

# Echo: Tour de Code

## The Director's Cut

A complete function-by-function trace of Echo's execution pipeline, with commentary explaining what's *really* going on and why.

File paths and line numbers accurate as of 2026-01-18.

Hey! Welcome to the Director's Cut of the Echo Tour de Code.

I'm going to walk you through this codebase like we're pair programming. When I see something clever, I'll tell you why it's clever. When there's a non-obvious design decision, I'll explain the trade-offs. When there's a potential footgun, I'll point it out.

The goal here isn't just to show you *what* the code does—any decent `grep` can do that. I want you to understand *why* it does it this way, and what would break if you changed it.

Let's dive in.

---



# Contents

<b>Echo: Tour de Code</b>	<b>1</b>
1. Intent Ingestion . . . . .	4
1.1 Function Signature . . . . .	4
1.2 Complete Call Trace . . . . .	4
1.3 Data Structures Modified . . . . .	9
2. Transaction Lifecycle . . . . .	10
2.1 Begin Transaction . . . . .	10
2.2 Abort Transaction . . . . .	10
3. Rule Matching . . . . .	11
3.1 Function Signature . . . . .	11
3.2 Key Steps . . . . .	11
4. Scheduler: Drain & Reserve . . . . .	14
4.1 Drain Phase (Radix Sort) . . . . .	14
4.2 Reserve Phase (Independence Check) . . . . .	16
4.3 GenSet: $O(1)$ Conflict Detection . . . . .	16
5. BOAW Parallel Execution . . . . .	17
5.1 Entry Point . . . . .	18
5.2 Sharding . . . . .	18
5.3 Work Stealing Loop . . . . .	18
6. Delta Merge & State Finalization . . . . .	19
6.1 Canonical Merge . . . . .	20
6.2 Operation Ordering . . . . .	21
7. Hash Computation . . . . .	21
7.1 State Root . . . . .	22
7.2 Commit Hash v2 . . . . .	23
8. Commit Orchestration . . . . .	23
Appendix A: Complexity Summary . . . . .	25
Appendix B: Determinism Boundaries . . . . .	26

Guaranteed Deterministic . . . . .	26
Intentionally Non-Deterministic (Handled by Merge) . . . . .	26
Protocol Constants (Frozen) . . . . .	26

---

## 1. Intent Ingestion

**Entry Point:** `Engine::ingest_intent()`

**File:** `crates/warp-core/src/engine_impl.rs:1216`

This is where everything starts. A user does something—clicks a button, submits a form, whatever—and that action gets serialized into bytes and fed into this function.

The first thing to understand: Echo doesn't care *what* those bytes mean. It treats them as opaque data. The semantics come later, when rules interpret the bytes. Right now, we're just doing bookkeeping.

### 1.1 Function Signature

```
pub fn ingest_intent(&mut self, intent_bytes: &[u8]) -> Result<IngestDisposition, E>
```

**Returns:**

- `IngestDisposition::Accepted { intent_id: Hash }` — New intent accepted
- `IngestDisposition::Duplicate { intent_id: Hash }` — Already ingested

Notice the return type. We don't just return “success” or “failure”—we tell the caller *what happened*. Did we actually ingest this intent, or did we already have it?

This matters because in a distributed system, the same intent might arrive multiple times (network retries, replays, etc.). The caller needs to know whether this is a fresh intent or a duplicate so they can decide what to do next.

### 1.2 Complete Call Trace

`Engine::ingest_intent(intent_bytes: &[u8])`

```
[1] compute_intent_id(intent_bytes) → Hash
    FILE: crates/warp-core/src/inbox.rs:205
    CODE:
        let mut hasher = blake3::Hasher::new();
        hasher.update(b"intent:");           // Domain separation
        hasher.update(intent_bytes);
        hasher.finalize().into()              // → [u8; 32]
```

Okay, stop right here. This is the most important line in the entire function.

See that `b"intent:"` prefix? That's called **domain separation**, and it's a cryptographic best practice that a lot of codebases get wrong.

Here's the problem it solves: imagine you have some bytes that represent an intent. Now imagine those *exact same bytes* could also be interpreted as a node ID, or an edge ID, or some other identifier. Without domain separation, they'd all hash to the same value, and you'd have collisions between completely different concepts.

By prefixing with `"intent:"`, we guarantee that an intent hash can *never* collide with a node hash (which uses `"node:"`), or a type hash (`"type:"`), etc. Even if the raw bytes are identical, the hashes will be different.

