# What Makes Echo Tick?

A comprehensive technical guide to the Echo deterministic graph-rewrite engine.

**Target Audience**: Developers who want to understand Echo's internals in exhaustive detail.

**Reading Time**: ~45 minutes for complete understanding.

---

## Table of Contents



---

## 1. Philosophy: Why Echo Exists

### 1.1 The Problem

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

- **Replay is hard**: You can't just "rewind" because state changes are scattered and untracked.

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

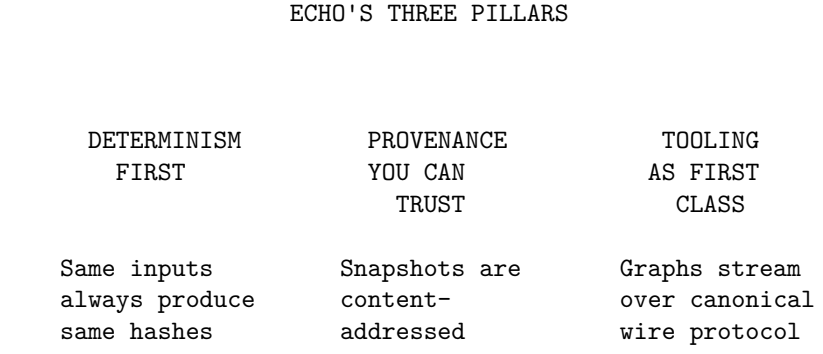
### 1.2 Echo's Answer

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

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

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

### 1.3 Core Design Principles

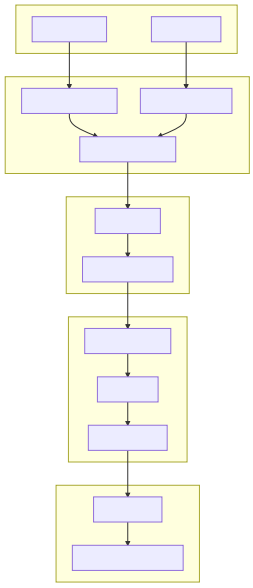



---

## 2. The Big Picture: Architecture Overview

### 2.1 System Layers

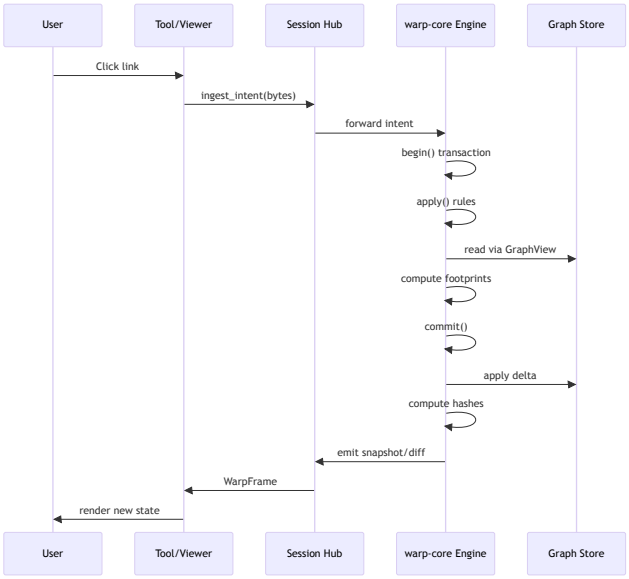
Echo is organized into distinct layers, each with a specific responsibility:



