RUNTIME LOOP v0.1

Status: Draft → locks when first compliant implementation passes validation
Domain: EngAIn / Enginality
Dependencies: ENGINEBILITY_CORE_CONTRACT_v0.1, ZON4D Temporal Law, AP Rule Engine
Role: Defines the atomic execution cycle that enforces Enginebility Contract at kernel clock.

This is **the implementation blueprint** for any compliant Enginality runtime.

OVERVIEW

The Runtime Loop is a **deterministic 11-step pipeline** executed once per Engine Tick.

Core Principle:
> The Loop exists only to enforce the Enginebility Contract. Any step that cannot guarantee contract compliance MUST abort the Tick and revert to last known-good Snapshot.

Tick Rate:
- Target: 60Hz (16.67ms per Tick) for real-time performance
- Guarantee: Deterministic output regardless of tick duration
- Degradation: If Tick exceeds budget → temporal fence + alert (never skip validation)

IMPLEMENTATION REQUIREMENTS (v0.1)

Before implementing the 11-step pipeline, the following concrete specifications must be locked to avoid ambiguity during code development.

1. DELTA STRUCTURE

Every Delta MUST contain these fields:

```
Delta:
  - id: UUID (unique Delta identifier)
  - source_id: EngineDomainID (user/AI/AP/temporal/external)
  - entity_ref: ZON4D path (e.g., "world/entities/door_01/state")
  - temporal_index: TIndex (int or rational, ZON4D t-coordinate)
  - temporal_scope: [t_start, t_end] or [t, t] for point mutations
```

- parent_ids: [DeltaID] (causality chain for ordering)
- payload: typed value/operation (ZON4D typed)
- metadata: Dict (freeform, not used in deterministic logic)
```

**Future Reference Rules:**
- **v0.1 Default:** If Delta references a Snapshot not yet created → **REJECT**
- **Branch Mode:** Only if `branch_policy` flag is explicitly set in config may the engine create a branch Timeline
- Never silently accept future references

**Causality Chain Rules:**
- Maximum causality depth: **64 levels** (prevents pathological graphs)
- Circular causality detected → **REJECT entire chain**
- Orphaned parent_ids (reference non-existent Deltas) → **REJECT**

## 2. CONFLICT DETECTION PERFORMANCE

**Upper Bounds (per Tick, per domain):**
- Maximum Deltas processed: **1,024** (default, configurable)
- If exceeded → apply temporal fence, push overflow to next Tick

**Conflict Matrix Optimization:**
- Conflict detection MUST be indexed by entity/field first (bucket by ZON4D path)
- Within each bucket, pairwise conflict checks (NOT global $O(n^2)$)
- Empty buckets skipped entirely

**Conflict Definition:**
Two Deltas conflict if ALL of:
1. Target same entity/field (ZON4D path match)
2. Temporal scopes overlap
3. Payloads are incompatible (deterministic merge policy fails)

## 3. AP ARBITRATION TIME BUDGETS

**Per-Tick AP Budget:**
- **Preflight (Step 5):** Max 5ms per Delta (configurable)
- **Finalization (Step 8):** Max 10ms total (configurable)
- If budget exceeded → apply timeout policy

**Timeout Policies:**
- **Preflight timeout:** REJECT the Delta (proceed with remaining Deltas)
- **Finalization timeout:** CRITICAL breach → halt engine, rollback to anchor
- Arbitration timeouts logged as CRITICAL alerts
```

Blocking Behavior:
- Preflight MAY reject instead of blocking indefinitely
- Finalization MUST NOT proceed without AP verdict (blocking is allowed, timeout triggers breach)

4. HYDRATION DEGRADATION POLICIES

Domain Classification:

Critical Domains (STRICT mode - failure triggers rollback):
- Spatial (3D positions, collision, navigation)
- AP Rule Domain (constraints, triggers)

Non-Critical Domains (GRACEFUL mode - failure triggers degradation):
- Audio (can mute/fade to silence)
- Visual (can reduce quality, disable effects)
- Animation (can fallback to T-pose)

Degradation Protocol:
1. Hydration failure detected in Step 7
2. Check domain classification
3. If critical → rollback entire Snapshot
4. If non-critical → disable domain, emit WARNING alert, continue
5. Log degradation state for operator review

Configuration:
Domains marked as `critical: true/false` in engine config. Default classifications above can be overridden per deployment.

