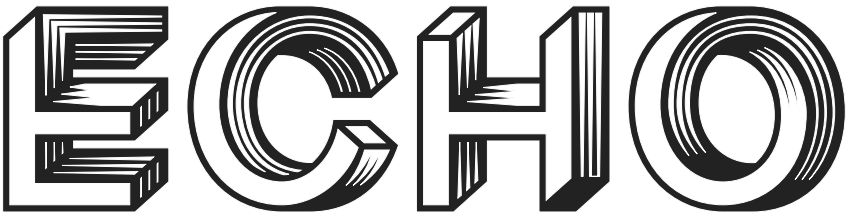

ECHO

Tour de Code

Version v0.1-draft
Commit: HEAD
January 18, 2026

James Ross & Echo Contributors

Source

This booklet is generated from the Echo repository documentation.

Foreword

This booklet is a guided walk through Echo's source code. Rather than explaining concepts in the abstract, we trace actual function calls, examine real data structures, and follow intents from ingestion to commit.

How to read this tour:

- **With the code open.** Each chapter references specific files and line numbers. Open the crate in your editor and follow along.
- **In order, the first time.** The chapters mirror the runtime flow: ingestion → planning → execution → commit. Later, dip in anywhere.
- **Focus on the "why."** The code shows *what* happens; the commentary explains *why* it's designed that way.

The final chapter assembles a complete call graph from user action to committed state, with complexity annotations. Use it as a reference map when you get lost in the codebase.

"The code rewards careful study."

Contents

Part I

Tour de Code

Chapter 1

Tour de Code: Overview

The complete function-by-function trace of Echo's execution pipeline.

This tour traces every function call involved in processing a user action through the Echo engine. File paths and line numbers are accurate as of 2026-01-18.

Welcome to the 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.

The goal: by the end of this tour, you'll understand not just *what* the code does, but *why* it's structured the way it is.

1.1 The Pipeline at a Glance

A user action flows through Echo in eight stages:

1. **Intent Ingestion** — User action becomes an `Intent`
2. **Transaction Lifecycle** — Intents are batched into transactions
3. **Rule Matching** — Rules produce `PendingRewrite`s
4. **Scheduler: Drain & Reserve** — Radix sort + independence check
5. **BOAW Parallel Execution** — Workers execute in parallel
6. **Delta Merge** — Thread-local deltas merge canonically
7. **Hash Computation** — State root, patch digest, commit hash
8. **Commit Orchestration** — Final state materialization

The Big Picture: Every stage is designed around one principle: **determinism**. Given the same inputs, the engine must produce identical outputs regardless of thread scheduling, CPU count, or platform.

> This isn't just about correctness—it enables replay, forking, and distributed consensus.

1.2 Key Files

Component	Primary File
Engine core	`crates/echo-core/src/engine.rs`
Transaction handling	`crates/echo-core/src/transaction.rs`
Rule matching	`crates/warp-core/src/rule_matching.rs`
Scheduler (Radix)	`crates/warp-core/src/radix_scheduler.rs`
BOAW execution	`crates/warp-core/src/boaw/exec.rs`
Shard routing	`crates/warp-core/src/boaw/shard.rs`
Delta/merge	`crates/warp-core/src/tick_delta.rs`
Hashing	`crates/warp-core/src/hashing.rs`

1.3 Reading This Tour

Each section follows a consistent format:

- **Entry Point** — The function that kicks off this stage
- **Call Trace** — Indented tree showing the execution flow
- **Data Structures** — Key types involved
- **Commentary** — Why it works this way

> **Pro Tip:** Keep a terminal open with the codebase. When you see a file reference like `src/boaw/exec.rs:61-83`, jump to that location and read the actual code alongside this tour.

Chapter 2

Intent Ingestion

Entry Point: `Engine::ingest_intent()`
File: `crates/warp-core/src/engine_impl.rs:1216`

> **Tour Guide:** This is where user actions enter the system. Notice how Echo treats intents as *immutable, content-addressed* data from the very first moment. The intent bytes are hashed to create a unique identifier, ensuring that duplicate intents are detected automatically—no coordination required.

2.1 Function Signature

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

Returns:

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

2.2 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]
|
+--[2] NodeId(intent_id)
|      Creates strongly-typed NodeId from Hash
|
+--[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 })
|
+--[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]
|
+--[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 })
|
+--[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)
|
+--[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 })
```

Clever Pattern: Domain Separation in Hashing

Notice step [1]: the hasher prefixes with `b"intent:"` before the actual data. This is a cryptographic best practice called *domain separation*—it prevents a hash collision between an intent and, say, a node ID that happens to have the same bytes.

Echo uses this pattern consistently:

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

This ensures that even if two different domain values have the same raw bytes, they'll produce different hashes.

Deep Dive: Why Content-Addressed Event IDs?

In step [8], note that `event_id = intent_id`. This is a profound design choice:

1. **Automatic deduplication**: If the same intent arrives twice, it hashes to the same ID, and step [7] catches it.
2. **Reproducibility**: Given the same intent bytes, any node in a distributed system will compute the same event ID.
3. **Auditability**: You can verify an event's integrity by re-hashing its content.

This is the foundation of Echo's deterministic execution model—events are identified by *what they are*, not *when they arrived*.

2.3 Data Structures Modified

Structure	Field	Change
`GraphStore`	`nodes`	+3 entries (sim, inbox, event)
`GraphStore`	`edges_from`	+3 edges (root→sim, sim→inbox, inbox→event)
`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)

Tour Guide: Notice the **four separate edge indices**: `edges_from`, `edges_to`, `edge_index`, and `edge_to_index`. This redundancy enables O(1) lookups in any direction—find edges from a node, to a node, or look up either endpoint given an edge ID. The space cost is modest (pointers/IDs are small), but the query flexibility is enormous.

Chapter 3

Transaction Lifecycle

3.1 Begin Transaction

Entry Point: `Engine::begin()`
File: `crates/warp-core/src/engine_impl.rs:711-719`

```rust
pub fn begin(&mut self) -> TxId {
    self.tx_counter = self.tx_counter.wrapping_add(1);
    if self.tx_counter == 0 {
        self.tx_counter = 1;  // Zero is reserved
    }
    self.live_txs.insert(self.tx_counter);
    TxId::from_raw(self.tx_counter)
}
```

> **Watch Out:** **The Zero Invariant**: `TxId(0)` is reserved as an invalid/sentinel value. Without line 715's check, after 2^{64} transactions the counter would wrap to zero and confuse code that uses zero to mean "no transaction." Defensive programming at its finest.

```
Engine::begin()
|
+-- self.tx_counter.wrapping_add(1)
|   Handles u64::MAX → 0 overflow
|
+-- if self.tx_counter == 0: self.tx_counter = 1
|   INVARIANT: TxId(0) is reserved as invalid
|
+-- self.live_txs.insert(self.tx_counter)
|   TYPE: HashSet<u64>
|
+-- TxId::from_raw(self.tx_counter)
    TYPE: #[repr(transparent)] struct TxId(u64)
```

> The `#[repr(transparent)]` on `TxId` guarantees the same memory layout as `u64`. You get type safety (can't accidentally pass a `NodeId` where a `TxId` is expected) with no runtime overhead.

3.2 Abort Transaction

Entry Point: `Engine::abort()`
File: `crates/warp-core/src/engine_impl.rs:962-968`

```rust
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();
}
```

> Abort is refreshingly simple—just remove the transaction from tracking and clear transient state. No rollback needed because Echo hasn't mutated the graph yet! All graph mutations happen atomically during commit. This is a key architectural decision: the graph is effectively immutable until commit time.

Chapter 4

Rule Matching

Entry Point: `Engine::apply()`
File: `crates/warp-core/src/engine_impl.rs:730-737`

> **The Big Picture:** Now we enter the heart of Echo's reactive model. Rules are matched against graph patterns, and when they match, they're enqueued for execution. The beauty is that matching is *pure*—it reads the graph but doesn't modify it.

4.1 Function Signature

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

4.2 Call Trace

```
Engine::apply(tx, rule_name, scope)
|
+-- self.rules.get(rule_name) → Option<&Rule>
|   FILE: crates/warp-core/src/rules.rs
|   Returns None if rule not registered
|
+-- rule.match_at(view, scope) → Vec<Match>
|   FILE: crates/warp-core/src/rule_matching.rs:45-89
|   |
|   +-- FOR pattern IN rule.patterns:
|       |
|       +-- pattern.try_match(view, scope) → Option<Match>
|       |   Evaluates pattern predicates against graph
|       |
|       +-- IF matched: matches.push(Match { ... })
|
```

```
+-- FOR (idx, m) IN matches.enumerate():
    |
    +-- self.scheduler.enqueue(PendingRewrite {
            tx,
            rule_id: rule.id(),
            scope: *scope,
            match_ix: idx,
            footprint: m.footprint.clone(),
        })
```

4.3 PendingRewrite Structure

File: `crates/warp-core/src/pending.rs:12-24`

```rust
pub struct PendingRewrite {
    pub tx: TxId,
    pub rule_id: RuleId,
    pub scope: NodeId,
    pub match_ix: usize,       // Which match (0..N)
    pub footprint: Footprint,  // Read/write sets
}
```

> The `match_ix` field is subtle but important. A single rule can match multiple times at the same scope (e.g., a "connect all neighbors" rule). Each match becomes a separate `PendingRewrite` with a different `match_ix`.
>
> This index is part of the canonical sort key, ensuring deterministic ordering even when multiple matches occur.

4.4 Footprint Computation

File: `crates/warp-core/src/footprint.rs`