## 2.2 Crate Map

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

### 2.3 Data Flow Overview

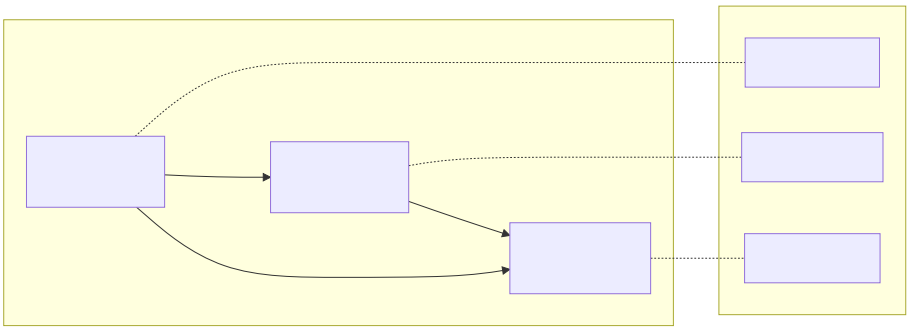



---

## 3. Core Concepts: The WARP Graph

### 3.1 What is a WARP Graph?

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

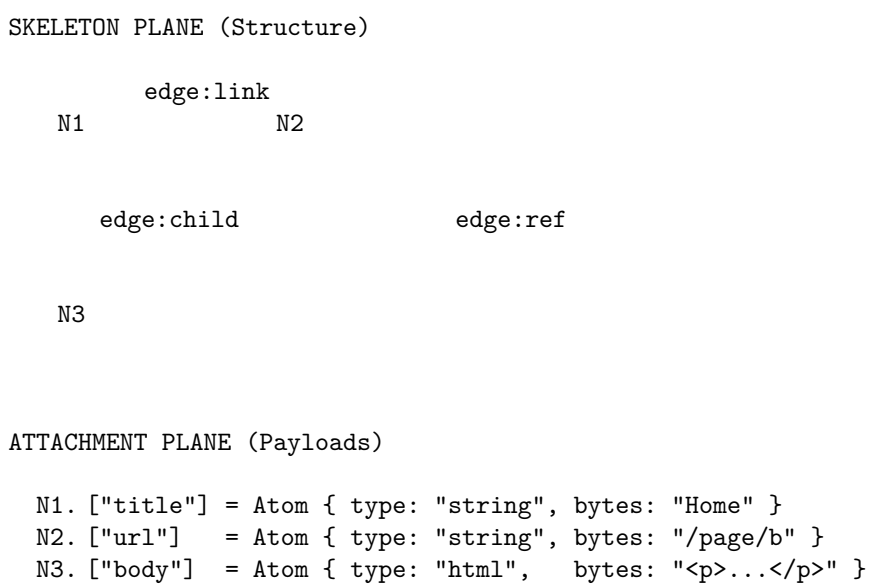


### 3.2 Two-Plane Architecture

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

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

**Why separate them?**




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

## 3.3 Node and Edge Identity

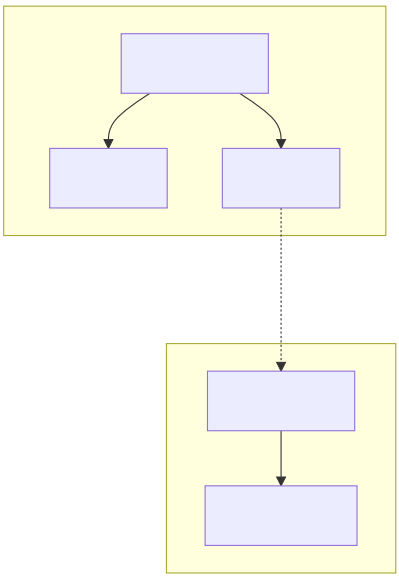
Every node and edge has a **32-byte identifier**:

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

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

## 3.4 WarpInstances: Graphs Within Graphs

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



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

---

# 4. The Engine: Heart of Echo

## 4.1 The Engine Struct

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

```
pub struct Engine {
    state: WarpState,                          // Multi-instance graph state
    rules: HashMap<RuleId, RewriteRule>,       // Registered rewrite rules
    scheduler: DeterministicScheduler,         // Deterministic ordering
    bus: MaterializationBus,                   // Output channels
    history: Vec<(Snapshot, TickReceipt, WarpTickPatchV1)>,
    tx_counter: u64,                           // Transaction counter
    live_txs: BTreeSet<TxId>,                  // Active transactions
    // ... more fields
}
```

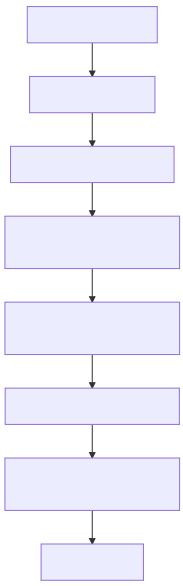
## 4.2 Construction

The engine is built via the `EngineBuilder`:

```
let engine = EngineBuilder::new(store, root_node_id)
    .with_policy_id(1)
```

```
    .with_telemetry(telemetry)
    .build();
```

