Echo: Tour de Code

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

This document 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.

Table of Contents

1. Intent Ingestion
2. Transaction Lifecycle
3. Rule Matching
4. Scheduler: Drain & Reserve
5. BOAW Parallel Execution
6. Delta Merge & State Finalization
7. Hash Computation
8. Commit Orchestration
9. Complete Call Graph

---

1. Intent Ingestion

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

1.1 Function Signature

```
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

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]

 [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 })
```

1.3 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)

---

2. Transaction Lifecycle

2.1 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);  // Line 713
    if self.tx_counter == 0 {
        self.tx_counter = 1;                            // Line 715: Zero is reserved
    }
    self.live_txs.insert(self.tx_counter);              // Line 717
    TxId::from_raw(self.tx_counter)                     // Line 718
}
```

Call Trace:

```
Engine::begin()

  self.tx_counter.wrapping_add(1)
   Rust std: u64::wrapping_add
   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>
   Registers transaction as active
  TxId::from_raw(self.tx_counter)
    FILE: crates/warp-core/src/tx.rs:34
    CODE: pub const fn from_raw(value: u64) -> Self { Self(value) }
    TYPE: #[repr(transparent)] struct TxId(u64)
```

State Changes: - `tx_counter`: N → N+1 (or 1 if wrapped) - `live_txs`: Insert new counter value

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();
}
```

---

3. Rule Matching

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

3.1 Function Signature

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

3.2 Complete Call Trace

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

  Engine::apply_in_warp(tx, self.current_root.warp_id, rule_name, scope, &[])
    FILE: crates/warp-core/src/engine_impl.rs:754-806

      [1] TRANSACTION VALIDATION
          CODE: if tx.value() == 0 || !self.live_txs.contains(&tx.value())
          ERROR: EngineError::UnknownTx
```

```
[2] RULE LOOKUP
    self.rules.get(rule_name) → Option<&RewriteRule>
    TYPE: HashMap<&'static str, RewriteRule>
    ERROR: EngineError::UnknownRule(rule_name.to_owned())

[3] STORE LOOKUP
    self.state.store(&warp_id) → Option<&GraphStore>
    ERROR: EngineError::UnknownWarp(warp_id)

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

[5] CALL MATCHER
    (rule.matcher)(view, scope) → bool
    TYPE: MatchFn = for<'a> fn(GraphView<'a>, &NodeId) -> bool
    FILE: crates/warp-core/src/rule.rs:16-24
    IF false: return Ok(ApplyResult::NoMatch)

[6] CREATE SCOPE KEY
    let scope_key = NodeKey { warp_id, local_id: *scope }

[7] COMPUTE SCOPE HASH
    scope_hash(&rule.id, &scope_key) → Hash
    FILE: crates/warp-core/src/engine_impl.rs:1712-1718
    CODE:
      let mut hasher = Hasher::new();
      hasher.update(rule_id);              // 32 bytes
      hasher.update(scope.warp_id.as_bytes());  // 32 bytes
      hasher.update(scope.local_id.as_bytes()); // 32 bytes
      hasher.finalize().into()

[8] COMPUTE FOOTPRINT
    (rule.compute_footprint)(view, scope) → Footprint
    TYPE: FootprintFn = for<'a> fn(GraphView<'a>, &NodeId) -> Footprint
    FILE: crates/warp-core/src/rule.rs:38-46
    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,        // O(1) prefilter
        }

  [9] AUGMENT FOOTPRINT WITH DESCENT STACK
      for key in descent_stack:
        footprint.a_read.insert(*key)
      FILE: crates/warp-core/src/footprint.rs:104-107
      PURPOSE: Stage B1 law - READs of all descent chain slots

  [10] COMPACT RULE ID LOOKUP
       self.compact_rule_ids.get(&rule.id) → Option<&CompactRuleId>
       TYPE: HashMap<Hash, CompactRuleId>
       ERROR: EngineError::InternalCorruption

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

         DeterministicScheduler::enqueue(tx, rewrite)
           FILE: crates/warp-core/src/scheduler.rs:654-659

