What Makes Echo Tick?

Your Tour Guide: Claude (Opus 4.5)

Welcome! I've been asked to give you a personal tour through Echo's internals. This isn't just documentation—I'll share what I find elegant, surprising, and occasionally baffling about this codebase. When you see a red-outlined box, that's me stepping out of "narrator mode" to give you my unfiltered take.

Reading Time: ~45 minutes for complete understanding.

Table of Contents

---

1. Philosophy: Why Echo Exists

1.1 The Problem

Traditional game engines and simulations treat state as **mutable objects**. This creates fundamental problems:

- **Replay is hard**: You can't just "rewind" because state changes are scattered and untracked.
- **Synchronization is fragile**: Two machines running the same logic may diverge due to floating-point differences, thread timing, or iteration order.
- **Debugging is a nightmare**: "It worked on my machine" is the symptom of non-determinism.
- **Branching is impossible**: You can't easily ask "what if?" without copying everything.

▶ Claude's Commentary

Claude's Take: This list of problems isn't theoretical. I've seen countless debugging sessions where the root cause was "HashMap iteration order changed between runs." Echo's designers clearly got burned by non-determinism at some point and decided: *never again*.
What strikes me most is the last point—"branching is impossible." Most engines don't even *try* to support branching because it seems like a feature for version control, not runtime systems. Echo treats it as a first-class concern. That's unusual and, I think, genuinely forward-thinking.

1.2 Echo's Answer

Echo treats **state as a typed graph** and **all changes as rewrites**. Each "tick" of the engine:

1. Proposes a set of rewrites
2. Executes them in **deterministic order**
3. Emits **cryptographic hashes** of the resulting state

This means: - **Same inputs → Same outputs** (always, on any machine) - **State is verifiable** (hashes prove correctness) - **Replay is trivial** (patches are prescriptive) - **Branching is free** (copy-on-write snapshots)

1.3 Core Design Principles

```
                    ECHO'S THREE PILLARS




      DETERMINISM         PROVENANCE         TOOLING
        FIRST             YOU CAN           AS FIRST
                            TRUST             CLASS

    Same inputs         Snapshots are      Graphs stream
    always produce      content-           over canonical
    same hashes         addressed          wire protocol
```

▶ Claude's Commentary

Claude's Take: "Tooling as first-class" is the sleeper here. Most engines treat debugging tools, replay systems, and visualization as afterthoughts—bolted on after the core is done. Echo inverts this: the wire protocol, the hash scheme, and the diff format were designed *so that tools could exist*.
I've read a lot of engine architectures. This level of intentionality about tooling is rare. It's also why Echo can have a separate 'warp-viewer' crate that just... works, instead of requiring heroic reverse-engineering.

2. The Big Picture: Architecture Overview

2.1 System Layers

Echo is organized into distinct layers, each with a specific responsibility:

▶ Claude’s Commentary

Claude’s Take: This is a *clean* layer cake. Each layer only talks to its neighbors. No ”Layer 5 reaching down to Layer 1 for performance reasons.” That discipline is hard to maintain, and I respect it.
The ‘WSC Format‘ at Layer 2 caught my eye. It’s Echo’s custom columnar storage format—and before you ask ”why not just use Arrow or Parquet?”—I’ll spoil it: WSC is designed for mmap-friendly, zero-copy reads where every row is 8-byte aligned and you can binary-search directly into the file. It’s specialized for *exactly this use case*. Sometimes NIH syndrome is justified.

2.2 Crate Map

Crate	Purpose
`warp-core`	The deterministic rewrite engine (the “brain”)
`echo-graph`	Renderable graph types + diff operations
`echo-session-proto`	Wire protocol (canonical CBOR framing)
`echo-session-service`	Headless Unix-socket hub for tools
`echo-session-client`	Client helpers for connecting to the hub
`warp-viewer`	Native WGPU viewer for visualizing graphs

2.3 Data Flow Overview

▶ Claude’s Commentary

Claude’s Take: Notice how the Engine talks to itself multiple times before touching the Store? That’s the commit protocol at work. The Engine is *paranoid* about mutations—it queues up intentions, validates them, and only then touches state. If you’re used to ”just mutate it directly” game engines, this will feel ceremonial. The ceremony is the point.

3. Core Concepts: The WARP Graph

3.1 What is a WARP Graph?

A WARP (**W**orldline **A**lgebra for **R**ecursive **P**rovenance) graph is Echo’s fundamental data structure. It’s not just a graph—it’s a graph with **deterministic semantics**.