**What happens during construction:**



## 4.3 Rewrite Rules

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

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

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

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

## 4.4 GraphView: Read-Only Access

The `GraphView` enforces BOAW's immutability contract:

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

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

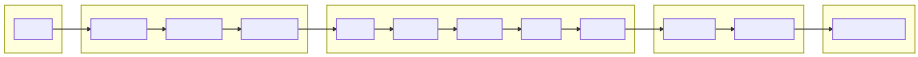
**No `DerefMut`, no `AsRef<GraphStore>`, no interior mutability.** This is enforced at the type level.

---

# 5. The Tick Pipeline: Where Everything Happens

## 5.1 Overview

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



## 5.2 Phase 1: Begin Transaction

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

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

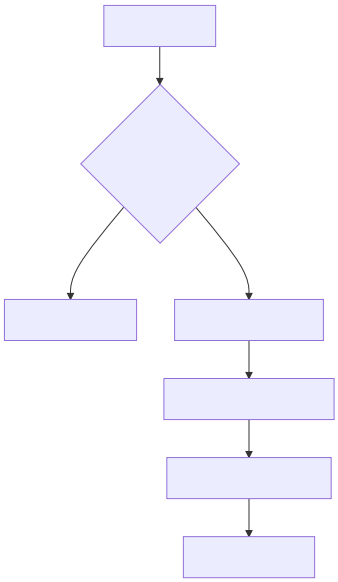
```
engine.begin()

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

## 5.3 Phase 2: Apply Rules

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

**What happens:**



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

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

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

## 5.4 Phase 3: Commit (The Heart of Determinism)

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

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

### 5.4.1 Drain

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

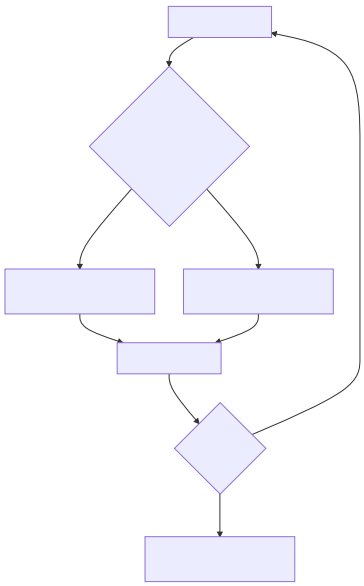
```
// RadixScheduler uses O(n) LSD radix sort
// 20 passes: 2 nonce + 2 rule_id + 16 scope_hash (16-bit digits)
let rewrites = scheduler.drain_for_tx(tx);  // Vec<PendingRewrite> in canonical order
```

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

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

### 5.4.2 Reserve (Independence Check)

For each rewrite in canonical order:



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

### 5.4.3 Execute (Parallel, Lockless)

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

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

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

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

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

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

#### 5.4.4 Merge (Canonical Sort)

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

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

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

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

#### 5.4.5 Finalize

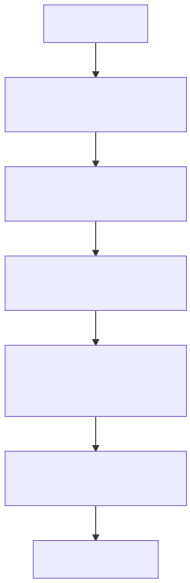
Apply the merged delta to produce the new state:

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

### 5.5 Phase 4: Hash Computation

#### State Root (BLAKE3)

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



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

### Commit Hash (v2)

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

## 5.6 Phase 5: Record to History

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

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

---

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

## 6.1 What is BOAW?

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

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

## 6.2 The Key Insight