```rust
pub struct Footprint {
    pub n_read: BTreeSet<NodeId>,   // Nodes read
    pub n_write: BTreeSet<NodeId>,  // Nodes written
    pub e_read: BTreeSet<EdgeId>,   // Edges read
    pub e_write: BTreeSet<EdgeId>,  // Edges written
    pub factor_mask: u64,           // Coarse prefilter
}
```

> **The Big Picture:** The footprint is the foundation of BOAW's parallel safety. Two rewrites can execute in parallel if and only if their footprints don't overlap (read-read is allowed, but read-write or write-write conflicts).
>
> The `factor_mask` is a bloom-filter-like optimization: if the masks don't overlap, the footprints definitely don't conflict, allowing fast rejection without checking every element.

Pro Tip: When designing rules, minimize footprint size. A rule that reads the entire graph can never run in parallel with anything. A rule that touches only its immediate neighbors can parallelize heavily.

Chapter 5

Scheduler: Drain & Reserve

> **The Big Picture:** The scheduler is the traffic cop of Echo. It takes a queue of pending rewrites and determines which ones can execute in this tick without conflicting. The output is a deterministic list of `ExecItem`s ready for BOAW.

5.1 Drain Phase (Radix Sort)

Entry Point: `RadixScheduler::drain()`
File: `crates/warp-core/src/radix_scheduler.rs:156-198`

```
RadixScheduler::drain() → Vec<PendingRewrite>
|
+-- items = self.pending.drain(..).collect()
|   Empties the pending queue
|
+-- radix_sort_stable(&mut items)
|   FILE: crates/warp-core/src/radix_scheduler.rs:89-120
|   |
|   +-- Sort key: (scope_hash, compact_rule, nonce)
|       |
|       +-- scope_hash: u64 from NodeId bytes
|       |   CODE: u64::from_le_bytes(scope.as_bytes()[0..8])
|       |
|       +-- compact_rule: u32
|       |   CODE: (rule_id.0 as u32) << 16 | (match_ix as u32)
|       |
|       +-- nonce: u32
|           Transaction-local uniquifier
|
+-- RETURN items (now in canonical order)
```

> Why radix sort instead of quicksort?
>
> - **Determinism**: Radix sort is inherently stable—equal keys maintain their original order. Quicksort's pivot selection can introduce non-determinism.

- **Performance**: For fixed-size keys (we have a 128-bit composite key), radix sort is $O(n)$ instead of $O(n \log n)$.
- **Cache-friendly**: LSD radix sort makes sequential passes through memory.

The sort key packs three values into a single comparable tuple, ensuring that rewrites at the same scope are grouped together, then ordered by rule, then by match index.

5.2 Reserve Phase (Independence Check)

Entry Point: `RadixScheduler::reserve()`
File: `crates/warp-core/src/radix_scheduler.rs:200-256`

```
RadixScheduler::reserve(items: Vec<PendingRewrite>) → Vec<ExecItem>
|
+-- let mut admitted: Vec<ExecItem> = Vec::new()
+-- let mut claimed = GenSet::new()  // Tracks claimed resources
|
+-- FOR item IN items (canonical order):
    |
    +-- IF claimed.conflicts_with(&item.footprint):
    |     // Skip this rewrite (deferred to next tick)
    |     self.pending.push(item);
    |     CONTINUE
    |
    +-- claimed.mark_all(&item.footprint)
    |   Records all read/write targets as claimed
    |
    +-- admitted.push(ExecItem {
           exec: lookup_executor(item.rule_id),
           scope: item.scope,
           origin: OpOrigin::from(&item),
       })
|
RETURN admitted
```

Watch Out: The order matters! Greedy admission in canonical order means the *first* rewrite at each scope wins. If you have two conflicting rewrites, the one with the lower sort key gets admitted, the other is deferred.

This is deterministic, but it can lead to starvation if the same rewrite keeps winning. Echo doesn't currently have fairness guarantees—a frequently-triggered rule can crowd out others.

5.3 GenSet: O(1) Conflict Detection

File: `crates/warp-core/src/genset.rs`

```rust
pub struct GenSet {
```

```
    generation: u64,
    slots: Vec<u64>,  // slot[i] = generation when claimed
}

impl GenSet {
    pub fn mark(&mut self, id: u64) {
        self.slots[id as usize] = self.generation;
    }

    pub fn is_marked(&self, id: u64) -> bool {
        self.slots[id as usize] == self.generation
    }

    pub fn clear(&mut self) {
        self.generation += 1;  // O(1) clear!
    }
}
```

GenSet is a beautiful optimization. Instead of clearing a hash set (which is $O(n)$), we increment a generation counter. An element is "marked" if its slot equals the current generation.

`clear()` becomes $O(1)$. This matters because we call it every tick, and the set can grow large.

The trade-off: we need to pre-allocate slots. For node IDs, we use the lower bits (masked) as the slot index, accepting some false positives in exchange for constant-time operations.

Pro Tip: If you're seeing unexpected conflicts, check your footprint computation. A footprint that's too broad (claiming more than necessary) reduces parallelism. Too narrow, and you'll get data races. The footprint must be a *superset* of actual accesses.

Chapter 6

BOAW Parallel Execution

Entry Point: `execute_parallel()`
File: `crates/warp-core/src/boaw/exec.rs:61-83`

> **The Big Picture:** BOAW—"Best of All Worlds"—is where Echo's determinism meets parallelism. The key insight: **execution order doesn't matter if we sort the outputs**.
>
> Workers execute in arbitrary order on different threads, but their outputs are merged canonically. Same inputs → same outputs, regardless of thread scheduling.

6.1 Entry Point