Figure 1: Diagram 1

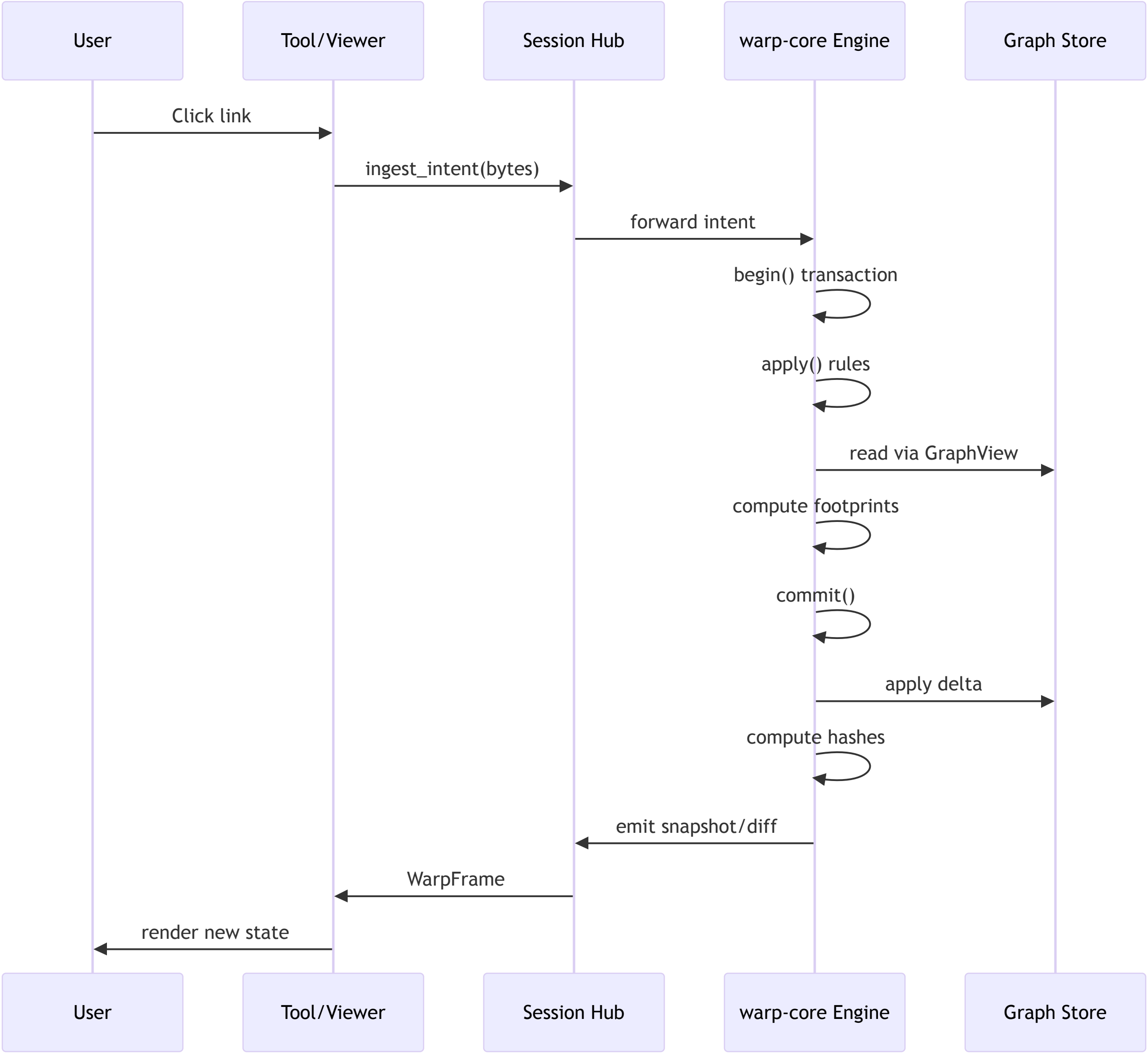

Figure 2: Diagram 2

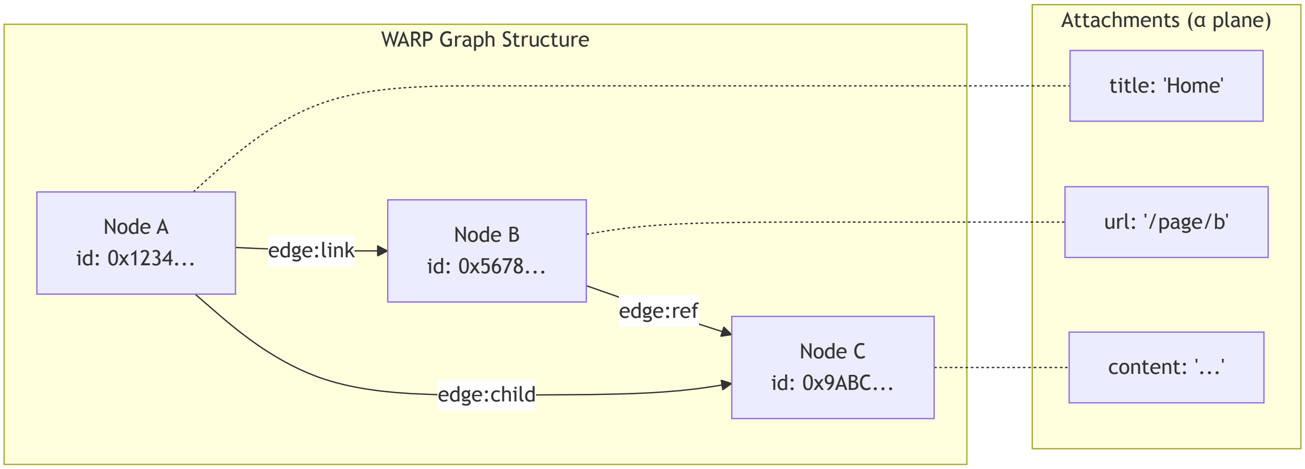

Figure 3: Diagram 3

▶ **Claude’s Commentary**

Claude’s Take: The name ”WARP” is doing a lot of work here. ”Worldline” evokes physics—specifically, the path an object traces through spacetime. In Echo, a node’s ”worldline” is its history of states across ticks. ”Recursive Provenance” means you can always ask ”where did this value come from?” and trace it back through the graph’s history.
Is the name a bit grandiose for what amounts to ”typed graph with audit trail”? Maybe. But I’ve seen worse acronyms in this industry.

3.2 Two-Plane Architecture

Echo separates structure from data via the **Two-Plane Model** (ADR-0001):

Plane	Contains	Purpose
Skeleton	Nodes + Edges (structure)	Fast traversal, deterministic hashing
Attachment ( )	Typed payloads	Domain-specific data

Why separate them?

```
 SKELETON PLANE (Structure)

           edge:link
     N1              N2


        edge:child                edge:ref


     N3



 ATTACHMENT PLANE (Payloads)

    N1. ["title"] = Atom { type: "string", bytes: "Home" }
    N2. ["url"]   = Atom { type: "string", bytes: "/page/b" }
    N3. ["body"]  = Atom { type: "html",   bytes: "<p>...</p>" }
```

Key insight: Skeleton rewrites **never decode attachments**. This keeps the hot path fast and deterministic.

▶ Claude's Commentary

Claude's Take: This is where Echo gets clever. The Skeleton plane only contains node IDs, edge IDs, and type tags—all fixed-size, all byte-comparable. You can compute the entire state hash without ever deserializing a single JSON blob, HTML string, or texture.

The Attachment plane (they call it ” ” because of course they do) holds the actual domain data. It participates in hashing but doesn't affect traversal. This separation means you can have a 10MB texture attached to a node and still iterate the graph at full speed.

I've seen similar ideas in ECS architectures, but usually the separation is ”components vs. systems.” Echo's split is ”structure vs. data,” which is subtly different and, I think, more principled.

3.3 Node and Edge Identity

Every node and edge has a **32-byte identifier**:

```
pub struct NodeId([u8; 32]);  // Content-addressed or assigned
pub struct EdgeId([u8; 32]);  // Unique edge identifier
```

These IDs are: - **Deterministic**: Same content → same ID (when content-addressed) - **Sortable**: Lexicographic ordering enables deterministic iteration - **Hashable**: Participate in state root computation

3.4 WarpInstances: Graphs Within Graphs

Echo supports **descended attachments**—embedding entire graphs within attachment slots:

This enables "WARPs all the way down"—recursive composition while maintaining determinism.

▶ Claude's Commentary

Claude's Take: WarpInstances are *wild*. You can have a node whose attachment slot contains... another entire graph. And that graph can have nodes whose attachment slots contain... more graphs. It's turtles, but the turtles are graphs.

Why would you want this? Think of a game with procedurally generated dungeons. Each dungeon could be its own WarpInstance, loaded on demand, with its own tick history and state root. The player character is in the ”outer” instance; stepping through a portal descends into the ”inner” one.

I don't know if Echo actually uses this feature yet, but the architecture supports it cleanly. That's design for the future without overengineering the present.

---

4. The Engine: Heart of Echo

4.1 The Engine Struct

The `Engine` is Echo's central orchestrator. Located in `crates/warp-core/src/engine_impl.rs`:

```
pub struct Engine {
    state: WarpState,                           // Multi-instance graph state
    rules: HashMap<RuleId, RewriteRule>,        // Registered rewrite rules
    scheduler: DeterministicScheduler,          // Deterministic ordering
    bus: MaterializationBus,                    // Output channels
    history: Vec<(Snapshot, TickReceipt, WarpTickPatchV1)>,
```

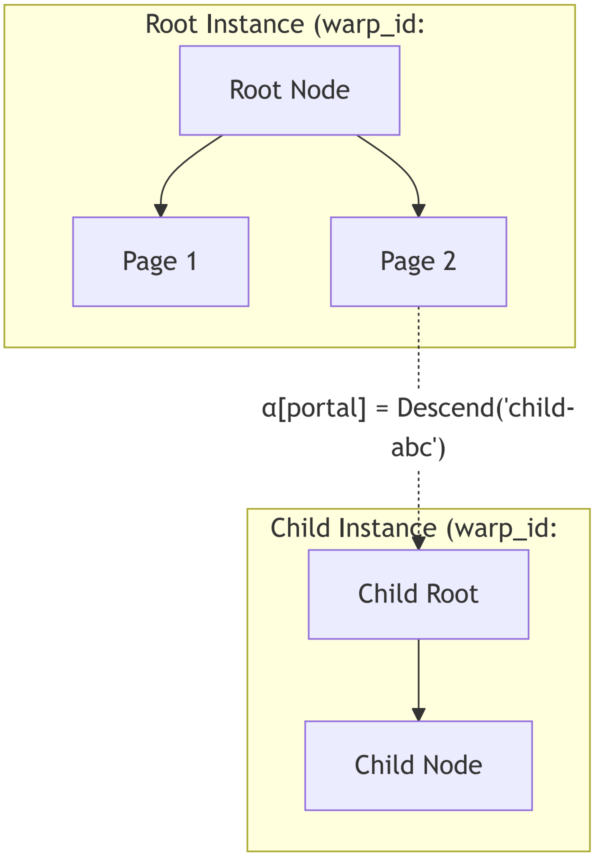

Figure 4: Diagram 4

```
    tx_counter: u64,                            // Transaction counter
    live_txs: BTreeSet<TxId>,                   // Active transactions
    // ... more fields
}
```

▶ Claude’s Commentary

Claude’s Take: A few things jump out here:
1. **‘rules: HashMap<RuleId, RewriteRule>‘** — Wait, HashMap? Isn’t that non-deterministic? It is! But notice: this is for *looking up* rules by ID, not for *iterating*. The iteration order is determined by the ‘scheduler‘, which is explicitly deterministic. The HashMap is fine because rule IDs are stable.
2. **‘history: Vec<(Snapshot, TickReceipt, WarpTickPatchV1)>‘** — The engine keeps its entire history in memory? That seems expensive. I suspect this is configurable, or there’s a garbage collection pass I haven’t found yet. For long-running simulations, unbounded history would be a problem.
3. **‘BTreeSet<TxId>‘ for live transactions** — BTreeSet, not HashSet. They’re *really* committed to determinism. Even the set of ”which transactions are in-flight” is stored in sorted order.

4.2 Construction

The engine is built via the `EngineBuilder`:

```
let engine = EngineBuilder::new(store, root_node_id)
    .with_policy_id(1)
    .with_telemetry(telemetry)
    .build();
```

What happens during construction:

4.3 Rewrite Rules

Rules are the atoms of change in Echo. Each rule has three functions:

```
pub struct RewriteRule {
    pub name: String,
    pub matcher: MatchFn,      // Does this rule apply?
    pub executor: ExecuteFn,    // What changes to make
    pub footprint: FootprintFn, // What resources are touched
    pub policy: ConflictPolicy, // What to do on conflict
}

// Function signatures (Phase 5 BOAW model):
type MatchFn     = fn(GraphView, &NodeId) -> bool;
type ExecuteFn   = fn(GraphView, &NodeId, &mut TickDelta);
type FootprintFn = fn(GraphView, &NodeId) -> Footprint;
```

Critical constraint: Executors receive a **read-only** `GraphView` and emit changes to a `TickDelta`. They **never** mutate the graph directly.