5. ROLLBACK SCOPE

Two rollback paths:

Fast Path (Inverse Delta Rollback)
When:
- Breach occurs within **current Tick only** (Steps 2-8)
- Timeline hash chain intact (no corruption detected)
- All inverse Deltas successfully computed (Step 6)

Action:
1. Apply inverse Deltas in reverse order
2. Restore previous Snapshot as canonical
3. Discard current Tick's mutations
4. Resume from last good state

Performance: O(n) where n = number of Deltas in failed Tick

Slow Path (Anchor Restore)
When:
- Timeline corruption suspected (hash mismatch, missing anchors)
- AP global contradiction (Step 8 finalization)
- Fast path rollback fails
- Breach spans multiple Ticks

Action:
1. Find most recent **Immutable Anchor**
2. Restore entire Snapshot from anchor
3. Discard ALL Ticks after anchor
4. Rebuild Timeline from anchor forward (if possible)
5. Alert operator for manual intervention

Performance: O(1) for anchor load, but loses all progress since anchor

Decision Rule:
```
if breach_step in [2, 3, 4, 5, 6, 7] and timeline_hash_ok:
    use fast_path_rollback()
elif breach_step == 8 or timeline_hash_mismatch:
    use slow_path_rollback()
else:
    use slow_path_rollback()  # safest default
```

THE 11-STEP PIPELINE

STEP 1: TICK INITIALIZATION

Purpose: Prepare clean execution context for this Tick.

Actions:
1. Increment Tick counter (monotonic, never reset)
2. Capture wall-clock timestamp (for logging only, NOT used in temporal logic)
3. Load current canonical Snapshot reference
4. Initialize empty Delta queue
5. Reset conflict detection buffer
6. Clear previous Tick's alert log

Enforces:

- G1 (Deterministic Mutation) - Clean slate per Tick
- G4 (Zero Ambiguity) - Explicit state boundaries

Breach Conditions:
- Tick counter overflow → CRITICAL alert, halt engine
- Cannot load canonical Snapshot → revert to last immutable anchor

Output:
- Initialized Tick context ready for Delta ingestion

STEP 2: DELTA QUEUE INGESTION

Purpose: Collect all pending mutations from all sources.

Input Sources:
1. User actions (input events, UI commands)
2. AI agent decisions (MrLore, ClutterBot, Trae outputs)
3. AP rule-generated Deltas (conditional triggers)
4. Temporal system (scheduled events, time-curve evaluations)
5. External integrations (network, file I/O)

Actions:
1. Call `ingest(delta)` for each pending Delta
2. Validate Delta structure (type, fields, temporal metadata)
3. Reject malformed Deltas immediately (R1 - Non-Canonical Writes)
4. Queue valid Deltas for temporal ordering

Enforces:
- P1 (Real-Time Delta Ingestion) - No Delta sits unprocessed
- R1 (Non-Canonical Writes) - Structural validation

Breach Conditions:
- Unknown Delta type → reject, log, continue
- Missing temporal metadata → reject (R4 - Ambiguous Temporal Scope)
- Queue overflow → temporal fence, alert operator

Output:
- Populated Delta queue (unsorted)
- Rejection log for malformed Deltas

STEP 3: TEMPORAL ORDERING

Purpose: Sort Deltas into deterministic execution order.

Algorithm:
1. Primary sort: `temporal_index` (ZON4D t-coordinate)
2. Secondary sort: causality depth (`len(parent_ids)`)
3. Tertiary sort: `source_id` (deterministic tiebreaker)

Critical Rules:
- Deltas targeting same temporal index MUST have explicit ordering via causality chain
- If ordering ambiguous → flag as conflict for Step 4
- Never use wall-clock time for ordering (violates G1)

Causality Validation:
- Maximum depth: 64 levels (per Implementation Requirements §1)
- Circular causality detected → REJECT entire chain
- Orphaned `parent_ids` → REJECT Delta
- Future Snapshot references → REJECT (unless `branch_policy` enabled)

Enforces:
- G1 (Deterministic Mutation) - Same Deltas → same order
- Section 31 (Temporal Syncpoints) - Anchor-based ordering
- Implementation Requirements §1 - Delta structure compliance

Breach Conditions:
- Circular causality chain → reject entire chain (R2 - Unanchored Mutations)
- Delta references future Snapshot → reject per Implementation Requirements §1
- Causality depth exceeds 64 → reject chain

Output:
- Ordered Delta sequence ready for conflict detection
- Flagged ambiguous orderings
- Rejected Deltas (causality violations)

STEP 4: CONFLICT SET DETECTION

Purpose: Identify Deltas that mutate overlapping canonical state.