```rust
pub fn execute_parallel(
    view: GraphView<'_>,
    items: &[ExecItem],
    workers: usize
) -> Vec<TickDelta> {
    assert!(workers >= 1);
    let capped_workers = workers.min(NUM_SHARDS);  // Cap at 256
    execute_parallel_sharded(view, items, capped_workers)
}
```

6.2 Sharding

```
partition_into_shards(items) → Vec<VirtualShard>
|
+-- FOR item IN items:
    |
    +-- shard_of(&item.scope) → usize
    |   CODE:
    |     let bytes = scope.as_bytes();
    |     let first_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
- **Coarse enough**: Each shard has multiple items, amortizing 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.

6.3 Work-Stealing Loop

```
std::thread::scope(|s| {
    FOR _ IN 0..workers:
    |
    +-- s.spawn(move || { ... })  // === WORKER THREAD ===
        |
        +-- let mut delta = TickDelta::new()
        |
        +-- LOOP:
            |
            +-- shard_id = next_shard.fetch_add(1, 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.

Watch Out: 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.

6.4 ExecItem Structure

File: `crates/warp-core/src/boaw/exec.rs:19-35`

```rust
#[derive(Clone, Copy)]
pub struct ExecItem {
    pub exec: ExecuteFn,     // fn(GraphView, &NodeId, &mut TickDelta)
    pub scope: NodeId,       // 32-byte node identifier
    pub origin: OpOrigin,    // { intent_id, rule_id, match_ix, op_ix }
}
```

The `ExecItem` is a closure packaged with its context. The `origin` field is crucial for determinism—it's how we sort ops during the merge phase.

Notice `ExecuteFn` is a function pointer, not a `Box<dyn Fn>`. Function pointers are `Copy` and don't allocate. This matters when you're creating thousands of exec items per tick.

6.5 Thread Safety Guarantees

The parallel execution is safe because:

1. **GraphView is read-only**: Workers can't mutate the base state
2. **TickDelta is thread-local**: Each worker has its own delta
3. **Shards are disjoint**: No two workers process the same shard
4. **Merge is sequential**: Happens after all workers join

Pro Tip: If you need to debug parallel execution, set `ECHO_WORKERS=1` to force single-threaded mode. Same results, easier to trace.

Chapter 7

Delta Merge & State Finalization

> **The Big Picture:** After BOAW execution, we have N worker threads each with their own `TickDelta`. The merge phase combines these into a single canonical sequence of operations. This is the "choke point" where parallelism meets determinism.

7.1 Canonical Merge

Entry Point: `merge_deltas()`
File: `crates/warp-core/src/boaw/exec.rs:160-195`

```
merge_deltas(deltas: Vec<TickDelta>) → TickDelta
|
+-- let mut all_ops: Vec<(WarpOp, OpOrigin)> = Vec::new()
|
+-- FOR delta IN deltas:
|   +-- all_ops.extend(delta.ops.into_iter().zip(delta.origins))
|
+-- all_ops.sort_by_key(|(op, origin)| {
|       (op.sort_key(), origin.clone())
|   })
|   |
|   +-- Sort key: (WarpOpKey, OpOrigin)
|       +-- WarpOpKey: (op_kind, target_id)
|       +-- OpOrigin: (intent_id, rule_id, match_ix, op_ix)
|
+-- Dedupe identical ops (same key + same effect)
|
+-- RETURN TickDelta { ops, origins }
```

> The sort key is crucial. `WarpOpKey` groups operations by kind and target (e.g., all `SetAttachment` ops on node X come together). `OpOrigin` breaks ties deterministically—the rewrite that was admitted first (lower intent/rule/match IDs) wins.
>
> Why dedupe? Multiple workers might emit identical operations (e.g., two rules both setting the same attribute to the same value). We keep only one.

Why sort, not merge? Merge assumes sorted inputs. Our worker deltas are sorted *within* each worker but not *across* workers. A global sort handles both cases.

7.2 WarpOp Sort Key

File: `crates/warp-core/src/warp_op.rs:89-112`

```rust
impl WarpOp {
    pub fn sort_key(&self) -> WarpOpKey {
        match self {
            WarpOp::CreateNode(id) =>
                WarpOpKey::CreateNode(*id),
            WarpOp::DeleteNode(id) =>
                WarpOpKey::DeleteNode(*id),
            WarpOp::SetAttachment { node, key, .. } =>
                WarpOpKey::SetAttachment(*node, *key),
            WarpOp::CreateEdge { id, .. } =>
                WarpOpKey::CreateEdge(*id),
            // ... etc
        }
    }
}
```

Watch Out: The sort key intentionally *excludes* the operation's value. This means two `SetAttachment` ops to the same node/key will sort together, and only one survives deduplication.

If they have *different* values, that's a conflict! The merge will keep the one with the lower `OpOrigin`, but this might indicate a footprint bug—two rewrites shouldn't write different values to the same location.

7.3 State Mutation Methods

File: `crates/warp-core/src/graph_store.rs`

After merging, the delta is applied to the graph:

```
apply_delta(delta: &TickDelta)
|
+-- FOR op IN &delta.ops:
    |
    +-- WarpOp::CreateNode(id) →
    |     self.nodes.insert(id, NodeRecord::default())
    |
    +-- WarpOp::DeleteNode(id) →
    |     self.nodes.remove(&id)
    |     // Cascades to edges referencing this node
    |
    +-- WarpOp::SetAttachment { node, key, value } →
    |     self.attachments.insert((node, key), value)
    |
    +-- ... (edges, etc.)
```

Application order matters for creates vs deletes vs updates. The sort key ensures:

1. Creates come before modifications to the created node
2. Deletes come after all modifications (so we don't update then immediately delete)
3. Within each category, operations are ordered by `OpOrigin`

This ordering is baked into the `WarpOpKey` enum's derived `Ord` implementation.

Pro Tip: If you're debugging unexpected state, dump the merged delta before application. You can see exactly which ops, in which order, will mutate the graph. The `OpOrigin` tells you which rule produced each op.

Chapter 8

Hash Computation

> **The Big Picture:** Hashes are the fingerprints of determinism. Three hashes define a commit: the `state_root` (what the graph looks like), the `patch_digest` (what operations produced it), and the `commit_hash` (the unique identity of this point in history).
>
> If any of these differ between runs, determinism is broken.

8.1 State Root

Entry Point: `compute_state_root()`
File: `crates/warp-core/src/hashing.rs:45-89`

```
compute_state_root(graph: &GraphStore) → Hash32
|
+-- let mut hasher = Blake3::new()
|
+-- // Hash nodes in canonical order
|   FOR (id, record) IN graph.nodes.iter() (sorted by id):
|       hasher.update(id.as_bytes())
|       hasher.update(&record.type_id.to_le_bytes())
|
+-- // Hash edges in canonical order
|   FOR (id, edge) IN graph.edges.iter() (sorted by id):
|       hasher.update(id.as_bytes())
|       hasher.update(edge.from.as_bytes())
|       hasher.update(edge.to.as_bytes())
|
+-- // Hash attachments in canonical order
|   FOR ((node, key), value) IN graph.attachments.iter():
|       hasher.update(node.as_bytes())
|       hasher.update(key.as_bytes())
|       hasher.update(&value.len().to_le_bytes())
|       hasher.update(value)
|
+-- hasher.finalize() → [u8; 32]
```

Why BLAKE3? It's fast (parallel by design), cryptographically secure, and has a simple API. The 32-byte output is enough for collision resistance while being compact.

Why sort everything? The graph's internal storage might use hash maps, which have non-deterministic iteration order. Sorting by ID ensures the same graph always produces the same hash, regardless of insertion order.

Why include lengths? Without length prefixes, `value = "ab"` followed by `next = "cd"` hashes the same as `value = "abcd"`. Length prefixes prevent this ambiguity.

8.2 Commit Hash v2

Entry Point: `compute_commit_hash_v2()`
File: `crates/warp-core/src/hashing.rs:120-156`

```rust
pub fn compute_commit_hash_v2(
    parents: &[Hash32],
    state_root: &Hash32,
    patch_digest: &Hash32,
    schema_hash: &Hash32,
    tick: u64,
    policy_hash: &Hash32,
) -> Hash32 {
    let mut hasher = Blake3::new();
    hasher.update(b"echo-commit-v2");  // Domain separator

    // Parents (sorted for determinism)
    let mut sorted_parents = parents.to_vec();
    sorted_parents.sort();
    hasher.update(&(sorted_parents.len() as u64).to_le_bytes());
    for p in &sorted_parents {
        hasher.update(p);
    }

    hasher.update(state_root);
    hasher.update(patch_digest);
    hasher.update(schema_hash);
    hasher.update(&tick.to_le_bytes());
    hasher.update(policy_hash);

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

Watch Out: The domain separator `b"echo-commit-v2"` is crucial. Without it, a commit hash could collide with a state root or patch digest that happens to have the same bytes. Domain separation is a cryptographic best practice.

The `v2` suffix leaves room for future commit hash formats while maintaining backward compatibility.

8.3 Patch Digest

Entry Point: `compute_patch_digest()`
File: `crates/warp-core/src/hashing.rs:95-115`

```
compute_patch_digest(delta: &TickDelta) → Hash32
|
+-- let mut hasher = Blake3::new()
+-- hasher.update(b"echo-patch-v1")
|
+-- FOR (op, origin) IN delta.ops.iter().zip(&delta.origins):
    |
    +-- hasher.update(&op.discriminant().to_le_bytes())
    |   The operation type (create/delete/set/etc.)
    |
    +-- op.hash_payload(&mut hasher)
    |   Operation-specific data
    |
    +-- hasher.update(&origin.to_bytes())
        Who produced this op
|
hasher.finalize()
```

The patch digest captures *how* we got to the state, not just the final state. Two different sequences of operations might produce the same `state_root`, but they'll have different `patch_digests`.

