# Converge Ledger

## The Memory of the System

**Distributed, Append-Only Runtime Substrate**

*For Distributed Systems Engineers & Architects*

# The Problem

## Large-Scale Agentic Systems Need Memory

**Context:**

- AI agents execute complex, multi-step workflows
- Reasoning is expensive (LLM calls = seconds, not milliseconds)
- Human-in-the-loop gates create unpredictable pauses

**Constraints:**

- Agents crash. Workflows pause. Progress must survive.
- Multiple observers need real-time visibility
- Recovery must be fast (not recompute-from-scratch)

# The Naive Solution

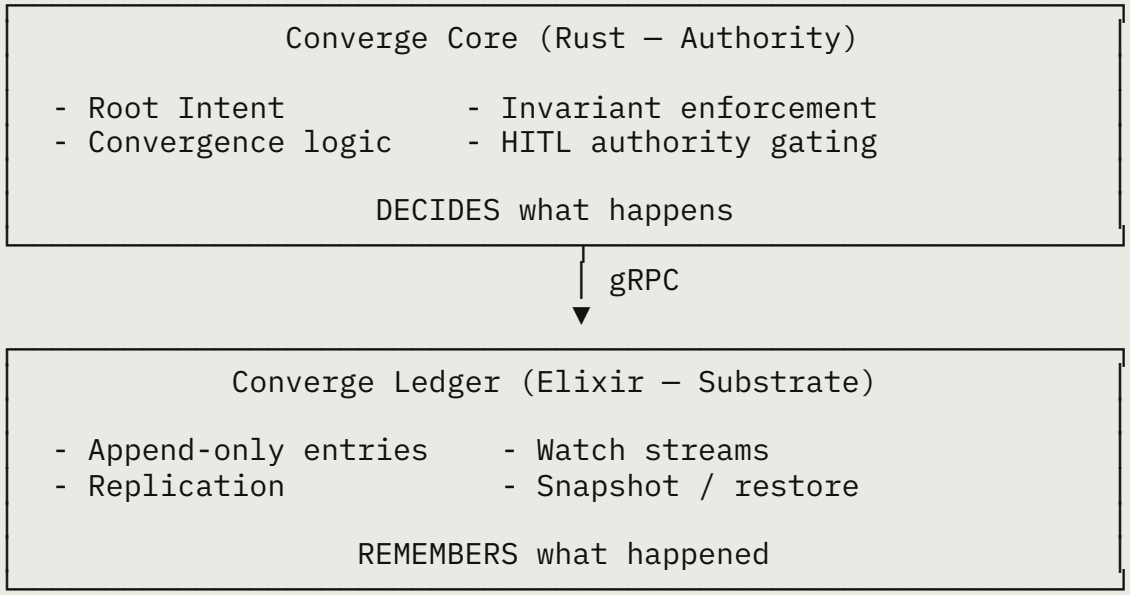
## "Just use Postgres"

### Why It Fails:

| Problem | Impact |
|---|---|
| Polling | Latency and load |
| Centralized | Single point of failure |
| No native clustering | Complex HA setup |
| No real-time streaming | More polling |
| Relational model | Fighting append-only semantics |

# The Architecture Principle

## "Functional Core, Imperative Shell"



# The Key Invariant

> **"The Ledger never decides. It only remembers."**

- The engine can run **without** the ledger
- The ledger is **meaningless** without the engine
- Losing ledger data must **not** break correctness
- The ledger **never** rewrites history

**All semantic authority lives in Converge Core (Rust).**

# The Single-Writer Model

## Why No Coordination?

For any given Root Intent:

| Property | Guarantee |
| --- | --- |
| Single Writer | Exactly one Converge engine appends |
| Multiple Readers | Any number of observers |
| Append-Only | No updates, deletes, or overwrites |

### What This Eliminates:

- No conflicts
- No merge logic
- No consensus protocols (Raft, Paxos)
- No distributed transactions

# Why Elixir/OTP?

## The Right Tool for the Job

| OTP Feature | How We Use It |
|---|---|
| Mnesia | Distributed, in-memory, soft-real-time storage |
| GenServer | Isolated failure domains |
| Supervisors | Automatic restart, self-healing |
| :net_kernel | Native clustering (no sidecars) |
| :pg | Zero-config service discovery |

### What We Explicitly Avoid in Elixir:

- Semantic validation → Rust
- Domain logic → Rust
- Invariant enforcement → Rust

# The API

## Five Operations. That's It.

| Operation | Purpose |
|---|---|
| `Append(context_id, entry)` | Add an immutable entry |
| `Get(context_id)` | Retrieve entries |
| `Snapshot(context_id)` | Serialize full context |
| `Load(context_id, blob)` | Restore from snapshot |
| `Watch(context_id)` | Stream new entries |

**Deliberately Missing:** Updates, Deletes, Transactions, Branching, Conditional writes, Queries

# Trust but Verify

## Merkle Trees for Cryptographic Integrity

```elixir
defmodule ConvergeLedger.Integrity.MerkleTree do
  def hash(content), do: :crypto.hash(:sha256, content)

  def combine(left, right), do: hash(left <> right)

  def compute_root([single]), do: combine(single, single)
  def compute_root(hashes), do: build_tree_level(hashes)
end
```