```
                    THE BOAW INSIGHT


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

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

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

## 6.3 Execution Strategies

### Phase 6A: Stride Partitioning (Legacy)

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

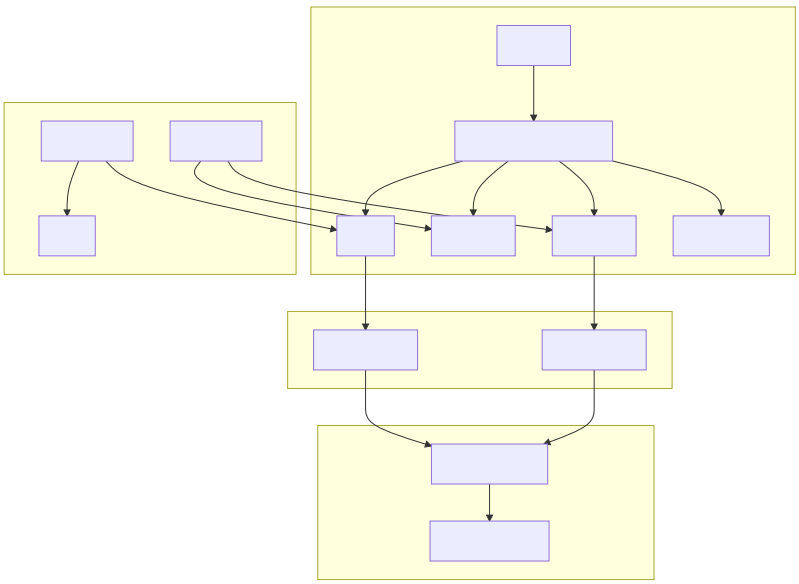
**Problem**: Poor cache locality—related items scatter across workers.

### Phase 6B: Virtual Shards (Current Default)

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

fn shard_of(node_id: &NodeId) -> usize {
    let bytes = node_id.as_bytes();
    let val = u64::from_le_bytes(bytes[0..8]);
```

```
    (val & 255) as usize  // Fast modulo via bitmask
}
```



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

## 6.4 The Execution Loop

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

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

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

```
                    if shard_id >= NUM_SHARDS { break; }

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

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

## 6.5 The Canonical Merge

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

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

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

    Ok(result)
}
```

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

---

# 7. Storage & Hashing: Content-Addressed Truth

## 7.1 The GraphStore

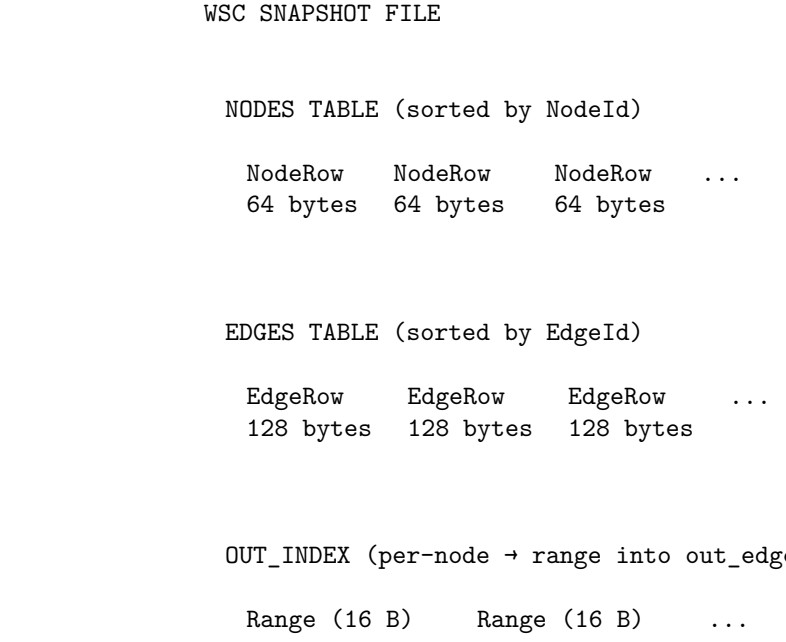
Located in `crates/warp-core/src/graph.rs`:

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

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

## 7.2 WSC: Write-Streaming Columnar Format

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

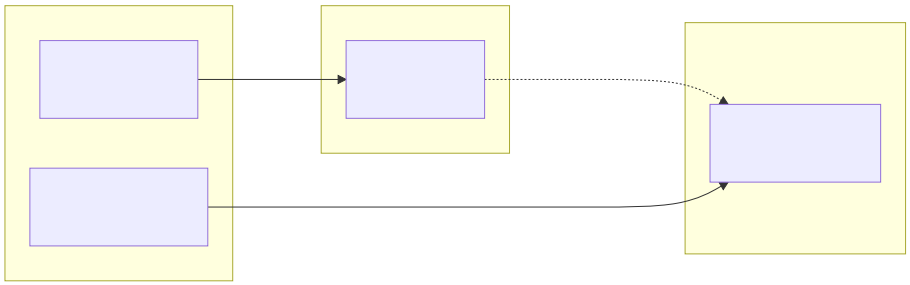