Echo does this everywhere:

- `"intent:"` for intent IDs
- `"node:"` for node IDs
- `"type:"` for type IDs
- `"edge:"` for edge IDs

If you're ever tempted to add a new ID type, remember to pick a unique prefix. Future you will thank present you.

```
[2] NodeId(intent_id)
    Creates strongly-typed NodeId from Hash
```

**Pro Tip:** These newtype wrappers (`NodeId`, `EdgeId`, `TypeId`, etc.) are all just 32 bytes under the hood. But Rust's type system won't let you accidentally pass a `NodeId` where an `EdgeId` is expected. Zero runtime cost, maximum compile-time safety.

```
[3] self.state.store_mut(&warp_id) → Option<&mut GraphStore>
    FILE: crates/warp-core/src/engine_impl.rs:1221
    ERROR: EngineError::UnknownWarp if None
```

```
[4] Extract root_node_id from self.current_root.local_id

[5] STRUCTURAL NODE CREATION (Idempotent)
    make_node_id("sim") → NodeId
      FILE: crates/warp-core/src/ident.rs:93
      CODE: blake3("node:" || "sim")

    make_node_id("sim/inbox") → NodeId
      CODE: blake3("node:" || "sim/inbox")

    make_type_id("sim") → TypeId
      FILE: crates/warp-core/src/ident.rs:85
      CODE: blake3("type:" || "sim")

    make_type_id("sim/inbox") → TypeId
    make_type_id("sim/inbox/event") → TypeId

    store.insert_node(sim_id, NodeRecord { ty: sim_ty })
      FILE: crates/warp-core/src/graph.rs:175
      CODE: self.nodes.insert(id, record)

    store.insert_node(inbox_id, NodeRecord { ty: inbox_ty })
```

Step [5] is doing something subtle: it's creating the structural scaffolding for intents *idempotently*.

What does that mean? Well, imagine this is the first intent ever ingested. The “sim” node doesn't exist yet, nor does the “sim/inbox” node. So we create them.

But what if this is the millionth intent? Those structural nodes already exist. And here's the key insight: **because the IDs are derived from the names deterministically**, we get the same ID every time. `make_node_id("sim")` *always* returns the same hash.

So when we call `store.insert_node(sim_id, ...)`, if the node already exists with that ID, it's just a no-op (or an update—same difference for immutable nodes).

This is the beauty of content-addressed storage. You don't need “if exists” checks everywhere. Just compute the ID, do the insert, and let the storage layer handle deduplication.

## [6] STRUCTURAL EDGE CREATION

```

make_edge_id("edge:root/sim") → EdgeId
FILE: crates/warp-core/src/ident.rs:109
CODE: blake3("edge:" || "edge:root/sim")

store.insert_edge(root_id, EdgeRecord { ... })
FILE: crates/warp-core/src/graph.rs:188
GraphStore::upsert_edge_record(from, edge)
FILE: crates/warp-core/src/graph.rs:196
UPDATES:
    self.edge_index.insert(edge_id, from)
    self.edge_to_index.insert(edge_id, to)
    self.edges_from.entry(from).or_default().push(edge)
    self.edges_to.entry(to).or_default().push(edge_id)

store.insert_edge(sim_id, EdgeRecord { ... }) [sim → inbox]

```

Look at all those index updates in `upsert_edge_record`. We're maintaining *four separate indices* for edges:

1. `edge_index`: edge ID → source node
2. `edge_to_index`: edge ID → target node
3. `edges_from`: source node → list of edges
4. `edges_to`: target node → list of edge IDs

Why so many? Because graph queries can go in any direction:

- “What edges leave this node?” → `edges_from`
- “What edges arrive at this node?” → `edges_to`
- “Given this edge, what's its source?” → `edge_index`
- “Given this edge, what's its target?” → `edge_to_index`

Each of these is  $O(1)$  lookup. Yes, it's more memory. Yes, it's more bookkeeping on mutations. But graph traversal is *constant* in Echo, and that's worth a lot.

## [7] DUPLICATE DETECTION

```

store.node(&event_id) → Option<&NodeRecord>
FILE: crates/warp-core/src/graph.rs:87

```

```
CODE: self.nodes.get(id)
IF Some(_): return Ok(IngestDisposition::Duplicate { intent_id })
```

Here's where the content-addressing pays off beautifully.

Remember how we computed `intent_id` by hashing the intent bytes? And remember how we're about to use that same ID as the event node's ID?

That means: **if this exact intent was ever ingested before, it created a node with this exact ID**. So we can detect duplicates just by checking if the node exists.

No database sequence numbers. No UUIDs. No distributed coordination. Just: hash the bytes, check if that node exists. That's it.

This is why content-addressed systems are so elegant. Deduplication is *free*.