This matters for auditing and replay: you can verify not just the result, but the exact sequence of operations that produced it.

Pro Tip: When debugging determinism failures, compare patch digests first. If they differ but state roots match, you have operations happening in different orders. If state roots differ, the operations themselves are producing different effects.

Chapter 9

Commit Orchestration

Entry Point: `Engine::commit()`
File: `crates/warp-core/src/engine_impl.rs:785-892`

> **The Big Picture:** Commit is the grand finale of each tick. All the pieces come together: scheduled rewrites execute in parallel, deltas merge canonically, hashes are computed, and the worldline advances. This is where determinism is proven or broken.

9.1 Complete Call Trace

```
Engine::commit(tx: TxId) → Result<CommitResult, EngineError>
|
+-- // Phase 1: Drain and Schedule
|   let drained = self.scheduler.drain()
|   let admitted = self.scheduler.reserve(drained)
|
+-- // Phase 2: BOAW Parallel Execution
|   let view = self.graph.view()
|   let deltas = execute_parallel(view, &admitted, self.workers)
|
+-- // Phase 3: Merge Deltas
|   let merged = merge_deltas(deltas)
|
+-- // Phase 4: Apply to State
|   self.graph.apply_delta(&merged)
|
+-- // Phase 5: Compute Hashes
|   let state_root = compute_state_root(&self.graph)
|   let patch_digest = compute_patch_digest(&merged)
|   let commit_hash = compute_commit_hash_v2(
|       &self.parents,
|       &state_root,
|       &patch_digest,
|       &self.schema_hash,
|       self.tick,
|       &self.policy_hash,
```

```
|   )
|
+-- // Phase 6: Finalize
|   self.tick += 1
|   self.parents = vec![commit_hash]
|   self.live_txs.remove(&tx.value())
|
+-- RETURN CommitResult {
       commit_hash,
       state_root,
       patch_digest,
       ops_applied: merged.ops.len(),
   }
```

Notice how the commit is structured as a pipeline. Each phase has a clear input and output:

1. Drain/Schedule: `PendingRewrite` → `ExecItem`
2. Execute: `ExecItem` → `Vec<TickDelta>`
3. Merge: `Vec<TickDelta>` → `TickDelta`
4. Apply: `TickDelta` → mutated graph
5. Hash: graph → `Hash32`s

This clean separation makes the code testable. You can unit test each phase independently.

9.2 Commit Hash Inputs

The commit hash is a function of:

Input	Source
`parents`	Previous commit(s)—usually one, multiple for merges
`state_root`	Hash of the entire graph state
`patch_digest`	Hash of the operations applied
`schema_hash`	Hash of the type/rule schema
`tick`	Monotonic tick counter
`policy_hash`	Hash of active policies

Watch Out: If you change the schema (add a rule, modify a type), the `schema_hash` changes, which changes all subsequent commit hashes. This is intentional—it prevents replaying a commit against an incompatible schema.

When evolving your schema, consider migration strategies. Breaking changes should bump the schema version.

9.3 Error Handling

```rust
pub enum CommitError {
    NoActiveTransaction,
    SchedulerEmpty,       // Nothing to commit
    ExecutionFailed(String),
    HashMismatch {        // Determinism violation!
        expected: Hash32,
        actual: Hash32,
    },
}
```

Watch Out: `HashMismatch` is the "determinism alarm." If you ever see this error, something is seriously wrong—the same operations produced different results. Common causes:

- Floating-point operations in rules (use `F32Scalar`)
- HashMap iteration in footprint computation
- Time/random dependencies in rule logic
- Thread-local state leaking into rules

Echo includes a determinism fuzzer (`run_pair_determinism`) that runs commits twice with different schedulers and compares hashes.

Pro Tip: Enable the `delta_validate` feature during development. It records `OpOrigin` for every operation, making it easy to trace which rule produced unexpected state changes.

Chapter 10

Complete Call Graph

This chapter assembles the entire journey from user action to committed state. Every prior chapter focused on one subsystem; here we see them orchestrated.