```
BLOB ARENA (variable-length data)
Referenced by (offset, length) tuples
```

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

## 7.3 Copy-on-Write Semantics

**Rule**: During a tick, nothing shared is mutated.



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

## 7.4 Hash Algorithm Details

**State Root** (BLAKE3, v2):

```
state_root = BLAKE3(
    root_id[32]                    ||
    instance_count[8, LE]          ||
    for each instance in BTreeMap order:
        warp_id_len[8, LE]         ||
        warp_id_bytes              ||
        node_count[8, LE]          ||
        for each node in ascending NodeId order:
            node_id[32]            ||
            node_type[32]          ||
            for each outbound edge in ascending EdgeId order:
                edge_id[32]        ||
                edge_type[32]      ||
                to_node[32]        ||
            for each attachment:
                key_len[8, LE]     ||
                key_bytes          ||
```

```
                type_id[32]         ||
                value_len[8, LE]    ||
                value_bytes
)
```

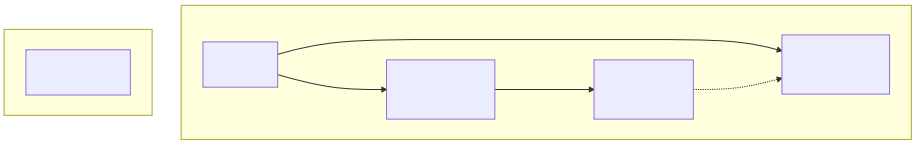
---

# 8. Worked Example: Tracing a Link Click

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

## 8.1 The Scenario

Imagine a simple site with two pages:



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

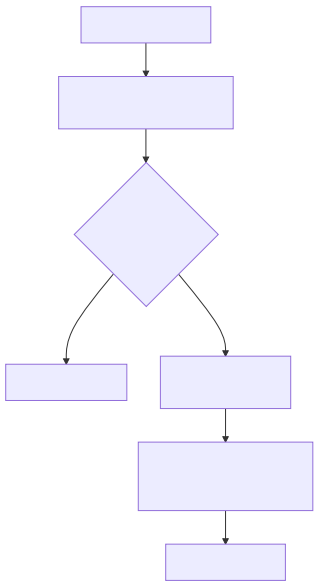
## 8.2 Step 1: Intent Ingestion

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

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

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

**What happens inside `ingest_intent`**:



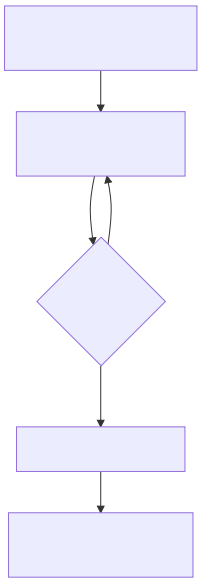
### 8.3 Step 2: Begin Transaction

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

### 8.4 Step 3: Dispatch Intent

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

**What happens**:



## 8.5 Step 4: Rule Matching

The `cmd/navigate` rule matches:

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

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

The rule is enqueued:

```
PendingRewrite

rule_id: "cmd/navigate"
scope: 0xABCD... (intent node)
```

```
footprint: { n_read: [intent, viewer], a_write: [current] }
tx: TxId(1)
```

## 8.6 Step 5: Commit

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

### 5a. Drain

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

### 5b. Reserve

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

### 5c. Execute

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

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

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

**TickDelta now contains**:

```
[
    (WarpOp::SetAttachment {
        node: viewer_node,
        key: "current",
        value: about_node_id
```

```
    }, OpOrigin { intent_id: 1, rule_id: 42, match_ix: 0, op_ix: 0 })
]
```

#### 5d. Merge

Only one delta, trivial merge:

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

#### 5e. Finalize

Apply to state:

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

### 8.7 Step 6: Hash Computation

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

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

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

### 8.8 Step 7: Emit to Tools

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

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