#### [8] EVENT NODE CREATION

```
store.insert_node(event_id, NodeRecord { ty: event_ty })
NOTE: event_id = intent_id (content-addressed)
```

#### [9] INTENT ATTACHMENT

```
AtomPayload::new(type_id, bytes)
FILE: crates/warp-core/src/attachment.rs:149
CODE: Self { type_id, bytes: Bytes::copy_from_slice(intent_bytes) }

store.set_node_attachment(event_id, Some(AttachmentValue::Atom(payload)))
FILE: crates/warp-core/src/graph.rs:125
CODE: self.node_attachments.insert(id, v)
```

The graph structure (nodes and edges) is just the skeleton. The actual *data*—the intent bytes—lives in an “attachment.”

Think of it like this: the node is the mailbox, and the attachment is the letter inside. The mailbox has a predictable address (the content-addressed ID), but the contents can be anything.

This separation is useful because you can query the graph structure without loading all the attachment data. For large payloads, that's a big memory savings.

#### [10] PENDING EDGE CREATION (Queue Membership)

```
pending_edge_id(&inbox_id, &intent_id) → EdgeId
FILE: crates/warp-core/src/inbox.rs:212
CODE: blake3("edge:" || "sim/inbox/pending:" || inbox_id || intent_id)
```

```

store.insert_edge(inbox_id, EdgeRecord {
    id: pending_edge_id,
    from: inbox_id,
    to: event_id,
    ty: make_type_id("edge:pending")
})

```

```
[11] return Ok(IngestDisposition::Accepted { intent_id })
```

**The Big Picture:** The “pending edge” is how Echo implements a queue using a graph.

The inbox node is the queue. Each pending edge from inbox to an event node represents “this event is waiting to be processed.” When a rule processes the event, it deletes the pending edge.

Why use a graph for a queue? Because now the queue is *part of the state that gets hashed and committed*. You can replay the entire system from any snapshot, and the queue will be exactly where it was.

No external message broker. No separate queue database. It’s all just graph.

### 1.3 Data Structures Modified

Structure	Field	Change
GraphStore	nodes	+3 entries (sim, inbox, event)
GraphStore	edges_from	+3 edges
GraphStore	edges_to	+3 reverse entries
GraphStore	edge_index	+3 edge→from mappings
GraphStore	edge_to_index	+3 edge→to mappings
GraphStore	node_attachments	+1 (event → intent payload)

## 2. Transaction Lifecycle

### 2.1 Begin Transaction

**Entry Point:** `Engine::begin()`

**File:** `crates/warp-core/src/engine_impl.rs:711-719`

```
pub fn begin(&mut self) -> TxId {
    self.tx_counter = self.tx_counter.wrapping_add(1); // Line 713
    if self.tx_counter == 0 {
        self.tx_counter = 1; // Line 715
    }
    self.live_txs.insert(self.tx_counter); // Line 717
    TxId::from_raw(self.tx_counter) // Line 718
}
```

This is refreshingly simple for a transaction begin, right? Just increment a counter and track it in a set.

But look at line 715. What's up with that `if tx_counter == 0` check?

Here's the deal: `TxId(0)` is reserved as an invalid/sentinel value throughout the codebase. It means "no transaction" or "null transaction." If you ever wrap around from `u64::MAX` back to 0, you'd suddenly have a valid-looking transaction ID that's actually invalid.

Now, will you ever hit  $2^{64}$  transactions? Almost certainly not. The sun will burn out first. But this check costs one branch that's basically never taken, and it eliminates an entire class of potential bugs.

This is defensive programming done right. The cost is negligible, and the safety is real.

**Pro Tip:** See that `#[repr(transparent)]` on `TxId`? That guarantees it has the exact same memory layout as a raw `u64`. You get type safety at compile time with zero runtime overhead. Use newtypes liberally—they're free!

### 2.2 Abort Transaction

**Entry Point:** `Engine::abort()`

**File:** `crates/warp-core/src/engine_impl.rs:962-968`

```
pub fn abort(&mut self, tx: TxId) {
    self.live_txs.remove(&tx.value());
    self.scheduler.finalize_tx(tx);
}
```

```

    self.bus.clear();
    self.last_materialization.clear();
    self.last_materialization_errors.clear();
}

```

Notice what's *not* here: there's no rollback of graph state.

Why? Because Echo hasn't touched the graph yet! All the matching and scheduling happens without mutating anything. The graph only changes during commit.

This is a fundamental architectural decision: **the graph is effectively immutable until commit**. You can abort at any point before commit and there's nothing to undo. Just clear the transient state and you're done.

Compare this to traditional databases where abort might mean replaying a undo log. Here it's just clearing some hash maps.

### 3. Rule Matching

**Entry Point:** `Engine::apply()`

**File:** `crates/warp-core/src/engine_impl.rs:730-737`

**The Big Picture:** Rules are the heart of Echo's reactive programming model. A rule says "when you see this pattern in the graph, do this thing."

But here's the key insight: matching is **pure**. The matcher function reads the graph, decides if the pattern matches, but doesn't modify anything. All the mutations happen later, during execution.