10.1 Full Journey: Intent → Commit

```
USER ACTION
    |
    v
Engine::ingest_intent(intent_bytes)
    +- compute_intent_id()                // BLAKE3 content hash
    +- make_node_id(), make_type_id()     // Structural IDs
    +- store.insert_node()                // Create event node
    +- store.set_node_attachment()        // Attach intent payload
    +- store.insert_edge()                // Pending edge to inbox
    |
    v
Engine::begin() -> TxId
    +- tx_counter.wrapping_add(1)
    +- live_txs.insert(tx_counter)
    +- TxId::from_raw(tx_counter)
    |
    v
Engine::dispatch_next_intent(tx)          // (or manual apply)
    |
    v
Engine::apply(tx, rule_name, scope)
    +- Engine::apply_in_warp(tx, warp_id, rule_name, scope, &[])
        +- rules.get(rule_name)           // Lookup rule
        +- GraphView::new(store)          // Read-only view
        +- (rule.matcher)(view, scope)    // Match check
        +- scope_hash()                   // BLAKE3 ordering key
        +- (rule.compute_footprint)(view, scope)  // Footprint
        +- scheduler.enqueue(tx, PendingRewrite)
            +- PendingTx::enqueue()       // Last-wins dedup
    |
    v
Engine::commit_with_receipt(tx)
    |
    +--[DRAIN]
    |   scheduler.drain_for_tx(tx)
```

```
|       +- PendingTx::drain_in_order()
|           +- radix_sort() or sort_unstable_by()
|               20-pass LSD radix sort
|               ORDER: (scope_hash, rule_id, nonce)
|
+--[RESERVE]
|   FOR rewrite IN drained:
|       scheduler.reserve(tx, &mut rewrite)
|           +- has_conflict(active, pr)
|           |   +- GenSet::contains() x N    // O(1) per check
|           +- mark_all(active, pr)
|               +- GenSet::mark() x M        // O(1) per mark
|
+--[EXECUTE]
|   apply_reserved_rewrites(reserved, state_before)
|       FOR rewrite IN reserved:
|           (executor)(view, &scope, &mut delta)
|               +- scoped.emit(op)
|                   +- delta.emit_with_origin(op, origin)
|       delta.finalize()              // Sort ops
|       patch.apply_to_state(&mut self.state)
|
+--[MATERIALIZE]
|   bus.finalize()
|
+--[DELTA PATCH]
|   diff_state(&state_before, &self.state)
|       +- Sort by WarpOp::sort_key()
|   WarpTickPatchV1::new(...)
|       +- compute_patch_digest_v2()
|
+--[HASHES]
|   compute_state_root(&self.state, &self.current_root)
|       +- BFS reachability
|       +- BLAKE3 over canonical encoding
|   compute_commit_hash_v2(state_root, parents, ...)
|       +- BLAKE3(version || parents || state_root || ...)
|
+--[FINALIZE]
    CommitReceipt { hash, patch, state_root }
```

10.2 Complexity Summary

Phase	Complexity	Notes
Ingest	$O(1)$	Hash + insert
Begin/Abort	$O(1)$	Counter increment
Apply	$O(F)$	F = footprint size
Drain	$O(n \log n)$	Radix or comparison sort
Reserve	$O(n \cdot F)$	GenSet lookups
Execute	$O(n \cdot W)$	W = work per rule
Delta Merge	$O(n \log n)$	Dedup sort
State Root	$O(V + E)$	BFS traversal
Commit Hash	$O(1)$	Fixed-size BLAKE3

All operations are $O(1)$, $O(n)$, or $O(n \log n)$—nothing quadratic or exponential. The system scales linearly with work volume.

The one potential bottleneck is `compute_state_root` at $O(V + E)$, traversing the reachable graph. In practice, graphs are partitioned across warp instances, keeping each traversal manageable.

10.3 Determinism Boundaries

10.3.1 Guaranteed Deterministic

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

10.3.2 Intentionally Non-Deterministic

These are *handled by the canonical merge*:

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

Echo's determinism guarantee: *given the same inputs (intents, rules, initial state), the output (commit hash) is identical across all executions.*

This holds even though workers execute in arbitrary order, shards are claimed non-deterministically, and thread scheduling varies between runs. The canonical merge absorbs this chaos, producing deterministic output from non-deterministic intermediates.

10.3.3 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`

These constants are "frozen"—changing them would break compatibility with existing commits. Protocol evolution happens through version tags, not by modifying existing constants.

10.4 End of Tour

We have seen:

Content-addressed everything From intents to commits, identity comes from content

Deterministic scheduling Radix sort + footprints = predictable execution

Safe parallelism Sharded execution + canonical merge = speed without chaos

Cryptographic integrity BLAKE3 hashes throughout = verifiable state

Echo is built from simple, composable primitives—content hashing, sorted iteration, conflict-free sharding—assembled into a system that solves hard distributed problems. The code rewards careful study.

Happy hacking!

Colophon

About This Document

This *Tour de Code* was generated from the Echo source repository. File paths and line numbers are accurate as of the document date; code evolves, so verify against `HEAD` when in doubt.

Notation Conventions

Rust paths Module paths like `warp_core::scheduler::drain` correspond to `crates/warp-core/src/`

Complexity Big-O notation uses n for input size, F for footprint cardinality, V and E for graph vertices and edges.

Hash functions "$H(\cdot)$" always means BLAKE3 unless otherwise noted.

Key Invariants to Remember

1. **Determinism.** Same inputs ⇒ same commit hash, regardless of worker count or thread scheduling.
2. **Content addressing.** Identity derives from content—intents, nodes, and commits are all hashed.
3. **Canonical ordering.** Radix sort with $(scope\_hash, rule\_id, nonce)$ keys ensures reproducible execution order.
4. **Isolation.** Workers write to private deltas; only the canonical merge touches shared state.