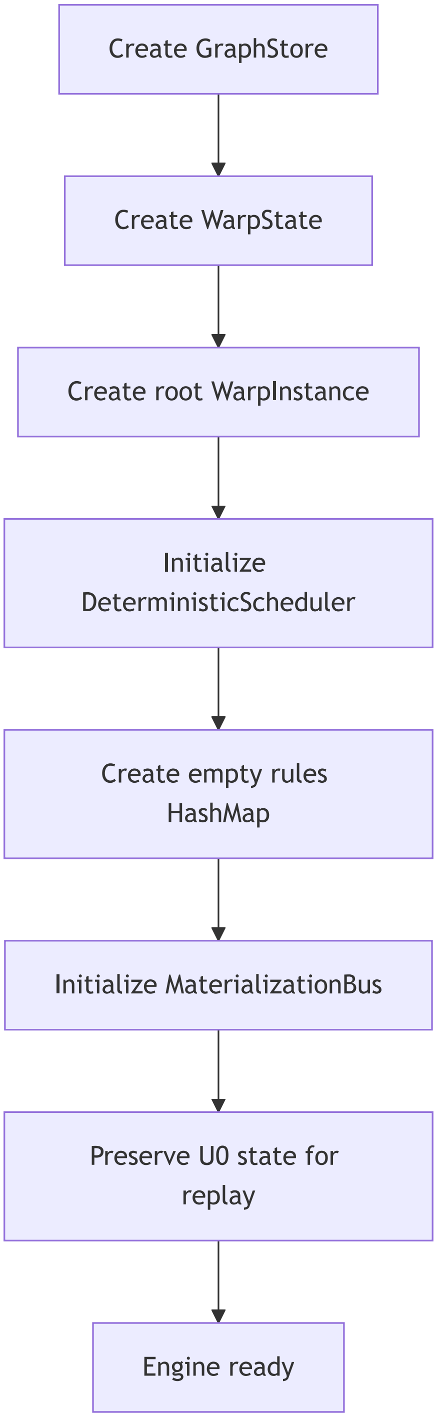
Create GraphStore
Create WarpState
Create root WarpInstance
Initialize DeterministicScheduler
Create empty rules HashMap
Initialize MaterializationBus
Preserve U0 state for replay
Engine ready

Figure 5: Diagram 5

▶ Claude’s Commentary

Claude’s Take: The ‘FootprintFn‘ is the secret sauce. Before executing a rule, Echo calls this function to ask: ”What nodes, edges, and attachments will you touch?” The footprint is a *conservative estimate*—you must declare everything you *might* read or write.

This enables Echo’s parallel execution model. If two rules have non-overlapping footprints, they can execute in parallel, in any order, and the result is guaranteed identical. If footprints overlap, they’re sequenced deterministically.

The burden on the rule author is significant: you must declare your footprint accurately, or you’ll get either conflicts (declared overlap when there was none) or silent bugs (undeclared overlap that corrupts state). This is a sharp edge in the API.

4.4 GraphView: Read-Only Access

The `GraphView` enforces BOAW’s immutability contract:

```rust
pub struct GraphView<'a> {
    store: &'a GraphStore,
    warp_id: WarpId,
}

impl<'a> GraphView<'a> {
    pub fn node(&self, id: &NodeId) -> Option<&NodeRecord>;
    pub fn edges_from(&self, id: &NodeId) -> impl Iterator<Item = &EdgeRecord>;
    pub fn node_attachment(&self, id: &NodeId, key: &str) -> Option<&AttachmentValue>;
    // ... read-only methods only
}
```

No `DerefMut`, no `AsRef<GraphStore>`, no interior mutability. This is enforced at the type level.

▶ Claude’s Commentary

Claude’s Take: I went looking for escape hatches here. ‘RefCell‘? No. ‘UnsafeCell‘? No. ‘Arc<Mutex<...»‘? No. The ‘GraphView‘ is genuinely immutable by construction.

This is Rust at its best: the borrow checker prevents you from shooting yourself in the foot. In C++, you’d need discipline and code review to enforce ”executors don’t mutate the graph.” In Rust, it’s just... not possible. The types don’t allow it.

5. The Tick Pipeline: Where Everything Happens

5.1 Overview

A “tick” is one complete cycle of the engine. It has five phases:

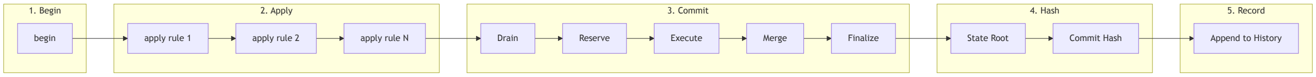

Figure 6: Diagram 6

▶ Claude’s Commentary

Claude’s Take: The ”Commit” phase has five sub-steps. *Five*. This is where I started to appreciate how much thought went into this system. Let me summarize what each does:
1. **Drain**: Pull all pending rewrites from the scheduler in canonical order 2. **Reserve**: Check footprints for conflicts, accept or reject each rewrite 3. **Execute**: Run the accepted rewrites (this is where parallelism happens) 4. **Merge**: Combine all ‘TickDelta‘ outputs into a single canonical operation list 5. **Finalize**: Apply the merged operations to produce the new state

The reservation phase is particularly clever. It’s like a two-phase commit: first you ”reserve” your footprint (claim your lock), then you execute. If your footprint conflicts with an already-reserved footprint, you’re rejected. No execution happens until all accepted rewrites have been validated.

5.2 Phase 1: Begin Transaction

```
let tx = engine.begin();
```

What happens: 1. Increment `tx_counter` (wrapping to avoid 0) 2. Add `TxId` to `live_txs` set 3. Return opaque transaction identifier

```
engine.begin()

tx_counter: 0 → 1
live_txs: {} → {TxId(1)}
returns: TxId(1)
```

5.3 Phase 2: Apply Rules

```
engine.apply(tx, "rule_name", &scope_node_id);
```

What happens:

The Footprint: A declaration of what resources the rule will read and write:

```
pub struct Footprint {
    pub n_read: BTreeSet<NodeId>,    // Nodes to read
    pub n_write: BTreeSet<NodeId>,   // Nodes to write
    pub e_read: BTreeSet<EdgeId>,    // Edges to read
    pub e_write: BTreeSet<EdgeId>,   // Edges to write
    pub a_read: BTreeSet<AttachmentKey>,   // Attachments to read
    pub a_write: BTreeSet<AttachmentKey>, // Attachments to write
    // ... ports, factor_mask
}
```

Scheduler deduplication: If the same `(scope_hash, rule_id)` is applied multiple times, **last wins**. This enables idempotent retry semantics.

5.4 Phase 3: Commit (The Heart of Determinism)

```
let (snapshot, receipt, patch) = engine.commit_with_receipt(tx);
```

This is where Echo’s magic happens. Let’s break it down:

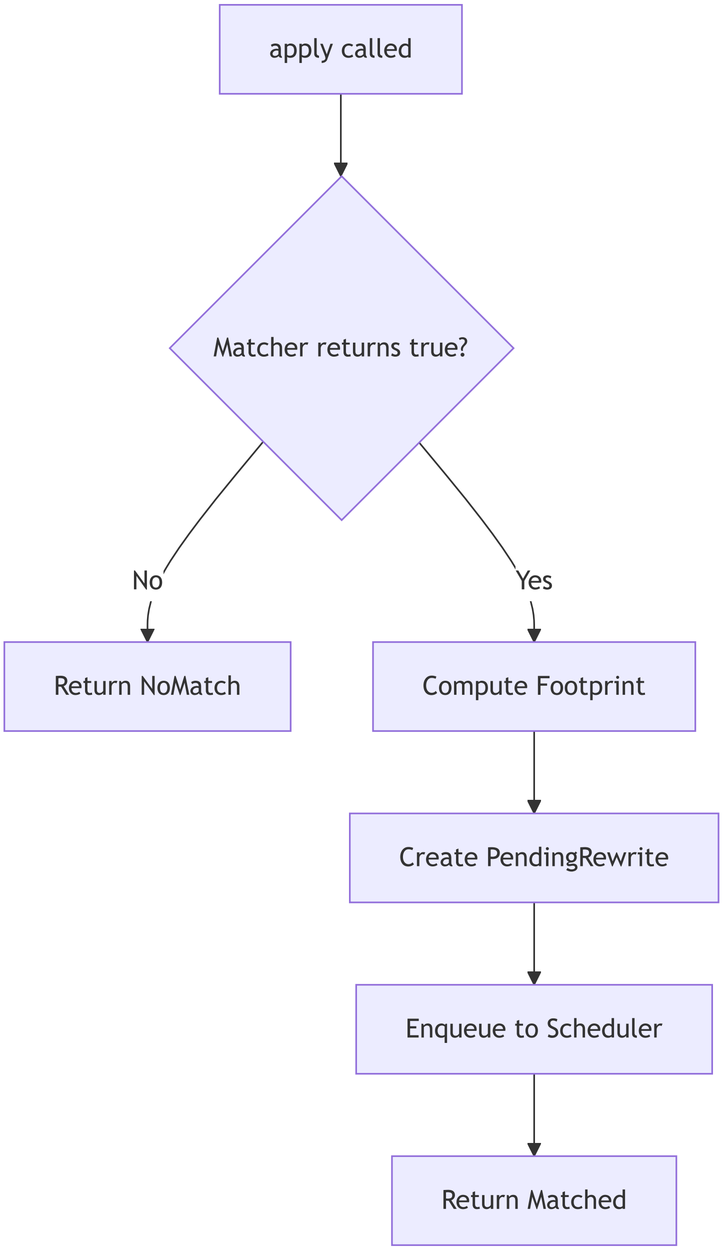

Figure 7: Diagram 7

5.4.1 Drain

The scheduler drains all pending rewrites in **canonical order**:

```
// RadixScheduler uses O(n) LSD radix sort
// 20 passes: 2 nonce + 2 rule_id + 16 scope_hash (16-bit digits)
let rewrites = scheduler.drain_for_tx(tx);  // Vec<PendingRewrite> in canonical order
```

Ordering key: `(scope_hash[0..32], rule_id, nonce)`

This ensures the **same rewrites always execute in the same order**, regardless of when they were applied.

▶ Claude’s Commentary

Claude’s Take: Radix sort! They’re using radix sort for the scheduler drain. Not quicksort, not merge sort—radix sort.
Why? Because radix sort is *stable* and *deterministic* by construction. Quicksort’s behavior depends on pivot selection, which can vary. Merge sort is deterministic, but radix sort is faster for fixed-size keys. Since the ordering key is exactly 36 bytes (32-byte scope hash + 2-byte rule ID + 2-byte nonce), radix sort is perfect.
This is the kind of detail that separates ”deterministic by accident” from ”deterministic by design.”

5.4.2 Reserve (Independence Check)

For each rewrite in canonical order:

Conflict detection: Uses `GenSet<K>` for O(1) lookups: - Read-read overlap: **allowed** - Write-write overlap: **conflict** - Read-write overlap: **conflict**

5.4.3 Execute (Parallel, Lockless)

Accepted rewrites execute against the **read-only snapshot**:

```
for rewrite in accepted {
    let rule = &rules[rewrite.rule_id];
    let view = GraphView::new(&state, rewrite.warp_id);

    // Executor reads from view, emits to delta
    (rule.executor)(view, &rewrite.scope, &mut delta);
}
```

Critical: `GraphView` is immutable. `TickDelta` accumulates operations:

```
pub struct TickDelta {
    ops: Vec<(WarpOp, OpOrigin)>,
}

// Operations emitted during execution:
delta.emit(WarpOp::UpsertNode { id, record });
delta.emit(WarpOp::UpsertEdge { from, edge });
delta.emit(WarpOp::DeleteNode { id });
delta.emit(WarpOp::SetAttachment { node, key, value });
```

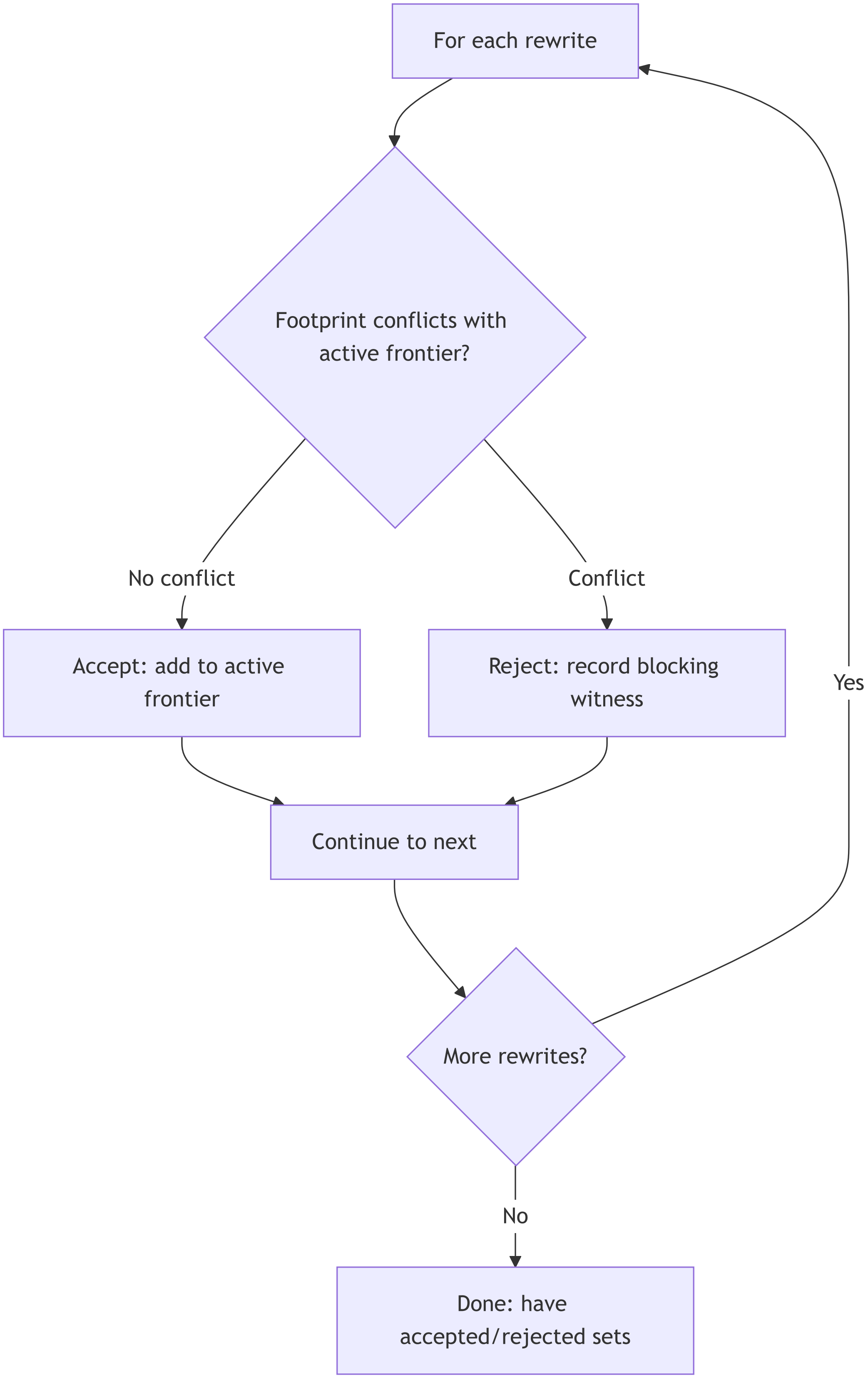
For each rewrite
Footprint conflicts with active frontier?
No conflict
Conflict
Accept: add to active frontier
Reject: record blocking witness
Yes
Continue to next
More rewrites?
No
Done: have accepted/rejected sets

Figure 8: Diagram 8

5.4.4 Merge (Canonical Sort)

All operations are sorted into **canonical replay order**:

```
// Sort by (WarpOpKey, OpOrigin)
ops.sort_by_key(|(op, origin)| (op.sort_key(), origin.clone()));

// Deduplicate identical ops
// Error on conflicting ops (footprint model violation)
```

Conflict handling: If two rewrites wrote **different values** to the same key, that's a bug in the footprint model. Echo errors loudly.