This separation of matching from execution is what enables parallel scheduling.

#### 3.1 Function Signature

```

pub fn apply(
    &mut self,
    tx: TxId,
    rule_name: &str,
    scope: &NodeId,
) -> Result<ApplyResult, EngineError>

```

#### 3.2 Key Steps

`Engine::apply(tx, rule_name, scope)`

## [4] CREATE GRAPHVIEW

```

    GraphView::new(store) → GraphView<'_>
    FILE: crates/warp-core/src/graph_view.rs
    TYPE: Read-only wrapper (Copy, 8 bytes)

```

This is one of my favorite patterns in Echo.

`GraphView` is a *read-only wrapper* around `GraphStore`. It's literally just a pointer (8 bytes), and it implements `Copy`, so passing it around is essentially free.

But here's the magic: `GraphView` only exposes read methods. No mutations. The Rust compiler *physically cannot* let you modify the graph through a `GraphView`.

This is Rust's type system doing real work. You don't need runtime checks for "is this a read-only transaction?" The type system guarantees it at compile time. Any code that takes a `GraphView` is provably read-only.

## [5] CALL MATCHER

```

    (rule.matcher)(view, scope) → bool
    TYPE: MatchFn = for<'a> fn(GraphView<'a>, &NodeId) -> bool
    IF false: return Ok(ApplyResult::NoMatch)

```

## [8] COMPUTE FOOTPRINT

```

    (rule.compute_footprint)(view, scope) → Footprint
    RETURNS:

```

```

    Footprint {
        n_read: IdSet,           // Nodes read
        n_write: IdSet,          // Nodes written
        e_read: IdSet,           // Edges read
        e_write: IdSet,          // Edges written
        a_read: AttachmentSet,   // Attachments read
        a_write: AttachmentSet,  // Attachments written
        b_in: PortSet,           // Input ports
        b_out: PortSet,          // Output ports
        factor_mask: u64,        // 0(1) prefilter
    }

```

The footprint is the **declaration of intent**.

Before a rule can execute, it must tell the scheduler exactly which nodes, edges, and attachments it plans to read and write. Not approximately. Not "somewhere

in this subgraph.” *Exactly* these IDs.

This is a constraint on rule authors, but it’s what makes parallelism tractable. If two rules have non-overlapping footprints, they can run concurrently. If they overlap, the scheduler serializes them.

Think of it like declaring your locks upfront, except you never actually acquire locks—you just declare your intentions and let the scheduler figure out what can run in parallel.

**Heads Up:** If your footprint is wrong—if you access something you didn’t declare—Bad Things happen. The parallel execution model assumes footprints are honest. There’s debug-mode validation, but in release mode, you’re on the honor system.

Always over-declare rather than under-declare. If you *might* read a node, put it in `n_read`. Correctness beats parallelism.

```
[11] ENQUEUE TO SCHEDULER
    self.scheduler.enqueue(tx, PendingRewrite { ... })

PendingTx::enqueue(scope_be32, rule_id, payload)
FILE: crates/warp-core/src/scheduler.rs:331-355

CASE 1: Duplicate (scope_hash, rule_id) - LAST WINS
    fat[thin[i].handle] = Some(payload) // Overwrite
    thin[i].nonce = next_nonce++        // Refresh nonce

CASE 2: New entry
    fat.push(Some(payload))
    thin.push(RewriteThin { scope_be32, rule_id, nonce, handle })
    index.insert(key, thin.len() - 1)
```

See “LAST WINS” on duplicate entries? This is subtle but important.

If you call `apply()` twice with the same rule and scope, you get one execution, not two. The second call *replaces* the first.

Why? Because matching a rule at a scope is *idempotent*. If the rule matches at that scope, you want to execute it once, regardless of how many times you tried to apply it.

The “nonce” gets refreshed on replacement, which affects sort order (we’ll see

why later), but the key point is: duplicate apply calls are collapsed into one.

## 4. Scheduler: Drain & Reserve

**The Big Picture:** This is where Echo's determinism guarantee gets forged.

You've enqueued a bunch of rules. They were enqueued in whatever order the application called `apply()`. Now we need to execute them in a **canonical order**—the same order every time, regardless of timing, regardless of which thread called what when.

The scheduler does this in two phases:

1. **Drain:** Sort all pending rewrites into canonical order
2. **Reserve:** Walk through them, checking for conflicts

### 4.1 Drain Phase (Radix Sort)

**Entry Point:** `RadixScheduler::drain_for_tx()`

**File:** `crates/warp-core/src/scheduler.rs:109-113`

`RadixScheduler::drain_for_tx(tx)`

`PendingTx::drain_in_order()`

DECISION: `n <= 1024 (SMALL_SORT_THRESHOLD)?`

YES: `sort_unstable_by(cmp_thin)`

NO: `radix_sort()`

`radix_sort()`

FOR `pass` IN `0..20`: // 20 PASSES

PHASE 1: COUNT BUCKETS

PHASE 2: PREFIX SUMS

PHASE 3: STABLE SCATTER

Twenty passes of radix sort. Let's unpack why.