```
    state_hash: 0x7890...,
}
```

## 8.9 Step 8: Viewer Applies Diff

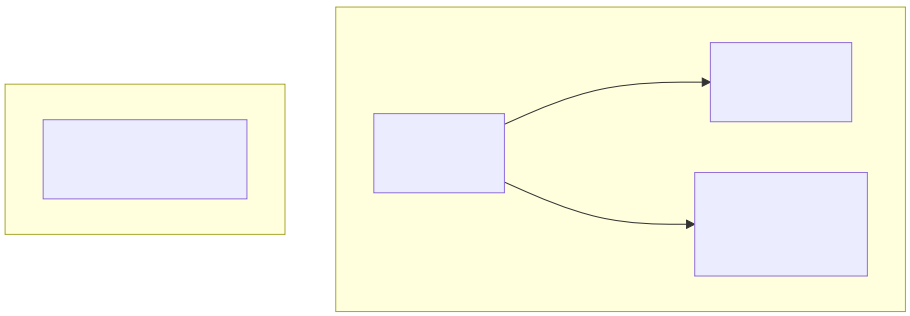
```rust
// In warp-viewer:
fn process_frames(viewer: &mut ViewerState, frames: Vec<WarpFrame>) {
    for frame in frames {
        match frame {
            WarpFrame::Diff(diff) => {
                // Verify we have the parent epoch
                assert_eq!(viewer.epoch, Some(diff.from_epoch));

                // Apply each operation
                for op in diff.ops {
                    viewer.wire_graph.apply_op(op);
                }

                // Update epoch
                viewer.epoch = Some(diff.to_epoch);

                // Verify hash matches!
                let computed = viewer.wire_graph.state_hash();
                assert_eq!(computed, diff.state_hash, "DESYNC!");
            }
            // ...
        }
    }
}
```

## 8.10 The Result

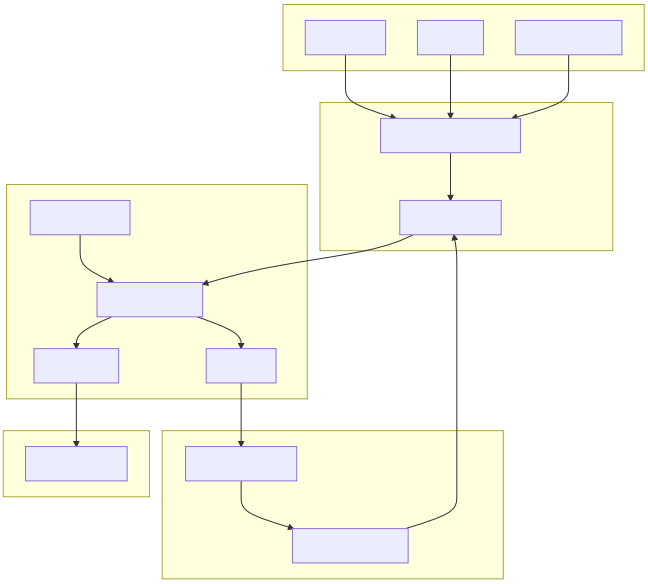


**The navigation is complete.** The viewer now displays the About page, and the state hash proves it happened deterministically.

# 9. The Viewer: Observing Echo

## 9.1 Event Handling Architecture

The viewer uses a **pure reducer pattern** (similar to Redux/Elm):



## 9.2 The UiEvent Enum

```rust
pub enum UiEvent {
    // Menu navigation
    ConnectClicked,
    SettingsClicked,
    ExitClicked,

    // Connection form
    ConnectHostChanged(String),
    ConnectPortChanged(u16),
    ConnectSubmit,

    // Overlays
    OpenMenu,
    CloseOverlay,
    OpenSettingsOverlay,
```

```
    // System
    ShutdownRequested,
    EnterView,
    ShowError(String),
}
```

## 9.3 The Pure Reducer

```
pub fn reduce(ui: &UiState, ev: UiEvent) -> (UiState, Vec<UiEffect>) {
    let mut next = ui.clone();
    let mut fx = Vec::new();

    match ev {
        UiEvent::ConnectClicked => {
            next.title_mode = TitleMode::ConnectForm;
        }
        UiEvent::ConnectSubmit => {
            next.screen = Screen::Connecting;
            fx.push(UiEffect::RequestConnect);
        }
        UiEvent::EnterView => {
            next.screen = Screen::View;
        }
        // ...
    }