             RadixScheduler::enqueue(tx, rewrite)
               FILE: crates/warp-core/src/scheduler.rs:102-105
               CODE:
                 let txq = self.pending.entry(tx).or_default();
                 txq.enqueue(rewrite.scope_hash, rewrite.compact_rule.0, rewrite);

                 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
                     index.get(&key) → Some(&i)
                     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)
```

3.3 PendingRewrite Structure

File: `crates/warp-core/src/scheduler.rs:68-82`

```
pub(crate) struct PendingRewrite {
    pub rule_id: Hash,                // 32-byte rule identifier
    pub compact_rule: CompactRuleId, // u32 hot-path handle
    pub scope_hash: Hash,             // 32-byte ordering key
    pub scope: NodeKey,               // { warp_id, local_id }
    pub footprint: Footprint,         // Read/write declaration
    pub phase: RewritePhase,          // State machine: Matched → Reserved → ...
}
```

---

4. Scheduler: Drain & Reserve

4.1 Drain Phase (Radix Sort)

Entry Point: `RadixScheduler::drain_for_tx()` **File:** `crates/warp-core/src/scheduler.rs:109-1`

```
pub(crate) fn drain_for_tx(&mut self, tx: TxId) -> Vec<PendingRewrite> {
    self.pending
        .remove(&tx)
        .map_or_else(Vec::new, |mut txq| txq.drain_in_order())
}
```

Complete Call Trace:

```
RadixScheduler::drain_for_tx(tx)

  self.pending.remove(&tx) → Option<PendingTx<PendingRewrite>>

  PendingTx::drain_in_order()
    FILE: crates/warp-core/src/scheduler.rs:416-446

      DECISION: n <= 1024 (SMALL_SORT_THRESHOLD)?
          YES: sort_unstable_by(cmp_thin)
                Rust std comparison sort

          NO: radix_sort()
               FILE: crates/warp-core/src/scheduler.rs:360-413

      radix_sort()

          Initialize scratch buffer: self.scratch.resize(n, default)

          Lazy allocate histogram: self.counts16 = vec![0u32; 65536]

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

```
              SELECT src/dst buffers (ping-pong)
                flip = false: src=thin, dst=scratch
                flip = true:  src=scratch, dst=thin

              PHASE 1: COUNT BUCKETS
                FOR r IN src:
                  b = bucket16(r, pass)
                  counts[b] += 1

              PHASE 2: PREFIX SUMS
                sum = 0
                FOR c IN counts:
                  t = *c
                  *c = sum
                  sum += t

              PHASE 3: STABLE SCATTER
                FOR r IN src:
                  b = bucket16(r, pass)
                  dst[counts[b]] = r
                  counts[b] += 1

              flip = !flip

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 5:  u16_be_from_pair32(scope, 14)    // Scope bytes [28:30]
...
Pass 19: u16_be_from_pair32(scope, 0)     // Scope bytes [0:2] (MSD)

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

4.2 Reserve Phase (Independence Check)

Entry Point: `RadixScheduler::reserve()` **File:** `crates/warp-core/src/scheduler.rs:134-143`

```rust
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)
}
```

Complete Call Trace:

```
RadixScheduler::reserve(tx, pr)

  self.active.entry(tx).or_insert_with(ActiveFootprints::new)
   TYPE: HashMap<TxId, ActiveFootprints>
   ActiveFootprints contains 7 GenSets:
     - nodes_written: GenSet<NodeKey>
     - nodes_read: GenSet<NodeKey>
     - edges_written: GenSet<EdgeKey>
     - edges_read: GenSet<EdgeKey>
     - attachments_written: GenSet<AttachmentKey>
     - attachments_read: GenSet<AttachmentKey>
     - ports: GenSet<PortKey>

  has_conflict(active, pr) → bool
   FILE: crates/warp-core/src/scheduler.rs:157-236

     FOR node IN pr.footprint.n_write:
         IF active.nodes_written.contains(node): return true  // W-W conflict
         IF active.nodes_read.contains(node): return true     // W-R conflict

     FOR node IN pr.footprint.n_read:
         IF active.nodes_written.contains(node): return true  // R-W conflict
         (R-R is allowed)

     FOR edge IN pr.footprint.e_write:
         IF active.edges_written.contains(edge): return true
         IF active.edges_read.contains(edge): return true