First: why radix sort instead of quicksort or mergesort?

1. **Determinism:** Radix sort is inherently stable—equal elements stay in their original order. Quicksort’s behavior depends on pivot selection, which can vary.
2.  **$O(n)$  complexity:** With fixed key size, radix sort is linear. We’re sorting by 160 bits (128 bits of `scope_hash` + 32 bits of `rule_id`), so it’s  $O(20n) = O(n)$ .
3. **Cache-friendly:** Each pass is a sequential scan. Modern CPUs love sequential access.

The 1024-element threshold is practical: for small arrays, the constant factors of radix sort (setting up histograms, etc.) exceed its benefits. Below that threshold, a comparison sort wins.

BUCKET EXTRACTION (`bucket16`):

FILE: `crates/warp-core/src/scheduler.rs:481-498`

```
Pass 0:  u16_from_u32_le(r.nonce, 0)      // Nonce bytes [0:2]
Pass 1:  u16_from_u32_le(r.nonce, 1)      // Nonce bytes [2:4]
Pass 2:  u16_from_u32_le(r.rule_id, 0)    // Rule ID bytes [0:2]
Pass 3:  u16_from_u32_le(r.rule_id, 1)    // Rule ID bytes [2:4]
Pass 4:  u16_be_from_pair32(scope, 15)    // Scope bytes [30:32]
...
Pass 19: u16_be_from_pair32(scope, 0)     // Scope bytes [0:2] (MSD)
```

SORT ORDER: (`scope_hash`, `rule_id`, `nonce`) ascending lexicographic

This is LSD (Least Significant Digit) radix sort—we process from least significant to most significant.

The final sort order is: (`scope_hash`, `rule_id`, `nonce`).

Why this order?

- **`scope_hash` first:** Rules at different scopes can potentially run in parallel. Grouping by scope makes conflict detection efficient.
- **`rule_id` second:** When multiple rules match at the same scope, we need a deterministic order.
- **`nonce` last:** The tiebreaker for duplicate (`scope`, `rule`) pairs. Remember “LAST WINS”? The nonce determines which duplicate survives.

Because it's LSD, we process in reverse order: nonce first (passes 0-1), then rule\_id (passes 2-3), then scope\_hash (passes 4-19).

## 4.2 Reserve Phase (Independence Check)

**Entry Point:** RadixScheduler::reserve()

**File:** crates/warp-core/src/scheduler.rs:134-143

```
pub(crate) fn reserve(&mut self, tx: TxId, pr: &mut PendingRewrite) -> bool {
    let active = self.active.entry(tx).or_insert_with(ActiveFootprints::new);
    if Self::has_conflict(active, pr) {
        return Self::on_conflict(pr);
    }
    Self::mark_all(active, pr);
    Self::on_reserved(pr)
}
```

This is classic two-phase locking... without the locks.

We walk through the sorted rewrites. For each one:

1. Check if its footprint conflicts with already-reserved footprints
2. If no conflict, mark its footprint as reserved and accept it
3. If conflict, reject it (it'll need to wait for a future tick)

The conflict matrix is what you'd expect:

	Read	Write
Read	✓	X
Write	X	X

Multiple readers are fine. Any writer conflicts with readers and other writers.

## 4.3 GenSet: O(1) Conflict Detection

**File:** crates/warp-core/src/scheduler.rs:509-535

```
pub(crate) struct GenSet<K> {
    gen: u32, // Current generation
    seen: FxHashMap<K, u32>, // Key → generation when marked
}

impl<K: Hash + Eq + Copy> GenSet<K> {
    pub fn contains(&self, key: K) -> bool {
```

```

        matches!(self.seen.get(&key), Some(&g) if g == self.gen)
    }

    pub fn mark(&mut self, key: K) {
        self.seen.insert(key, self.gen);
    }
}

```

Okay, this is my favorite data structure in the entire codebase. It's so simple and so clever.

The problem: we need to track which keys are “in the set” for conflict detection. Between transactions, we need to clear the set.

The naive approach: call `hash_map.clear()` between transactions. That's  $O(n)$  where  $n$  is the number of keys.

The clever approach: **generational clearing**.

Instead of storing just keys, we store (key, generation). A key is “in the set” only if its stored generation matches the current generation.

To “clear” the set? Just increment `gen`. That's it.  $O(1)$ .