**Use Cases:**

- Snapshot verification (tamper detection)
- O(log n) sync/diff between replicas
- Audit proofs without revealing all entries

# Merkle Tree Performance

| Operation | Complexity | 10K entries |
|---|---|---|
| `compute_root` | O(n) | ~500ms |
| `generate_proof` | O(n) | ~100ms |
| `verify_proof` | O(log n) | ~0.01ms |

**Properties:**

- Deterministic: Same inputs → same root
- Collision-resistant: Different inputs → different root
- Bitcoin-style: Single element duplicated

# Causal Ordering

## Lamport Clocks — "Who Did What When"

**The Problem:** Wall clocks lie. Machine clocks drift.

**The Solution:** Logical time.

```elixir
defmodule ConvergeLedger.Integrity.LamportClock do
  defstruct time: 0

  def tick(%{time: t} = clock) do
    new_time = t + 1
    {%{clock | time: new_time}, new_time}
  end

  def update(%{time: local} = clock, received) do
    new_time = max(local, received) + 1
    {%{clock | time: new_time}, new_time}
  end
end
```

# Lamport Clock Guarantee

> If event A happened-before event B, then `clock(A) < clock(B)`

**Cross-Context Causality:**

```elixir
# Context A creates entry
{:ok, a} = Store.append("ctx-a", "facts", "data")
# a.lamport_clock = 1

# Context B receives A's data and extends
{:ok, b} = Store.append_with_received_time(
  "ctx-b", "facts", "derived", 1
)
# b.lamport_clock = 2 (guaranteed > a)
```

# Zero-Config Discovery

## How Nodes Find Each Other

```elixir
defmodule ConvergeLedger.Discovery do
  def join(context_id) do
    :pg.join({:context, context_id}, self())
  end

  def members(context_id) do
    :pg.get_members({:context, context_id})
  end

  def broadcast(context_id, message) do
    members(context_id)
    |> Enum.each(&send(&1, message))
  end
end
```

**No sidecars. No service mesh. No configuration.**

# Cluster Self-Healing

## Automatic Topology Management

```elixir
defmodule ConvergeLedger.Cluster.MnesiaManager do
  use GenServer

  def init(_) do
    :net_kernel.monitor_nodes(true)
    {:ok, %{}}
  end

  def handle_info({:nodeup, node}, state) do
    :mnesia.change_config(:extra_db_nodes, [node])
    replicate_tables()
    {:noreply, state}
  end
end
```

**On join:** Connect Mnesia, replicate tables
**On leave:** Automatic failover

# The Data Model

## Context Entries

| Context: growth-strategy-001 | | |
|---|---|---|
| seq=1 | facts | market_size: 2.4B |
| seq=2 | intents | objective: increase_demand |
| seq=3 | traces | agent:analyst started |
| seq=4 | proposals | strategy: partnership_model |
| seq=5 | evals | confidence: 0.73 |

**No updates. No deletes. No rewrites.**

# Supervision Tree

## Fault Isolation

```
ConvergeLedger.Supervisor
├── StorageSupervisor
│   └── Mnesia initialization
├── WatchRegistry (GenServer)
│   └── Subscriber management
├── MnesiaManager (GenServer)
│   └── Cluster topology
└── GrpcServerSupervisor
    └── External API
```

Each component has an **isolated failure domain**.

# Q&A: Why Not Kafka?

| Concern | Answer |
| --- | --- |
| Weight | Too heavy for our use case |
| Snapshotting | We need state recovery, not just events |
| Integrity | No built-in Merkle verification |
| Use case | Kafka is pub/sub; we need context memory |

# Q&A: Why Not Redis?

| Concern | Answer |
| --- | --- |
| Clustering | Complex native clustering |
| Integrity | No Merkle trees, no Lamport clocks |
| Persistence | Memory-only or complex setup |
| Model | Key-value, not append-only contexts |

# Q&A: Why Not Postgres?

| Concern | Answer |
| --- | --- |
| Streaming | Polling, not real-time |
| Clustering | No native clustering |
| Model | Relational fights append-only |
| Supervision | No process fault isolation |

# Architecture Danger Signs

## When to Stop and Re-Read the Docs

| If you want... | You need something else |
|---|---|
| Conditional writes | Coordination layer |
| Conflict resolution | Consensus protocol |
| Complex queries | A real database |
| Background jobs | Workflow engine |
| Event handlers | Pub/sub system |

**This is specialized memory, not a general-purpose system.**

# Summary

## Key Takeaways

1. **Separation of concerns:** Rust decides, Elixir remembers
2. **Single-writer model:** No coordination needed
3. **Append-only:** No conflicts, no merges
4. **Cryptographic integrity:** Merkle trees + Lamport clocks
5. **OTP primitives:** Battle-tested infrastructure
6. **Minimal API:** Five operations, no more

# The Golden Rule

> **Converge converges in Rust.**
> **Anything distributed exists only to remember what already happened.**

# Resources

- **README.md** — Overview and philosophy
- **ARCHITECTURE.md** — Detailed design decisions
- **AGENTS.md** — Process architecture
- **docs/INTEGRITY.md** — Merkle trees and Lamport clocks

# Thank You

## Questions?

*Converge Ledger — Optional, Derivative, Replaceable*

**github.com/converge-zone/converge-ledger**