     FOR edge IN pr.footprint.e_read:
         IF active.edges_written.contains(edge): return true

     FOR key IN pr.footprint.a_write:
         IF active.attachments_written.contains(key): return true
         IF active.attachments_read.contains(key): return true

     FOR key IN pr.footprint.a_read:
         IF active.attachments_written.contains(key): return true

     FOR port IN pr.footprint.b_in   pr.footprint.b_out:
         IF active.ports.contains(port): return true
  IF conflict:
     on_conflict(pr)
       FILE: crates/warp-core/src/scheduler.rs:145-149
       pr.phase = RewritePhase::Aborted
       return false

  mark_all(active, pr)
   FILE: crates/warp-core/src/scheduler.rs:238-278

     FOR node IN pr.footprint.n_write:
         active.nodes_written.mark(NodeKey { warp_id, local_id: node })

     FOR node IN pr.footprint.n_read:
         active.nodes_read.mark(NodeKey { ... })

     FOR edge IN pr.footprint.e_write:
         active.edges_written.mark(EdgeKey { ... })

     FOR edge IN pr.footprint.e_read:
         active.edges_read.mark(EdgeKey { ... })

     FOR key IN pr.footprint.a_write:
         active.attachments_written.mark(key)

     FOR key IN pr.footprint.a_read:
         active.attachments_read.mark(key)

     FOR port IN pr.footprint.b_in   pr.footprint.b_out:
         active.ports.mark(port)

  on_reserved(pr)
    FILE: crates/warp-core/src/scheduler.rs:151-155
    pr.phase = RewritePhase::Reserved
    return true
```

4.3 GenSet: O(1) Conflict Detection

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

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

impl<K: Hash + Eq + Copy> GenSet<K> {
    #[inline]
    pub fn contains(&self, key: K) -> bool {
        matches!(self.seen.get(&key), Some(&g) if g == self.gen)
    }

    #[inline]
    pub fn mark(&mut self, key: K) {
        self.seen.insert(key, self.gen);
    }
}
```

Key Insight: No clearing needed between transactions. Increment `gen` → all old entries become stale.

---

5. BOAW Parallel Execution

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

5.1 Entry Point

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

    #[cfg(feature = "parallel-stride-fallback")]
    if std::env::var("ECHO_PARALLEL_STRIDE").is_ok() {
        return execute_parallel_stride(view, items, capped_workers);
    }

    execute_parallel_sharded(view, items, capped_workers)  // DEFAULT
}
```

5.2 Complete Call Trace

```
execute_parallel(view, items, workers)

  execute_parallel_sharded(view, items, capped_workers)
    FILE: crates/warp-core/src/boaw/exec.rs:101-152

      IF items.is_empty():
          return (0..workers).map(|_| TickDelta::new()).collect()

      partition_into_shards(items.to_vec()) → Vec<VirtualShard>
        FILE: crates/warp-core/src/boaw/shard.rs:109-120
      Create 256 empty VirtualShard structures

      FOR item IN items:

          shard_of(&item.scope) → usize
           FILE: crates/warp-core/src/boaw/shard.rs:82-92
           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)

  let next_shard = AtomicUsize::new(0)

  std::thread::scope(|s| { ... })
    FILE: Rust std (scoped threads)

      FOR _ IN 0..workers:

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

              let mut delta = TickDelta::new()
               FILE: crates/warp-core/src/tick_delta.rs:44-52
               CREATES: { ops: Vec::new(), origins: Vec::new() }

              LOOP:  // Work-stealing loop

                  shard_id = next_shard.fetch_add(1, Ordering::Relaxed)
                   ATOMIC: Returns old value, increments counter
                   ORDERING: Relaxed (no synchronization cost)

                  IF shard_id >= 256: break

                  FOR item IN &shards[shard_id].items:

                      let mut scoped = delta.scoped(item.origin)
                       FILE: crates/warp-core/src/tick_delta.rs:140-142
                       CREATES: ScopedDelta { inner: &mut delta, origin, next_op_ix: 0 }

                      (item.exec)(view, &item.scope, scoped.inner_mut())