All the old entries are still in the hash map, but they have stale generations, so `contains()` returns false for them. They're ghosts.

The map grows over time, but since the same keys tend to be accessed repeatedly (temporal locality), it stabilizes quickly. And we never pay the  $O(n)$  clear cost.

This pattern is criminally underused. Remember it.

## 5. BOAW Parallel Execution

**Entry Point:** `execute_parallel()`

**File:** `crates/warp-core/src/boaw/exec.rs:61-83`

**The Big Picture:** BOAW stands for “Bag of Asynchronous Work.” The idea is simple but powerful:

1. Partition work items into shards based on their scope
2. Spin up worker threads
3. Workers claim shards and execute items

#### 4. Merge all the outputs into a single canonical result

The key insight: **execution order doesn't matter if we sort the outputs**. Workers can execute in any order, claim shards in any order, even race against each other—as long as the merge produces the same result, we're deterministic.

### 5.1 Entry Point

```
pub fn execute_parallel(view: GraphView<'_,>, items: &[ExecItem], workers: usize) ->
```

### 5.2 Sharding

```
partition_into_shards(items.to_vec()) → Vec<VirtualShard>
```

```
FOR item IN items:
```

```
    shard_of(&item.scope) → usize
```

```
    CODE:
```

```
        let bytes = scope.as_bytes();
        let first_8: [u8; 8] = [bytes[0..8]];
        let val = u64::from_le_bytes(first_8);
        (val & 255) as usize // SHARD_MASK = 255
```

```
    shards[shard_id].items.push(item)
```

The sharding is beautifully simple: take the first 8 bytes of the node ID, interpret as a little-endian u64, mask with 255. You get a shard number from 0 to 255.

Why 256 shards?

- **Fine enough:** With random node IDs, work distributes evenly across shards.
- **Coarse enough:** Each shard has multiple items, amortizing per-shard overhead.
- **Power of 2:** Masking is just a bitwise AND, no division needed.

Why is this deterministic? Because shard assignment depends only on the node ID, which is content-addressed. The same node always lands in the same shard.

### 5.3 Work Stealing Loop

```
FOR _ IN 0..workers:
```

```
s.spawn(move || { ... }) // WORKER THREAD

LOOP:

    shard_id = next_shard.fetch_add(1, Ordering::Relaxed)
    ATOMIC: Returns old value, increments counter

    IF shard_id >= 256: break

    FOR item IN &shards[shard_id].items:
        (item.exec)(view, &item.scope, &mut delta)
```

Each worker runs a loop: atomically claim the next shard number, process all items in that shard, repeat until no shards remain.

See `Ordering::Relaxed`? That's the weakest memory ordering—basically “no synchronization, just do the atomic operation.”

Why is that safe here?

1. Each shard is processed by exactly one worker (atomic fetch-add guarantees unique assignment)
2. Workers don't need to see each other's results until after `join()`
3. The `join()` provides the synchronization barrier

Using `Relaxed` instead of `SeqCst` avoids expensive memory barriers. On a 16-core machine, that matters.

**Heads Up:** The shard claim order is non-deterministic. Worker 1 might claim shard 5 before worker 2 claims shard 3, or vice versa.

This is fine! The merge phase sorts the outputs canonically. The execution order doesn't affect the final result.

But if you're debugging and wondering why execution traces look different between runs, this is why.

---

## 6. Delta Merge & State Finalization

**The Big Picture:** Multiple workers have produced their deltas. Now we need to merge them into a single canonical result.

The merge does three things:

1. Flatten all operations from all deltas
2. Sort them by a canonical key
3. Deduplicate, detecting conflicts along the way

## 6.1 Canonical Merge

**Entry Point:** `merge_deltas()`

**File:** `crates/warp-core/src/boaw/merge.rs:36-75`

```
merge_deltas(deltas: Vec<TickDelta>) → Result<Vec<WarpOp>, MergeConflict>
```

[1] FLATTEN ALL OPS WITH ORIGINS

[2] CANONICAL SORT

```
flat.sort_by(|a, b| (&a.0, &a.1).cmp(&(&b.0, &b.1)));
ORDER: (WarpOpKey, OpOrigin) lexicographic
```

[3] DEDUPE & CONFLICT DETECTION

GROUP by WarpOpKey

IF all ops in group are identical: keep one

ELSE: return Err(MergeConflict { writers })

The magic is in step 3: **benevolent coincidence**.

If two rules independently decide to create the same edge, with the same properties, that's fine! They're in agreement. We keep one copy.

But if they produce *different* operations for the same key—say, one sets an attachment to value A and another sets it to value B—that's a conflict. The rules disagree, and we can't pick a winner.

This policy allows natural redundancy in rule definitions. Multiple rules can create the same structural elements without coordinating. As long as they agree on the result, it works.