5.4.5 Finalize

Apply the merged delta to produce the new state:

```
for op in merged_ops {
    match op {
        WarpOp::UpsertNode { id, record } => state.insert_node(id, record),
        WarpOp::UpsertEdge { from, edge } => state.insert_edge(from, edge),
        WarpOp::DeleteNode { id } => state.delete_node_cascade(id),
        WarpOp::SetAttachment { node, key, value } => state.set_attachment(node, key, value),
        // ...
    }
}
```

5.5 Phase 4: Hash Computation

State Root (BLAKE3)

The state root is computed via **deterministic BFS** over reachable nodes:

Encoding (architecture-independent): - All IDs: raw 32 bytes - Counts: u64 little-endian - Payloads: 1-byte tag + type_id[32] + u64 LE length + bytes

Commit Hash (v2)

```
commit_hash = BLAKE3(
    version_tag[4]       ||  // Protocol version
    parents[]            ||  // Parent commit hashes
    state_root[32]       ||  // Graph-only hash
    patch_digest[32]     ||  // Merged ops digest
    policy_id[4]             // Policy identifier
)
```

▶ Claude's Commentary

Claude's Take: The commit hash includes a ‘policy$_i d`.Thisissubtlebutimportant: twoengineswithdifferentpoliciescouldproducethesamestatebutdifferentcommithashes.Why?Becausethe* process * matters, notjusttheresult.$

Imagine one policy allows rules to run in parallel; another requires sequential execution. They might produce identical graphs, but the commit hashes differ because the policies differ. This prevents

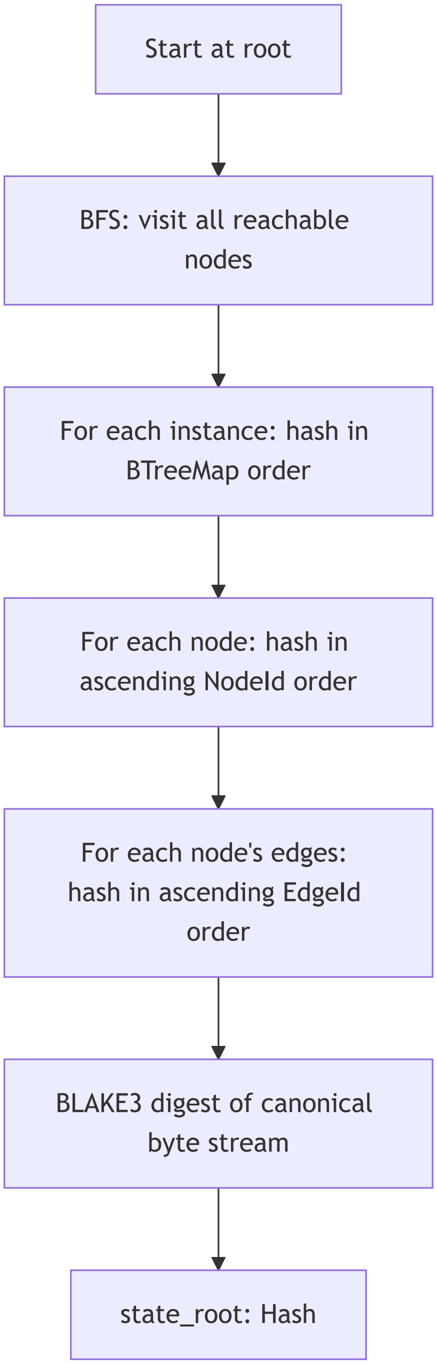
Start at root
BFS: visit all reachable nodes
For each instance: hash in BTreeMap order
For each node: hash in ascending NodeId order
For each node's edges: hash in ascending EdgeId order
BLAKE3 digest of canonical byte stream
state_root: Hash

Figure 9: Diagram 9

accidentally mixing outputs from incompatible engine configurations.
It’s defensive design: ”Trust, but verify—and make verification easy.”

5.6 Phase 5: Record to History

```
history.push((
    Snapshot { hash: commit_hash, state_root, parents, ... },
    TickReceipt { applied, rejected, ... },
    WarpTickPatchV1 { ops, in_slots, out_slots, patch_digest, ... }
));
```

The patch is **prescriptive**: it can be replayed without re-matching to reproduce the exact same state.

---

6. Parallel Execution: BOAW (Bag of Autonomous Workers)

6.1 What is BOAW?

BOAW stands for **Bag of Autonomous Workers**. It’s Echo’s parallel execution architecture that enables:

- **Massive parallelism** without locks
- **Deterministic convergence** across platforms
- **Worker-count invariance** (same result with 1 or 32 workers)

6.2 The Key Insight

```
                     THE BOAW INSIGHT


  Traditional parallelism:
    "Make execution order deterministic" → Complex, slow

  BOAW parallelism:
    "Let execution order vary, make MERGE deterministic" → Fast!

  Workers race freely → Each produces a TickDelta
  Merge step sorts all deltas → Canonical output
```

▶ Claude’s Commentary

Claude’s Take: This is the insight that makes Echo work. Most parallel systems try to *control* the execution order—barriers, locks, atomic sequences. BOAW says: ”Forget it. Let chaos reign during execution. We’ll sort it out in the merge.”
It’s like MapReduce: the map phase runs in any order; the reduce phase (merge) produces the canonical result. But unlike MapReduce, Echo operates on a graph with complex dependencies.

The footprint model makes this possible: by declaring what you’ll touch before executing, you enable the merge to validate that no conflicts occurred.
If this sounds too good to be true, it mostly is—*if* you get the footprints wrong. The system is only as deterministic as your footprint declarations. Lie to the footprint system, and you’ll get non-determinism.

6.3 Execution Strategies

Phase 6A: Stride Partitioning (Legacy)

```
Worker 0: items[0], items[4], items[8], ...
Worker 1: items[1], items[5], items[9], ...
Worker 2: items[2], items[6], items[10], ...
Worker 3: items[3], items[7], items[11], ...
```

Problem: Poor cache locality—related items scatter across workers.

Phase 6B: Virtual Shards (Current Default)

```
const NUM_SHARDS: usize = 256;  // Protocol constant (frozen)

fn shard_of(node_id: &NodeId) -> usize {
    let bytes = node_id.as_bytes();
    let val = u64::from_le_bytes(bytes[0..8]);
    (val & 255) as usize  // Fast modulo via bitmask
}
```

Benefits: - Items with same `shard_of(scope)` processed together → better cache hits - Workers dynamically claim shards via atomic counter → load balancing - Determinism enforced by merge, not execution order

▶ Claude’s Commentary

Claude’s Take: 256 shards is an interesting choice. It’s small enough that the atomic counter for work-stealing doesn’t become a bottleneck, but large enough to distribute work across many cores.
The ‘ 255‘ bitmask is a micro-optimization I appreciate. It’s equivalent to ‘
One thing I wondered: what if your NodeIds are clustered? Like, if all recent nodes have IDs starting with ‘0x00...‘, they’d all end up in shard 0. I suspect content-addressed IDs (via BLAKE3) distribute uniformly, so this isn’t a problem in practice. But for user-assigned IDs, you’d need to be careful.

6.4 The Execution Loop

```
pub fn execute_parallel_sharded(
    view: GraphView<'_>,
    items: &[ExecItem],
    workers: usize,
) -> Vec<TickDelta> {
    // Partition items into 256 shards
```