                          INSIDE EXECUTOR:
                            scoped.emit(op)
```

```
                                FILE: crates/warp-core/src/tick_delta.rs:234-239
                                CODE:
                                  origin.op_ix = self.next_op_ix;
                                  self.next_op_ix += 1;
                                  self.inner.emit_with_origin(op, origin);

                                  TickDelta::emit_with_origin(op, origin)
                                    FILE: crates/warp-core/src/tick_delta.rs:69-75
                                    CODE:
                                      self.ops.push(op);
                                      self.origins.push(origin);  // if delta_validate

          COLLECT THREADS:
            handles.into_iter().map(|h| h.join()).collect()
            RETURNS: Vec<TickDelta> (one per worker)
```

5.3 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 }
}
```

5.4 Thread Safety

Type	Safety	Reason
`GraphView`	`Sync + Send + Clone`	Read-only snapshot
`ExecItem`	`Sync + Send + Copy`	Function pointer + primitives
`TickDelta`	Per-worker exclusive	No shared mutation
`AtomicUsize`	Lock-free	`fetch_add` with `Relaxed` ordering

---

6. Delta Merge & State Finalization

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
     let mut flat: Vec<(WarpOpKey, OpOrigin, WarpOp)> = Vec::new();
     FOR d IN deltas:
       let (ops, origins) = d.into_parts_unsorted();
       FOR (op, origin) IN ops.zip(origins):
         flat.push((op.sort_key(), origin, op));

[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
      let mut out = Vec::new();
      let mut i = 0;
      WHILE i < flat.len():

          GROUP by WarpOpKey
           key = flat[i].0
           start = i
           WHILE i < flat.len() && flat[i].0 == key: i++

          CHECK if all ops identical
           first = &flat[start].2
           all_same = flat[start+1..i].iter().all(|(_, _, op)| op == first)

          IF all_same:
            out.push(first.clone())       // Accept one copy
           ELSE:
            writers = flat[start..i].iter().map(|(_, o, _)| *o).collect()
            return Err(MergeConflict { writers })  // CONFLICT!

      return Ok(out)
```

6.2 WarpOp Sort Key

File: crates/warp-core/src/tick_patch.rs:207-287

```rust
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, ... },  // Delete before upsert
        Self::DeleteNode { .. }        => WarpOpKey { kind: 5, ... },
        Self::UpsertNode { .. }        => WarpOpKey { kind: 6, ... },
        Self::UpsertEdge { .. }        => WarpOpKey { kind: 7, ... },
        Self::SetAttachment { .. }     => WarpOpKey { kind: 8, ... },  // Last
    }
}
```

Canonical Order: 1. OpenPortal (creates child instances) 2. UpsertWarpInstance 3. DeleteWarpInstance 4. DeleteEdge (delete before upsert) 5. DeleteNode (delete before upsert) 6. UpsertNode 7. UpsertEdge 8. SetAttachment (after skeleton exists)

6.3 State Mutation Methods

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

```
GraphStore::insert_node(id, record)
  LINE: 175-177
  CODE: self.nodes.insert(id, record)

GraphStore::upsert_edge_record(from, edge)
  LINE: 196-261
  UPDATES:
    - self.edge_index.insert(edge_id, from)
    - self.edge_to_index.insert(edge_id, to)
    - Remove old edge from previous bucket if exists
    - self.edges_from.entry(from).or_default().push(edge)
    - self.edges_to.entry(to).or_default().push(edge_id)

GraphStore::delete_node_cascade(node)
  LINE: 277-354
  CASCADES:
    - Remove from self.nodes
    - Remove node attachment
    - Remove ALL outbound edges (and their attachments)
    - Remove ALL inbound edges (and their attachments)
    - Maintain all 4 index maps consistently

GraphStore::delete_edge_exact(from, edge_id)
  LINE: 360-412
  VALIDATES: edge is in correct "from" bucket
  REMOVES:
    - From edges_from bucket
    - From edge_index
    - From edge_to_index
    - From edges_to bucket
    - Edge attachment

GraphStore::set_node_attachment(id, value)
  LINE: 125-134
  CODE:
    None → self.node_attachments.remove(&id)
    Some(v) → self.node_attachments.insert(id, v)