Performance Bounds:
- Maximum Deltas per Tick: **1,024** (per Implementation Requirements §2)
- If exceeded → temporal fence, overflow pushed to next Tick

- Conflict detection MUST use entity/field bucketing (NOT O(n²) naive matrix)

Conflict Definition:
Two Deltas conflict if ALL of:
1. Target same `entity_ref` (ZON4D path match)
2. `temporal_scope` values overlap
3. Payloads are incompatible (deterministic merge fails)

Actions:
1. Bucket Deltas by `entity_ref` (group by ZON4D path)
2. For each bucket with >1 Delta:
   - Pairwise conflict checks within bucket
   - Check if deterministic merge policy exists
   - If yes → apply merge, mark resolved
   - If no → flag for AP arbitration (Step 5)
3. Tag conflicting Deltas with conflict set ID
4. Skip empty buckets (optimization)

Enforces:
- P2 (Concurrency Support) - Multi-source mutations handled
- Section 24 (Oracle Law) - Conflict arbitration preparation
- Implementation Requirements §2 - Performance bounds

Breach Conditions:
- Delta count exceeds 1,024 → temporal fence, alert
- Merge policy produces invalid state → reject merge, escalate to AP
- Conflict set too large (>threshold) → temporal fence, manual review

Output:
- Resolved conflicts (merged Deltas)
- Unresolved conflicts (flagged for Step 5)
- Overflow Deltas (if queue exceeded)

STEP 5: AP PREFLIGHT

Purpose: Validate Deltas against AP rules BEFORE mutation.

Time Budget:
- Max 5ms per Delta (per Implementation Requirements §3)
- If timeout → REJECT Delta, proceed with remaining
- Preflight MAY reject instead of blocking indefinitely

Actions:
1. For each Delta (or merged Delta):
   - Load applicable AP Rule Set for target entity/field
   - Evaluate with timeout: ACCEPT / REJECT / ARBITRATE
2. ACCEPT → continue to Step 6
3. REJECT → discard Delta, log reason, alert originator
4. ARBITRATE → invoke AP arbitration protocol:
   - Present conflict context to AP Kernel
   - Wait for arbitration result (with 5ms timeout)
   - If timeout → REJECT Delta, log CRITICAL alert
   - If resolved → apply arbitrated resolution

Enforces:
- G3 (AP Compliance) - No commits without rule validation
- R3 (AP Contradictions) - Explicit arbitration path
- Section 20 (AP Hooks on ZON4D) - Rule evaluation points
- Implementation Requirements §3 - Time budgets

Breach Conditions:
- AP Rule Set unavailable → temporal fence, cannot proceed
- Arbitration timeout (>5ms) → REJECT Delta, CRITICAL alert
- AP verdict contradicts ZON4D typing → CRITICAL breach, halt

Output:
- ACCEPTED Deltas ready for application
- REJECTED Deltas logged (including timeouts)
- ARBITRATED Deltas resolved and ready

STEP 6: DELTA APPLICATION

Purpose: Mutate the canonical Snapshot.

Actions:
1. Clone current canonical Snapshot (copy-on-write)
2. For each ACCEPTED Delta (in temporal order):
   - Apply mutation to cloned Snapshot
   - Log: old value, new value, Delta ID, causality chain
   - Compute inverse Delta for reversibility (G2)
3. Validate resulting Snapshot:
   - Check ZON4D type constraints
   - Verify no orphaned references
   - Confirm all mandatory fields present

Enforces:
- G1 (Deterministic Mutation) - Ordered application
- G2 (Reversible Edits) - Inverse Deltas computed
- R1 (Non-Canonical Writes) - Type validation post-mutation

Breach Conditions:
- Mutation produces invalid ZON4D state → rollback to pre-Tick Snapshot
- Inverse Delta cannot be computed → reject mutation (G2 violation)

Output:
- New candidate Snapshot (not yet canonical)
- Mutation log with inverse Deltas
- Validation status

STEP 7: SNAPSHOT HYDRATION

Purpose: Populate all execution domains from the new Snapshot.