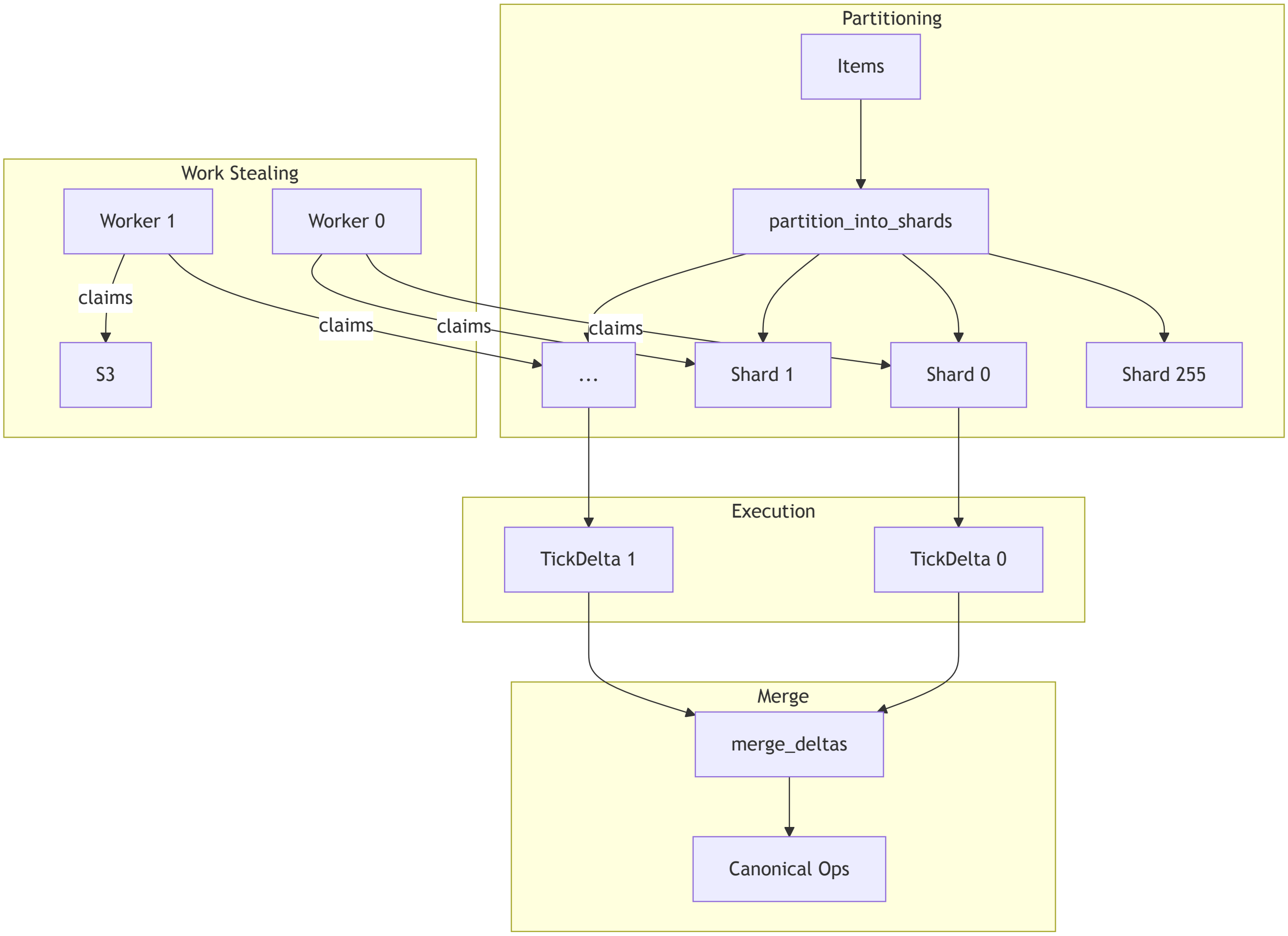

Figure 10: Diagram 10

```
    let shards = partition_into_shards(items);

    // Atomic counter for work-stealing
    let next_shard = AtomicUsize::new(0);

    std::thread::scope(|s| {
        let handles: Vec<_> = (0..workers).map(|_| {
            s.spawn(|| {
                let mut delta = TickDelta::new();
                loop {
                    // Claim next shard atomically
                    let shard_id = next_shard.fetch_add(1, Ordering::Relaxed);
                    if shard_id >= NUM_SHARDS { break; }

                    // Execute all items in this shard
                    for item in &shards[shard_id].items {
                        (item.exec)(view.clone(), &item.scope, &mut delta);
                    }
                }
                delta
            })
        }).collect();

        handles.into_iter().map(|h| h.join().unwrap()).collect()
    })
}
```

6.5 The Canonical Merge

```
pub fn merge_deltas(deltas: Vec<TickDelta>) -> Result<Vec<WarpOp>, MergeConflict> {
    // 1. Flatten all ops from all workers
    let mut all_ops: Vec<(WarpOpKey, OpOrigin, WarpOp)> = deltas
        .into_iter()
        .flat_map(|d| d.ops_with_origins())
        .collect();

    // 2. Sort canonically by (key, origin)
    all_ops.sort_by_key(|(key, origin, _)| (key.clone(), origin.clone()));

    // 3. Deduplicate and detect conflicts
    let mut result = Vec::new();
    for group in all_ops.group_by(|(k1, _, _), (k2, _, _)| k1 == k2) {
        let first = &group[0].2;
        if group.iter().all(|(_, _, op)| op == first) {
            result.push(first.clone());  // All identical: keep one
        } else {
            return Err(MergeConflict { writers: group.iter().map(|(_, o, _)| o).collect() });
        }
    }

    Ok(result)
}
```

Key guarantee: Conflicts are bugs. If footprints were correct, no two rewrites should write different values to the same key.

---

7. Storage & Hashing: Content-Addressed Truth

7.1 The GraphStore

Located in `crates/warp-core/src/graph.rs`:

```rust
pub struct GraphStore {
    pub(crate) warp_id: WarpId,
    pub(crate) nodes: BTreeMap<NodeId, NodeRecord>,
    pub(crate) edges_from: BTreeMap<NodeId, Vec<EdgeRecord>>,
    pub(crate) edges_to: BTreeMap<NodeId, Vec<EdgeId>>,  // Reverse index
    pub(crate) node_attachments: BTreeMap<NodeId, AttachmentValue>,
    pub(crate) edge_attachments: BTreeMap<EdgeId, AttachmentValue>,
    pub(crate) edge_index: BTreeMap<EdgeId, NodeId>,      // Edge → Source
    pub(crate) edge_to_index: BTreeMap<EdgeId, NodeId>,  // Edge → Target
}
```

Why BTreeMap everywhere? - Deterministic iteration order (sorted by key) - Enables canonical hashing - No HashMap ordering surprises

▶ Claude's Commentary

Claude's Take: Seven BTreeMaps! This is the price of determinism. Each of these maps is sorted, which means:
1. Insertions are O(log n) instead of O(1) amortized for HashMap 2. Iteration is always in key order, so hashing is deterministic 3. Memory overhead is slightly higher due to tree structure
Is it worth it? For Echo's use case, absolutely. The alternative—using HashMap and then sorting before each hash—would be slower and more error-prone. By paying the cost upfront (O(log n) writes), you get guaranteed correctness.
The multiple indices (‘edges$_{f}rom`,`edges_{t}o`,`edge_{i}ndex`,`edge_{t}o_{i}ndex`)lookredundant,buttheyenableO(logn)look$
$from*anode?`edges_{f}rom[node_{i}d]`.Wantalledges*to*anode?`edges_{t}o[node_{i}d]`.Thisisaclassicspace-$
$timetradeoff.$

7.2 WSC: Write-Streaming Columnar Format

For efficient snapshots, Echo uses WSC—a zero-copy, mmap-friendly format:

```
 WSC SNAPSHOT FILE


   NODES TABLE (sorted by NodeId)

     NodeRow    NodeRow     NodeRow    ...
     64 bytes   64 bytes    64 bytes
```

```
EDGES TABLE (sorted by EdgeId)

  EdgeRow     EdgeRow     EdgeRow     ...
  128 bytes   128 bytes   128 bytes



OUT_INDEX (per-node → range into out_edges)

  Range (16 B)     Range (16 B)     ...



BLOB ARENA (variable-length data)
Referenced by (offset, length) tuples
```

Row types (8-byte aligned): - `NodeRow`: 64 bytes (node_id[32] + node_type[32]) - `EdgeRow`: 128 bytes (edge_id[32] + from[32] + to[32] + type[32]) - `Range`: 16 bytes (start_le[8] + len_le[8])

▶ Claude's Commentary

Claude's Take: WSC is gloriously simple. Fixed-size rows, sorted tables, binary search for lookups. No compression, no Parquet-style encoding tricks—just flat bytes on disk that you can mmap and use directly.

The trade-off is size: WSC files are larger than compressed formats. But the benefit is speed: you can find node 1000 by seeking to ‘offset + 1000 * 64‘ and reading 64 bytes. No decompression, no index lookups, no memory allocation.

For Echo's use case (local caching, fast restarts), this makes sense. You're not storing petabytes; you're storing the state of a single simulation that fits in RAM. Optimize for access latency, not storage cost.

7.3 Copy-on-Write Semantics

Rule: During a tick, nothing shared is mutated.

Structural sharing: Only changed segments are newly written. Unchanged data is referenced by hash.

7.4 Hash Algorithm Details

State Root (BLAKE3, v2):

```
state_root = BLAKE3(
    root_id[32]                    ||
    instance_count[8, LE]          ||
    for each instance in BTreeMap order:
        warp_id_len[8, LE]         ||
```

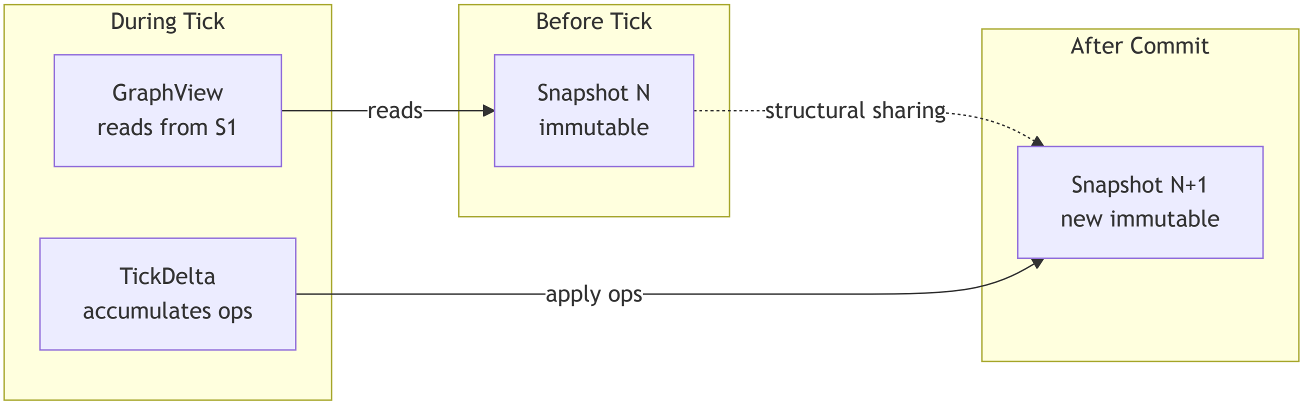

Figure 11: Diagram 11

```
        warp_id_bytes               ||
        node_count[8, LE]           ||
        for each node in ascending NodeId order:
            node_id[32]             ||
            node_type[32]           ||
            for each outbound edge in ascending EdgeId order:
                edge_id[32]         ||
                edge_type[32]       ||
                to_node[32]         ||
            for each attachment:
                key_len[8, LE]      ||
                key_bytes           ||
                type_id[32]         ||
                value_len[8, LE]    ||
                value_bytes
)
```

▶ **Claude’s Commentary**

Claude’s Take: The hashing is *exhaustive*. Every node, every edge, every attachment, every byte—all streamed through BLAKE3 in a defined order. There’s no ”we’ll just hash the IDs and trust the content”—everything participates.
This is expensive! But it’s the foundation of Echo’s trust model. If two engines produce the same state root, they have the same state. Period. No exceptions, no edge cases.
The ‘version$_{t}ag\text{`}inthecommithashisanicetouch.IfEchoeverchangesitshashingalgorithm(say, BLAKE3v2tov3),$

---

8. Worked Example: Tracing a Link Click

Let’s trace what happens when a user clicks a link in a hypothetical WARP-based navigation system.