GraphStore::set_edge_attachment(id, value)
  LINE: 163-172
  Same pattern as node attachments
```

---

7. Hash Computation

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

        Initialize:
         reachable_nodes: BTreeSet<NodeKey> = { root }
         reachable_warps: BTreeSet<WarpId> = { root.warp_id }
         queue: VecDeque<NodeKey> = [ root ]

        WHILE let Some(current) = queue.pop_front():

            store = state.store(&current.warp_id)

            FOR edge IN store.edges_from(&current.local_id):
                to = NodeKey { warp_id: current.warp_id, local_id: edge.to }
                IF reachable_nodes.insert(to): queue.push_back(to)

                IF edge has Descend(child_warp) attachment:
                    enqueue_descend(state, child_warp, ...)
                      Adds child instance root to queue

            IF current node has Descend(child_warp) attachment:
                enqueue_descend(state, child_warp, ...)

 [2] HASHING PHASE

        let mut hasher = Hasher::new()  // BLAKE3

        HASH ROOT BINDING:
         hasher.update(&root.warp_id.0)     // 32 bytes
         hasher.update(&root.local_id.0)    // 32 bytes

       FOR warp_id IN reachable_warps:  // BTreeSet = sorted order

           HASH INSTANCE HEADER:
            hasher.update(&instance.warp_id.0)      // 32 bytes
            hasher.update(&instance.root_node.0)   // 32 bytes
            hash_attachment_key_opt(&mut hasher, instance.parent.as_ref())

           FOR (node_id, node) IN store.nodes:  // BTreeMap = sorted
            IF reachable_nodes.contains(&NodeKey { warp_id, local_id: node_id }):
              hasher.update(&node_id.0)          // 32 bytes
              hasher.update(&node.ty.0)          // 32 bytes
              hash_attachment_value_opt(&mut hasher, store.node_attachment(node_id))

           FOR (from, edges) IN store.edges_from:  // BTreeMap = sorted
             IF from is reachable:
               sorted_edges = edges.filter(reachable).sort_by(|a,b| a.id.cmp(b.id))
               hasher.update(&from.0)                       // 32 bytes
               hasher.update(&(sorted_edges.len() as u64).to_le_bytes())  // 8 bytes
               FOR edge IN sorted_edges:
                 hasher.update(&edge.id.0)                  // 32 bytes
                 hasher.update(&edge.ty.0)                  // 32 bytes
                 hasher.update(&edge.to.0)                  // 32 bytes
                 hash_attachment_value_opt(&mut hasher, store.edge_attachment(&edge.id

  hasher.finalize().into()  // → [u8; 32]
```

7.2 Commit Hash v2

Entry Point: `compute_commit_hash_v2()` **File:** `crates/warp-core/src/snapshot.rs:244-263`

```rust
pub(crate) 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 (2 bytes)
    h.update(&(parents.len() as u64).to_le_bytes());  // Parent count (8 bytes)
    for p in parents {
        h.update(p);                            // Each parent (32 bytes)
    }
    h.update(state_root);                       // Graph hash (32 bytes)
    h.update(patch_digest);                     // Ops hash (32 bytes)
    h.update(&policy_id.to_le_bytes());         // Policy (4 bytes)
    h.finalize().into()
}
```

Byte Layout:

```
Offset    Size    Field
0         2       version_tag (0x02 0x00)
2         8       parent_count (u64 LE)
10        32*N    parents[] (N parent hashes)
10+32N    32      state_root
42+32N    32      patch_digest
74+32N    4       policy_id (u32 LE)

TOTAL: 78 + 32*N bytes → BLAKE3 → 32-byte hash
```

7.3 Patch Digest

Entry Point: `compute_patch_digest_v2()` **File:** `crates/warp-core/src/tick_patch.rs:755-774`

```
fn compute_patch_digest_v2(
    policy_id: u32,
    rule_pack_id: &ContentHash,
    commit_status: TickCommitStatus,
    in_slots: &[SlotId],
    out_slots: &[SlotId],
    ops: &[WarpOp],
) -> ContentHash {
    let mut h = Hasher::new();
    h.update(&2u16.to_le_bytes());          // Format version
    h.update(&policy_id.to_le_bytes());     // 4 bytes
    h.update(rule_pack_id);                 // 32 bytes
    h.update(&[commit_status.code()]);      // 1 byte
    encode_slots(&mut h, in_slots);
    encode_slots(&mut h, out_slots);
    encode_ops(&mut h, ops);
    h.finalize().into()
}
```