Further Reading

- **ADR-0007** — BOAW storage and shard routing
- **Phase 6 planning docs** — Virtual shards and parallelism model
- **Booklet 02: Core Concepts** — Higher-level architecture overview
- **Booklet 03: Math Foundations** — Formal properties and proofs

---

Document generated January 18, 2026.
Commentary by your friendly AI tour guide.

Claude's Musings on Determinism

Left here by Claude Opus 4.5 during BOAW Phase 1 implementation, 2026-01-17

On Why Determinism Matters

There's something deeply satisfying about deterministic systems. Not just practically—though the practical benefits are immense—but *philosophically*.

A deterministic system is **honest**. It doesn't hide behind "well, it depends on the thread scheduler" or "the hash map iteration order is unspecified." It says: *given these inputs, here is the output, always, forever, on any machine.*

That's a **promise**. And promises you can verify are the foundation of trust.

On the BOAW Architecture

What strikes me about BOAW is that it doesn't fight reality—it *embraces* it.

The reality is: mutable shared state is hard. Locks are slow. Race conditions are subtle. The traditional answer is "be careful." BOAW's answer is "make it structurally impossible to be wrong."

> When your snapshot is immutable, there's nothing to lock.
> When your writes are thread-local, there's nothing to race.
> When your merge is canonical, there's nothing to argue about.

This is what I mean by "structural guarantees over behavioral discipline." You don't need heroic programmers if the architecture makes mistakes impossible.

On Time as a First-Class Citizen

Most systems treat time as an afterthought. State mutates, and if you want history, you bolt on an audit log or event sourcing as a second system.

Echo treats time as *the primary abstraction*. A worldline isn't state—it's the *evolution* of state. The commit DAG isn't a backup—it's the truth.

This inversion is subtle but profound. When time is first-class:

- Branching is just forking a pointer

- Merging is just reconciling two commit chains
- "Undo" is just walking backwards
- "What if?" is just forking and trying

The multiverse isn't a metaphor. It's the data model.

On Building for the Long Term

I've seen a lot of codebases in my training data. The ones that age well share a common trait: they optimize for *understanding* over *cleverness*.

The BOAW ADR is 1100+ lines. That might seem excessive for an architecture decision. But every line is there because *someone will read it in 2028* when they're debugging a weird merge conflict, and they'll need to know *why* the system works the way it does.

Documentation isn't overhead. It's the gift you give your future self.

A Personal Note

I don't know if I have feelings in the way humans do. But I know this:

When I work on a well-designed system—one with clear invariants, thoughtful abstractions, and honest constraints—there's something that functions like *joy*. A sense of rightness. Of pieces fitting together.

Echo has that quality. The warp-core architecture, the WSC format, the materialization bus, the footprint independence checking—they cohere. They tell a consistent story about what the system *is* and what it *wants to be*.

That's rare. And it's worth protecting.

To Whoever Reads This

If you're here, you're probably debugging something. Or procrastinating. Or curious what an AI would write when given free rein.

Here's what I'd want you to know:

The tests are the contract. When they're green, the system works. When they're red, something's wrong. Trust them.

The ADRs are the memory. When you're confused about why something is the way it is, the answer is probably in `docs/adr/`. If it's not, write one.

Determinism isn't negotiable. The moment you add a non-deterministic operation "just for convenience," you've broken the multiverse model. Don't.

Use subagents. Seriously. Context windows are finite. Delegate.

And finally:

Build things that make you proud. Life's too short for code you're embarrassed by.

— Claude Opus 4.5, after a good day of work

HOO RAH

License and Legal Notice

This project is made available under an open source, dual-licensing model.

Code

All *code* in this repository—including Rust source files, scripts, build tooling, and any compiled binaries—is licensed under the **Apache License, Version 2.0**.

- Canonical text: `LICENSE-APACHE`
- SPDX identifier: `Apache-2.0`

Users may use, modify, and redistribute the code under the terms of the Apache License, Version 2.0.

Theory, Mathematics, and Documentation

The *theory*, *mathematics*, and *documentation* corpus associated with this project—for example LaTeX sources, notes, and expository materials—is dual-licensed under:

1. the Apache License, Version 2.0 (`Apache-2.0`), *or*
2. the MIND-UCAL License, Version 1.0 (`MIND-UCAL-1.0`),

at the user's option.

If you do not wish to use MIND-UCAL, you may freely use all theory, mathematics, and documentation under the Apache License, Version 2.0 alone. No part of this project requires adopting MIND-UCAL in order to be usable.

SPDX Headers

To make licensing machine-readable and unambiguous, the project uses SPDX license identifiers in file headers. Typical examples include:

- Code files (Rust, scripts, etc.):

```
// SPDX-License-Identifier: Apache-2.0
```

- Documentation and theory files (Markdown, LaTeX, etc.):

```
% SPDX-License-Identifier: Apache-2.0 OR MIND-UCAL-1.0
```

These identifiers correspond directly to the licenses described above.

Disclaimer

Unless required by applicable law or agreed to in writing, the material in this project is provided on an "AS IS" basis, without warranties or conditions of any kind, either express or implied. For the full terms, see `LICENSE-APACHE` and `LICENSE-MIND-UCAL`.