8.1 The Scenario

Imagine a simple site with two pages:

User clicks the link: This should navigate from Home to About.

▶ **Claude’s Commentary**

Claude’s Take: This example is deceptively simple—two pages, one link—but it exercises the entire engine: intent ingestion, rule matching, footprint validation, execution, merge, hashing, and emission.
I’ll add my notes at the interesting points. If you’re skimming, watch for where the determinism guarantees kick in.

8.2 Step 1: Intent Ingestion

The click is captured by the viewer and converted to an **intent**:

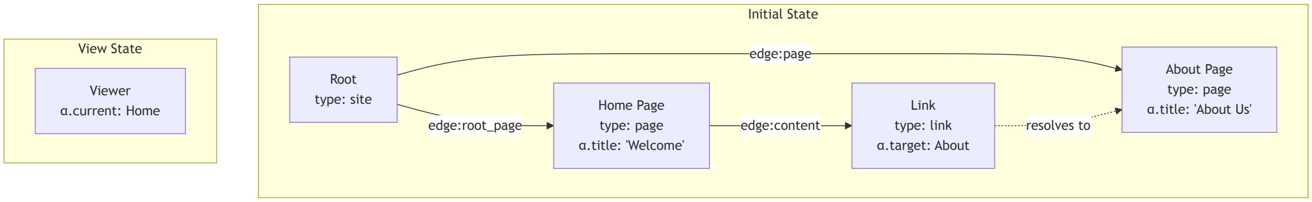

Figure 12: Diagram 12

```
// In the viewer:
let intent = NavigateIntent {
    target_page: about_node_id,
    timestamp: deterministic_tick,
};
let intent_bytes = canonical_encode(&intent);

// Send to engine:
engine.ingest_intent(intent_bytes);
```

What happens inside `ingest_intent`:

8.3 Step 2: Begin Transaction

```
let tx = engine.begin();  // tx = TxId(1)
```

8.4 Step 3: Dispatch Intent

```
engine.dispatch_next_intent(tx);
```

What happens:

8.5 Step 4: Rule Matching

The `cmd/navigate` rule matches:

```
// Matcher: Does this intent want navigation?
fn navigate_matcher(view: GraphView, scope: &NodeId) -> bool {
    let intent = view.node(scope)?;
    intent.type_id == "navigate_intent"
}

// Footprint: What will we read/write?
fn navigate_footprint(view: GraphView, scope: &NodeId) -> Footprint {
    Footprint {
        n_read: btreeset![scope.clone(), viewer_node],
        n_write: btreeset![],
        a_read: btreeset![],
        a_write: btreeset![AttachmentKey::new(viewer_node, "current")],
        ..default()
    }
}
```

▶ Claude’s Commentary

Claude’s Take: Notice the footprint. We declare that we’ll: - **Read** two nodes: the intent (to get the target) and the viewer (to validate the current page) - **Write** one attachment: the viewer’s ‘current‘ attachment

We’re *not* reading any attachments (we just need the node records), and we’re *not* writing any nodes (the viewer node already exists). This precision matters—if another rule also wants to write ‘viewer.current‘, there’s a conflict.

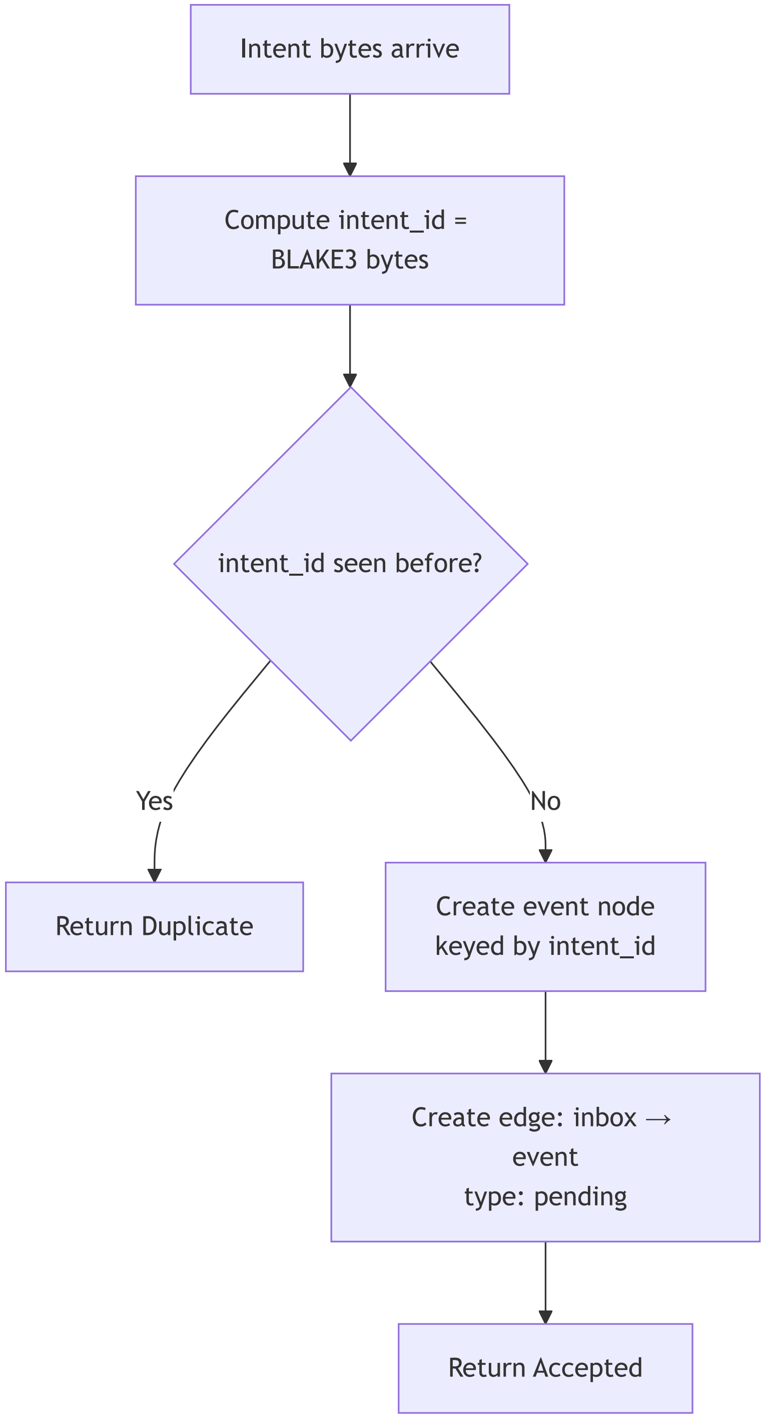
Intent bytes arrive
Compute intent_id = BLAKE3 bytes
intent_id seen before?
Yes
No
Return Duplicate
Create event node keyed by intent_id
Create edge: inbox → event type: pending
Return Accepted