Domains:
1. **Narrative** - dialogue state, story flags, character relationships (non-critical)
2. **Spatial** - 3D positions, collision data, navigation graphs (CRITICAL)
3. **Audio/Visual** - sound state, animation curves, camera rules (non-critical)
4. **AP Rule Domain** - active constraints, trigger conditions (CRITICAL)

Actions:
1. For each domain:
   - Extract relevant fields from Snapshot
   - Hydrate domain-specific structures
   - Validate domain consistency
2. If any domain fails hydration:
   - Log failure reason
   - Check domain classification (Implementation Requirements §4):
     - **CRITICAL** (Spatial, AP Rule): rollback entire Snapshot (STRICT mode)
     - **Non-Critical** (Audio/Visual, Narrative): disable domain, emit WARNING, continue (GRACEFUL mode)
3. Log degradation state for operator review

Enforces:
- P4 (Cross-Domain Hydration) - One Snapshot → all domains
- G4 (Zero Ambiguity) - State vs Memory separation maintained
- Implementation Requirements §4 - Degradation policies

Breach Conditions:
- Critical domain (Spatial, AP Rule) fails → rollback
- Non-critical domain fails → degrade, alert
- Hydration produces inconsistent cross-domain state → rollback

Output:
- Fully hydrated multi-domain state
- Degradation flags (if any)
- Hydration success status

STEP 8: AP FINALIZATION

Purpose: Post-commit rule validation.

Why Needed:
Preflight (Step 5) validates individual Deltas. Finalization validates the **combined result** after all mutations applied.

Time Budget:
- Max 10ms total for finalization (per Implementation Requirements §3)
- If timeout → CRITICAL breach, halt engine, rollback
- Finalization MUST NOT proceed without AP verdict (blocking allowed, timeout triggers breach)

Actions:
1. Load AP Rule Set for entire world state
2. Evaluate global constraints with timeout (e.g., "no player in two locations", "narrative flags consistent")
3. Verdict:
   - ACCEPT → proceed to Step 9
   - REJECT → rollback to pre-Tick Snapshot, CRITICAL alert
   - ARBITRATE → invoke AP, apply resolution, re-validate (with remaining time budget)

Enforces:
- G3 (AP Compliance) - Global state validation
- Section 24 (Oracle Law) - Kernel arbitration
- Implementation Requirements §3 - Time budgets

Breach Conditions:
- Global constraint violated → rollback, log violation
- Finalization timeout (>10ms) → CRITICAL breach, halt engine
- AP arbitration produces new violation → CRITICAL, halt
- Re-validation after arbitration fails → CRITICAL, halt

Output:

- Finalized AP-compliant Snapshot
- Global constraint validation status

STEP 9: ANCHOR UPDATE PROTOCOL

Purpose: Mark the new Snapshot as canonical and update anchors.

Actions:
1. Compute Snapshot hash (Section 32 - HASH32 Spec)
2. Compare to previous Snapshot hash (Section 33 - Snapshot Diff)
3. Update Timeline:
   - Append new Snapshot to canonical Timeline
   - Link to previous Snapshot via hash chain
4. Anchor Classification:
   - **Soft Anchor**: Normal Tick, can be retconned later
   - **Hard Anchor**: Explicitly marked immutable checkpoint
   - **Immutable Anchor**: Critical savepoint, never modified
5. Prune old Snapshots per retention policy (keep anchors, discard intermediates)

Enforces:
- Section 31 (Temporal Syncpoints) - Hash-verified anchors
- Section 32 (Canonical Hashing) - Deterministic hash computation
- P3 (Temporal Fences) - Anchor mutability rules

Breach Conditions:
- Hash collision detected → CRITICAL, investigate data corruption
- Cannot link to previous Snapshot → Timeline discontinuity, halt
- Anchor promotion fails validation → revert to soft anchor

Output:
- New canonical Snapshot with hash anchor
- Updated Timeline reference
- Pruned historical Snapshots (if applicable)

STEP 10: CROSS-DOMAIN EXPOSURE

Purpose: Make the new canonical state available to all subsystems.