Conflicts indicate a bug in rule definitions. The receipt includes the conflicting writers so you can debug.

## 6.2 Operation Ordering

```
pub(crate) fn sort_key(&self) -> WarpOpKey {
    match self {
        Self::OpenPortal { .. }      => WarpOpKey { kind: 1, ... },
        Self::UpsertWarpInstance { .. } => WarpOpKey { kind: 2, ... },
        Self::DeleteWarpInstance { .. } => WarpOpKey { kind: 3, ... },
        Self::DeleteEdge { .. }      => WarpOpKey { kind: 4, ... },
        Self::DeleteNode { .. }      => WarpOpKey { kind: 5, ... },
        Self::UpsertNode { .. }      => WarpOpKey { kind: 6, ... },
        Self::UpsertEdge { .. }      => WarpOpKey { kind: 7, ... },
        Self::SetAttachment { .. }   => WarpOpKey { kind: 8, ... },
    }
}
```

The operation order is carefully chosen to maintain invariants:

1. **OpenPortal first:** Creates warp instances that later ops may reference
2. **Deletes before upserts:** If you delete then upsert the same thing, you get a fresh entity. If you upsert then delete, you get nothing. Deletes first is the saner default.
3. **Nodes before edges:** Edges reference nodes, so nodes must exist first
4. **Attachments last:** Attachments attach to nodes/edges, so the skeleton must exist

This ordering means rules can emit ops in any order. The merge sorts them into the correct sequence. One less thing for rule authors to worry about.

## 7. Hash Computation

**The Big Picture:** Echo uses hashing for two things:

1. **State root:** A fingerprint of what the graph looks like right now
2. **Commit hash:** A fingerprint of this entire commit (state + how we got here)

If two nodes compute the same commit hash, they have identical state. This is how consensus works without comparing the full state.

## 7.1 State Root

**Entry Point:** `compute_state_root()`

**File:** `crates/warp-core/src/snapshot.rs:88-209`

```
compute_state_root(state: &WarpState, root: &NodeKey) → Hash
```

[1] BFS REACHABILITY TRAVERSAL

Only hash nodes/edges reachable from root

[2] HASHING PHASE

```
FOR warp_id IN reachable_warps: // BTreeSet = sorted order
  FOR (node_id, node) IN store.nodes: // BTreeMap = sorted
    hash(node_id, node.type, attachment)
  FOR (from, edges) IN store.edges_from: // BTreeMap = sorted
    sorted_edges = edges.sort_by(id)
    hash(from, edges)
```

```
hasher.finalize().into()
```

Two critical details here:

**1. Reachability:** We only hash nodes/edges reachable from the root via BFS. Unreachable “garbage” doesn’t affect the hash.

This is subtle but important. It means you can safely delete subgraphs without affecting the hash of nodes that don’t reference them. It’s also the foundation for garbage collection—unreachable data can be purged without breaking consensus.

**2. BTreeMap/BTreeSet:** Notice the iteration is over B-tree collections, not hash maps.

Why? Because B-trees iterate in *sorted order*. Hash maps iterate in arbitrary order (based on hashing, which might differ between machines or Rust versions).

If we used hash maps, two machines with identical state might produce different hashes just because they iterated in different orders. That would be catastrophic.

BTreeMap/BTreeSet cost  $O(\log n)$  instead of  $O(1)$  for operations, but they guarantee deterministic iteration. For hashing, that’s non-negotiable.

## 7.2 Commit Hash v2

**Entry Point:** `compute_commit_hash_v2()`

**File:** `crates/warp-core/src/snapshot.rs:244-263`

```
fn compute_commit_hash_v2(
    state_root: &Hash,
    parents: &[Hash],
    patch_digest: &Hash,
    policy_id: u32,
) -> Hash {
    let mut h = Hasher::new();
    h.update(&2u16.to_le_bytes());           // Version tag
    h.update(&(parents.len() as u64).to_le_bytes()); // Parent count
    for p in parents { h.update(p); }         // Parents
    h.update(state_root);                     // State
    h.update(patch_digest);                   // Operations
    h.update(&policy_id.to_le_bytes());       // Policy
    h.finalize().into()
}
```

The commit hash includes:

- **state\_root:** What the graph looks like
- **patch\_digest:** What operations got us here
- **parents:** Which commit(s) we're building on
- **policy\_id:** Which policy version we're using

The 2u16 version tag is future-proofing. If we ever need to change the commit hash format, we bump the version. Old and new formats produce different hashes, which is correct—they're different protocols.

Everything is little-endian (`to_le_bytes()`) because we need byte-identical encoding across platforms. Big-endian and little-endian machines must produce the same hash.