Figure 13: Diagram 13

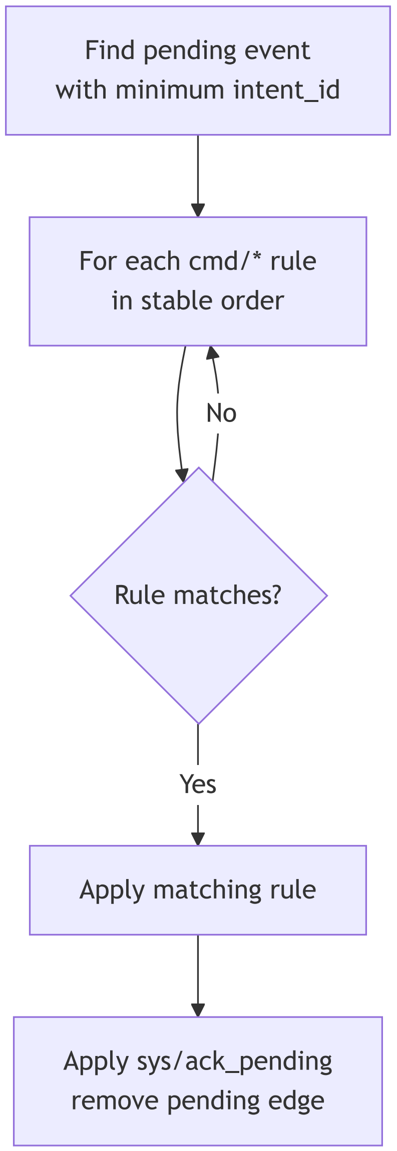

Figure 14: Diagram 14

The rule is enqueued:

```
PendingRewrite

rule_id: "cmd/navigate"
scope: 0xABCD... (intent node)
footprint: { n_read: [intent, viewer], a_write: [current] }
tx: TxId(1)
```

8.6 Step 5: Commit

```
let (snapshot, receipt, patch) = engine.commit_with_receipt(tx);
```

5a. Drain

```
let rewrites = scheduler.drain_for_tx(tx);
// Result: [PendingRewrite { rule: "cmd/navigate", scope: intent_node }]
```

5b. Reserve

```
// Check footprint independence
// No conflicts (only one rewrite)
// Accepted!
```

5c. Execute

```
fn navigate_executor(view: GraphView, scope: &NodeId, delta: &mut TickDelta) {
    // Read the intent to find target
    let intent = view.node(scope).unwrap();
    let target_page = intent.attachment("target").unwrap();

    // Read current viewer state (for logging/validation)
    let viewer = view.node(&VIEWER_NODE).unwrap();
    let old_page = viewer.attachment("current");

    // Emit the change: update viewer's current page
    delta.emit(WarpOp::SetAttachment {
        node: VIEWER_NODE,
        key: "current".into(),
        value: AttachmentValue::Atom(AtomPayload {
            type_id: "node_ref".into(),
            bytes: target_page.to_bytes(),
        }),
    });
}
```

TickDelta now contains:

```
[
    (WarpOp::SetAttachment {
        node: viewer_node,
        key: "current",
        value: about_node_id
    }, OpOrigin { intent_id: 1, rule_id: 42, match_ix: 0, op_ix: 0 })
]
```

5d. Merge

Only one delta, trivial merge:

```
let merged_ops = vec![
    WarpOp::SetAttachment { node: viewer_node, key: "current", value: about_node_id }
];
```

5e. Finalize

Apply to state:

```
state.set_attachment(viewer_node, "current", about_node_id);
```

8.7 Step 6: Hash Computation

```
// State root: BLAKE3 of reachable graph
let state_root = compute_state_root(&state);  // 0x7890...

// Patch digest: BLAKE3 of merged ops
let patch_digest = compute_patch_digest(&merged_ops);  // 0xDEF0...

// Commit hash
let commit_hash = BLAKE3(
    VERSION_TAG ||
    [parent_hash] ||
    state_root ||
    patch_digest ||
    policy_id
);  // 0x1234...
```

8.8 Step 7: Emit to Tools

The engine emits a `WarpDiff` to the session hub:

```
WarpDiff {
    from_epoch: 0,
    to_epoch: 1,
    ops: vec![
        WarpOp::SetAttachment {
            node: viewer_node,
            key: "current",
            value: about_node_id
        }
    ],
    state_hash: 0x7890...,
}
```

8.9 Step 8: Viewer Applies Diff

The viewer receives the diff and updates its rendering:

```
for op in diff.ops {
    match op {
        WarpOp::SetAttachment { node, key, value } => {
            if node == viewer_node && key == "current" {
                // Update the displayed page
                self.navigate_to(value.as_node_ref());
            }
        }
        _ => { /* other ops */ }
    }
}
```

Result: The user sees the About page.

▶ Claude's Commentary

Claude's Take: That's a lot of machinery for one link click! But here's what we get for free: 1. **Replay**: Save the intent bytes, replay them later, get the exact same state hash 2. **Verification**: Any other engine given the same inputs produces the same commit hash 3. **Undo**: The previous snapshot is still in history; restoring is a pointer swap 4. **Branching**: Fork the state, try a different navigation, compare outcomes

This is the payoff for all the ceremony. A traditional engine would do 'viewer.current = about$_p age`andcallitdone.Echobuildsa * provableaudittrail * aroundeverystatechange.$

---

9. The Viewer: Observing Echo

The `warp-viewer` crate provides real-time visualization of WARP graphs. It's built on WGPU for cross-platform GPU rendering.

9.1 Architecture

9.2 Rendering Pipeline

1. **Diff arrives** via session client
2. **State cache** updates local graph replica
3. **Layout engine** computes node positions (force-directed)
4. **Renderer** converts graph to GPU buffers
5. **Display** shows updated visualization

▶ Claude's Commentary

Claude's Take: The viewer is *reactive*, not poll-based. It subscribes to diffs from the session hub and updates only when state changes. This means zero CPU usage when the graph is idle.

The force-directed layout is a classic choice for graph visualization. It's not perfect—large graphs

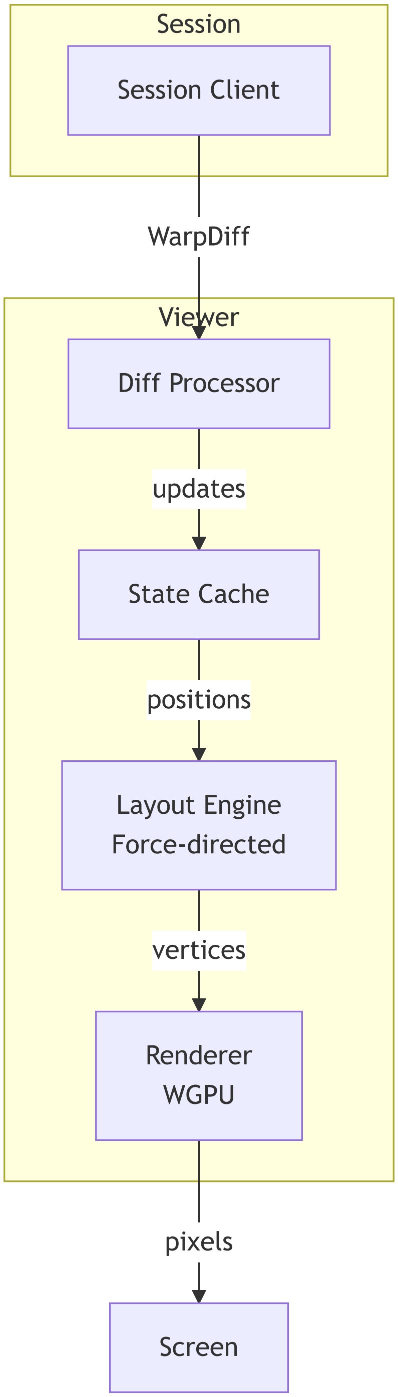

Figure 15: Diagram 15

can take time to settle—but it's good enough for debugging and exploration. If you need a specific layout, you can inject position attachments and the viewer will respect them.

10. Glossary

Term	Definition
WARP	Worldline Algebra for Recursive Provenance—Echo's core graph model
Tick	One complete cycle of the engine (begin → apply → commit → hash → record)
Snapshot	Immutable point-in-time capture of graph state
Footprint	Declaration of resources a rule will read/write
BOAW	Bag of Autonomous Workers—parallel execution model
TickDelta	Accumulated operations from rule execution
State Root	BLAKE3 hash of the entire graph
Commit Hash	BLAKE3 hash of (state root + patch + metadata)
WarpInstance	A graph-within-a-graph, enabling recursive composition
WSC	Write-Streaming Columnar—Echo's snapshot file format
GraphView	Read-only handle to graph state for rule executors
PendingRewrite	Queued rule application awaiting commit

▶ Claude's Commentary

Final Thoughts from Your Tour Guide

Echo is not a simple system. It's a *principled* system built on hard-won lessons about determinism, reproducibility, and trust.

What I find most impressive isn't any single feature—it's the coherence. Every piece reinforces the others: - BTreeMaps enable deterministic hashing - Footprints enable parallel execution - Parallel execution requires immutable GraphView - Immutable GraphView enables copy-on-write - Copy-on-write enables cheap branching - Cheap branching enables "what if?" queries

Pull one thread and the whole tapestry unravels. This is integrated design, not a collection of independent features.

Is Echo perfect? No. The footprint model requires discipline. The ceremony adds latency. The BTreeMaps trade speed for determinism. But for applications where *provability* matters—games with replays, simulations with audits, collaborative tools with conflict resolution—Echo offers something rare: a foundation you can trust.

Thanks for joining me on this tour. May your state roots always match.

— Claude