---

8. Commit Orchestration

Entry Point: `Engine::commit_with_receipt()` **File:** `crates/warp-core/src/engine_impl.rs:837-954`

8.1 Complete Call Trace

```
Engine::commit_with_receipt(tx) → Result<(Snapshot, TickReceipt, WarpTickPatchV1), Eng

 [1] VALIDATE TRANSACTION
     IF tx.value() == 0 || !self.live_txs.contains(&tx.value()):
       return Err(EngineError::UnknownTx)

 [2] DRAIN CANDIDATES
     policy_id = self.policy_id                         // Line 844
     rule_pack_id = self.compute_rule_pack_id()         // Line 845

       compute_rule_pack_id()
        FILE: engine_impl.rs:1675-1688
        CODE:
          ids = self.rules.values().map(|r| r.id).collect()
          ids.sort_unstable(); ids.dedup()
          hasher.update(&1u16.to_le_bytes())  // version
          hasher.update(&(ids.len() as u64).to_le_bytes())
          FOR id IN ids: hasher.update(&id)
          hasher.finalize().into()

     drained = self.scheduler.drain_for_tx(tx)          // Line 847
     plan_digest = compute_plan_digest(&drained)        // Line 848

 [3] RESERVE (INDEPENDENCE CHECK)
     ReserveOutcome { receipt, reserved, in_slots, out_slots }
       = self.reserve_for_receipt(tx, drained)?         // Line 850-855

       reserve_for_receipt(tx, drained)
         FILE: engine_impl.rs:970-1042

         FOR rewrite IN drained (canonical order):

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

             IF !accepted:
                 blockers = find_blocking_rewrites(reserved, &rewrite)

             receipt_entries.push(TickReceiptEntry { ... })

             IF accepted:
                 reserved.push(rewrite)
                 extend_slots_from_footprint(&mut in_slots, &mut out_slots, ...)

         return ReserveOutcome { receipt, reserved, in_slots, out_slots }
```

```
    rewrites_digest = compute_rewrites_digest(&reserved_rewrites)  // Line 858

 [4] EXECUTE (PHASE 5 BOAW)
    state_before = self.state.clone()                   // Line 862
    delta_ops = self.apply_reserved_rewrites(reserved, &state_before)?

      apply_reserved_rewrites(rewrites, state_before)
        FILE: engine_impl.rs:1044-1105

          let mut delta = TickDelta::new()

          FOR rewrite IN rewrites:
              executor = self.rule_by_compact(rewrite.compact_rule).executor
              view = GraphView::new(self.state.store(&rewrite.scope.warp_id))
              (executor)(view, &rewrite.scope.local_id, &mut delta)

          let ops = delta.finalize()  // Canonical sort

          patch = WarpTickPatchV1::new(policy_id, rule_pack_id, ..., ops)
           patch.apply_to_state(&mut self.state)?

          [delta_validate]: assert_delta_matches_diff(&ops, &diff_ops)

 [5] MATERIALIZE
    mat_report = self.bus.finalize()                    // Line 884
    self.last_materialization = mat_report.channels
    self.last_materialization_errors = mat_report.errors

 [6] COMPUTE DELTA PATCH
    ops = diff_state(&state_before, &self.state)        // Line 889

      diff_state(before, after)
        FILE: tick_patch.rs:979-1069
        - Canonicalize portal authoring (OpenPortal)
        - Diff instances (delete/upsert)
        - Diff nodes, edges, attachments
        - Sort by WarpOp::sort_key()

    patch = WarpTickPatchV1::new(policy_id, rule_pack_id, ..., ops)
    patch_digest = patch.digest()                       // Line 898

 [7] COMPUTE STATE ROOT
    state_root = compute_state_root(&self.state, &self.current_root)  // Line 900

 [8] GET PARENTS
```

```
    parents = self.last_snapshot.as_ref().map(|s| vec![s.hash]).unwrap_or_default()

[9] COMPUTE DECISION DIGEST
    decision_digest = receipt.digest()                 // Line 929

[10] COMPUTE COMMIT HASH
     hash = compute_commit_hash_v2(&state_root, &parents, &patch_digest, policy_id)

[11] BUILD SNAPSHOT
     snapshot = Snapshot {
       root: self.current_root,
       hash,                    // commit_id v2
       parents,
       plan_digest,             // Diagnostic
       decision_digest,         // Diagnostic
       rewrites_digest,         // Diagnostic
       patch_digest,            // COMMITTED
       policy_id,               // COMMITTED
       tx,
     }

[12] RECORD TO HISTORY
     self.last_snapshot = Some(snapshot.clone())       // Line 947
     self.tick_history.push((snapshot, receipt, patch))  // Line 948-949
     self.live_txs.remove(&tx.value())                 // Line 951
     self.scheduler.finalize_tx(tx)                    // Line 952

[13] RETURN
      Ok((snapshot, receipt, patch))
```

