards
/vaultguard
/runtime
/utils
/examples
/tests
```

---

14. Versioning and Upgrades

LVS upgrades follow: - backwards-compatible extensions, - optional feature flags, - soft updates via drift cycles, - no hard forks.

---

15. Conclusion

This Developer Guide provides a complete reference for building LVS-compatible nodes, tools, and applications. It ensures consistency, security, and stability across the entire ecosystem.