## 8. Commit Orchestration

**Entry Point:** `Engine::commit_with_receipt()`

**File:** `crates/warp-core/src/engine_impl.rs:837-954`

**The Big Picture:** This is the grand finale. All the pieces come together:

1. Drain the scheduler (get sorted rewrites)
2. Reserve (check for conflicts)
3. Execute (run the rules, collect deltas)
4. Merge (combine deltas canonically)
5. Apply (mutate the graph)
6. Hash (compute state root and commit hash)
7. Record (save to history)

If any step fails, we haven't mutated anything permanent. The graph only changes when everything succeeds.

```
Engine::commit_with_receipt(tx)
```

```
[2] DRAIN CANDIDATES
```

```
    drained = self.scheduler.drain_for_tx(tx)
```

```
[3] RESERVE (INDEPENDENCE CHECK)
```

```
    FOR rewrite IN drained:
```

```
        accepted = self.scheduler.reserve(tx, &mut rewrite)
```

```
[4] EXECUTE
```

```
    state_before = self.state.clone() // Snapshot before mutation!
```

```
    FOR rewrite IN reserved:
```

```
        (executor)(view, &scope, &mut delta)
```

```
    delta.finalize()
```

```
    patch.apply_to_state(&mut self.state)
```

```
[6] COMPUTE DELTA PATCH
```

```
    ops = diff_state(&state_before, &self.state)
```

```
[7] COMPUTE STATE ROOT
```

```
    state_root = compute_state_root(&self.state, &root)
```

```
[10] COMPUTE COMMIT HASH
```

```
    hash = compute_commit_hash_v2(state_root, parents, patch_digest, policy_id)
```

[12] RECORD TO HISTORY

```
tick_history.push((snapshot, receipt, patch))
```

See `state_before = self.state.clone()` in step [4]?

We snapshot the state *before* executing anything. This enables:

1. `diff_state()`: Compare before/after to get the actual ops
2. Validation: The delta from execution should match the diff
3. Potential rollback: If something goes wrong, we have the original

The clone isn't as expensive as it looks. Rust's clone is typically copy-on-write under the hood for large structures. Most of the data is shared until mutation.

And that's it! That's the complete journey from user action to committed state.

Every step is deterministic. Every hash is content-addressed. The same inputs always produce the same outputs, regardless of timing, thread scheduling, or which machine runs the code.

This is what makes Echo special. It's not just a graph database. It's a *deterministic computation engine* that happens to store its state in a graph.

Thanks for sticking with me through this tour. Now go read the actual code—you'll understand it much better now.

## Appendix A: Complexity Summary

Operation	Complexity	Notes
<code>ingest_intent</code>	$O(1)$	Fixed structural insertions
<code>begin</code>	$O(1)$	Counter increment + set insert
<code>apply</code>	$O(m)$	$m$ = footprint size
<code>drain_for_tx</code>	$O(n)$	$n$ = candidates, 20 radix passes
<code>reserve</code> per rewrite	$O(m)$	$m$ = footprint size, $O(1)$ per check
<code>execute_parallel</code>	$O(n/w)$	$n$ = items, $w$ = workers
<code>merge_deltas</code>	$O(k \log k)$	$k$ = total ops
<code>compute_state_root</code>	$O(V + E)$	$V$ = nodes, $E$ = edges

Nothing quadratic. Nothing exponential. The system scales linearly with the amount of work. That's by design.

The one potential bottleneck is `compute_state_root`—it traverses the entire reachable graph. For very large graphs, that's expensive. In practice, graphs are partitioned across warp instances, keeping each traversal manageable.

## Appendix B: Determinism Boundaries

### Guaranteed Deterministic

- Radix sort ordering (20-pass LSD)
- BTreeMap/BTreeSet iteration
- BLAKE3 hashing
- GenSet conflict detection
- Canonical merge deduplication

### Intentionally Non-Deterministic (Handled by Merge)

- Worker execution order in BOAW
- Shard claim order (atomic counter)

The non-deterministic parts are carefully contained. Workers race against each other, but the merge absorbs that chaos and produces a deterministic result.

Think of it as a funnel: chaos at the wide end (parallel execution), order at the narrow end (merged output). The merge is the bottleneck that enforces determinism.

### Protocol Constants (Frozen)

- `NUM_SHARDS` = 256
- `SHARD_MASK` = 255
- Shard routing: `LE_u64(node_id[0..8]) & 255`
- Commit hash v2 version tag: `0x02 0x00`

**Heads Up:** These constants are **frozen**. Changing them would break compatibility with all existing commits.

If you're tempted to "optimize" by tweaking `NUM_SHARDS`, remember: every

historical commit was created with these values. Changing them makes replay impossible.

Protocol evolution happens through version tags, not constant changes.

---

*Document generated 2026-01-18. Director's commentary by your friendly AI pair programmer.*