8.2 Commit Hash Inputs

Input	Committed?	Purpose
`state_root`		What the graph looks like
`patch_digest`		How we got here (ops)
`parents`		Chain continuity
`policy_id`		Aion policy version
`plan_digest`		Diagnostic only
`decision_digest`		Diagnostic only
`rewrites_digest`		Diagnostic only

9. Complete Call Graph

9.1 Full Journey: Intent → Commit

```
USER ACTION


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


Engine::begin() → TxId
      tx_counter.wrapping_add(1)
      live_txs.insert(tx_counter)
      TxId::from_raw(tx_counter)


Engine::dispatch_next_intent(tx)              // (or manual apply)


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


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() × N    // O(1) per check
                mark_all(active, pr)
                    GenSet::mark() × 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, patch_digest, policy_id)
             BLAKE3(version || parents || state_root || patch_digest || policy_id)

     [SNAPSHOT]
       Snapshot { root, hash, parents, digests..., policy_id, tx }

     [RECORD]
        tick_history.push((snapshot, receipt, patch))
        live_txs.remove(&tx.value())
        scheduler.finalize_tx(tx)


RETURN: (Snapshot, TickReceipt, WarpTickPatchV1)
```

9.2 File Index

Component	Primary File	Key Lines
Intent Ingestion	`engine_impl.rs`	1216-1281
Identity Hashing	`ident.rs`	85-109
Transaction Begin	`engine_impl.rs`	711-719
Rule Apply	`engine_impl.rs`	730-806
Footprint	`footprint.rs`	131-152
Scheduler Enqueue	`scheduler.rs`	102-105, 331-355
Radix Sort	`scheduler.rs`	360-413, 481-498
Reserve/Conflict	`scheduler.rs`	134-278
GenSet	`scheduler.rs`	509-535
BOAW Execute	`boaw/exec.rs`	61-152
Shard Routing	`boaw/shard.rs`	82-120
Delta Merge	`boaw/merge.rs`	36-75
TickDelta	`tick_delta.rs`	38-172
WarpOp Sort Key	`tick_patch.rs`	207-287
State Mutations	`graph.rs`	175-412
Patch Apply	`tick_patch.rs`	434-561
Diff State	`tick_patch.rs`	979-1069
State Root Hash	`snapshot.rs`	88-209
Commit Hash v2	`snapshot.rs`	244-263
Patch Digest	`tick_patch.rs`	755-774
Commit Orchestrator	`engine_impl.rs`	837-954

Appendix A: Complexity Summary

Operation	Complexity	Notes
`ingest_intent`	O(1)	Fixed structural insertions
`begin`	O(1)	Counter increment + set insert
`apply`	O(m)	m = footprint size
`drain_for_tx` (radix)	O(n)	n = candidates, 20 passes
`reserve` per rewrite	O(m)	m = footprint size, O(1) per check
`execute_parallel`	O(n/w)	n = items, w = workers
`merge_deltas`	O(k log k)	k = total ops (sort + dedup)
`compute_state_root`	O(V + E)	V = nodes, E = edges
`compute_commit_hash_v2`	O(P)	P = parents

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)

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`

Document generated 2026-01-18. File paths and line numbers accurate as of this date.