    (next, fx)
}
```

**Benefits**: - **Testable**: Pure function, easy to unit test - **Predictable**: Same input always produces same output - **Debuggable**: State transitions are explicit

## 9.4 Frame Loop

Each frame:

```
pub fn frame(&mut self) {
    // 1. Drain session notifications
    for notification in self.session.drain_notifications(64) {
        self.handle_notification(notification);
    }

    // 2. Process incoming frames
    let frames = self.session.drain_frames(64);
    let outcome = process_frames(&mut self.ui, &mut self.viewer, frames);
```

```
    // 3. Handle state changes
    if outcome.enter_view {
        self.apply_ui_event(UiEvent::EnterView);
    }

    // 4. Handle pointer interaction (3D view)
    if self.ui.screen == Screen::View {
        self.handle_pointer(dt, aspect, width, height, window);
    }

    // 5. Render UI
    match self.ui.screen {
        Screen::Title => draw_title_screen(ctx, self),
        Screen::View => draw_view_hud(ctx, self),
        // ...
    }
}
```

---

## 10. Glossary

| Term | Definition |
|---|---|
| **WARP** | Worldline Algebra for Recursive Provenance—Echo's graph formalism |
| **BOAW** | Bag of Autonomous Workers—parallel execution architecture |
| **Tick** | One complete cycle of the engine (begin → apply → commit) |
| **Footprint** | Declaration of resources a rule will read/write |
| **TickDelta** | Accumulator for operations during execution |
| **WarpOp** | A single graph mutation operation |
| **GraphView** | Read-only wrapper enforcing BOAW contract |
| **Snapshot** | Immutable, hashable state at a point in time |
| **WSC** | Write-Streaming Columnar—zero-copy snapshot format |
| **State Root** | BLAKE3 hash of reachable graph state |
| **Commit Hash** | Combined hash of state + patch + parents + policy |
| **Intent** | External input that causes state changes |
| **MaterializationBus** | Channel system for emitting data to tools |
| **Scheduler** | Component ensuring deterministic rewrite ordering |
| **Virtual Shard** | Cache-locality optimization (256 shards) |

| Term | Definition |
|---|---|
| **OpOrigin** | Metadata tracking which intent/rule produced an op |

---

# Appendix A: Key File Locations

| Component | Path | Lines |
|---|---|---|
| Engine | `crates/warp-core/src/engine_impl.rs` | 302-954 |
| GraphStore | `crates/warp-core/src/graph.rs` | 1-300 |
| GraphView | `crates/warp-core/src/graph_view.rs` | 42-100 |
| Scheduler | `crates/warp-core/src/scheduler.rs` | 59-712 |
| Snapshot | `crates/warp-core/src/snapshot.rs` | 49-263 |
| TickDelta | `crates/warp-core/src/tick_delta.rs` | 38-172 |
| BOAW Exec | `crates/warp-core/src/boaw/exec.rs` | 38-192 |
| BOAW Shard | `crates/warp-core/src/boaw/shard.rs` | 82-120 |
| BOAW Merge | `crates/warp-core/src/boaw/merge.rs` | 36-75 |
| UI State | `crates/warp-viewer/src/ui_state.rs` | 8-127 |
| Viewer Frame | `crates/warp-viewer/src/app_frame.rs` | 24-349 |

---

# Appendix B: Architecture Decision Records

| ADR | Title | Key Decision |
|---|---|---|
| ADR-0001 | Two-Plane Model | Separate skeleton from attachments |
| ADR-0002 | WarpInstances | Flattened indirection for nested graphs |
| ADR-0003 | MaterializationBus | Causality-first API, no direct writes |
| ADR-0004 | No Global State | Dependency injection only |
| ADR-0005 | Physics | Deterministic scheduled rewrites |
| ADR-0006 | Ban Non-Determinism | CI enforcement scripts |
| ADR-0007 | BOAW Storage | Immutable base + overlay + merge |

---

*Document generated 2026-01-18. For the latest information, consult the source code and ADRs.*