Subsystems:
1. **Renderer** - 3D scene, camera, lighting

2. **Audio Engine** - sound state, music transitions
3. **Animation System** - character poses, facial visemes
4. **Dialogue Engine** - active speaker, emotion curves
5. **AI Agents** - world state for decision-making
6. **Network Layer** - state sync for multiplayer (future)

Actions:
1. For each subsystem:
   - Expose relevant domain data from hydrated Snapshot (Step 7)
   - Signal state update event
   - Provide read-only access (no direct mutation allowed)
2. Log exposure timestamps for debugging

Enforces:
- G4 (Zero Ambiguity) - Subsystems read State, not History or Memory
- P4 (Cross-Domain Hydration) - All domains receive consistent state

Breach Conditions:
- Subsystem attempts direct state mutation → block, alert
- Exposure fails for critical subsystem (Renderer) → degrade, temporal fence

Output:
- State exposed to all subsystems
- Subsystem acknowledgment signals
- Exposure success log

STEP 11: PERFORMANCE PASS SCHEDULING

Purpose: Queue rendering, audio, and animation tasks based on new state.

Performance Domains:
1. **Visual** - render frames, camera moves
2. **Audio** - play sounds, update envelopes (Section 19)
3. **Animation** - apply curves, blend poses
4. **Dialogue** - trigger voice lines, update visemes (Section 18)
5. **SceneTrack** - manage multi-track performance (Section 22)

Actions:
1. For each domain:
   - Compare new state to previous state (delta detection)
   - Queue only changed elements (optimization)
   - Respect temporal curves (ZON4D time → performance time)

2. Schedule next Tick:
   - If real-time mode: schedule at target Hz (60Hz)
   - If step mode: wait for manual trigger
   - If overloaded: insert temporal fence, skip frame if needed

Enforces:
- Section 13 (Dialogue Intensity Tracks) - Emotion-driven performance
- Section 19 (Audio Envelopes) - Sound curve evaluation
- Section 22 (SceneTrack) - Multi-track synchronization

Breach Conditions:
- Scheduling budget exceeded → temporal fence, drop low-priority tasks
- Performance queue overflow → alert, degrade quality

Output:
- Scheduled performance tasks
- Next Tick scheduled
- **TICK COMPLETE**

TICK COMPLETE

At this point, the Engine has:
1. ✓ Ingested all pending Deltas
2. ✓ Ordered them deterministically
3. ✓ Resolved conflicts
4. ✓ Validated with AP (preflight + finalization)
5. ✓ Mutated the canonical Snapshot
6. ✓ Hydrated all domains
7. ✓ Anchored the new state
8. ✓ Exposed state to subsystems
9. ✓ Scheduled performance tasks

The Loop repeats.

BREACH HANDLING ACROSS STEPS

If **any step** detects a contract violation:

1. **HALT** the current Tick (do not proceed to next step)
2. **EMIT** CRITICAL alert to Kernel/ZWAlerts

3. **PERSIST** repro data:
   - Current Snapshot ID
   - Pending Delta queue
   - AP verdicts
   - Step where breach occurred
   - Timeline hash status
4. **REVERT** using appropriate rollback path (Implementation Requirements §5):

Fast Path Rollback (Inverse Delta)
Used when:
- Breach in Steps 2-7 (current Tick only)
- Timeline hash chain intact
- All inverse Deltas computed successfully

Process:
1. Apply inverse Deltas in reverse order
2. Restore previous Snapshot as canonical
3. Discard current Tick's mutations
4. Resume from last good state

Slow Path Rollback (Anchor Restore)
Used when:
- Breach in Step 8 (AP Finalization)
- Timeline corruption detected (hash mismatch)
- Fast path rollback fails
- Breach spans multiple Ticks

Process:
1. Find most recent Immutable Anchor
2. Restore entire Snapshot from anchor
3. Discard ALL Ticks after anchor
4. Rebuild Timeline from anchor forward (if possible)
5. Alert operator for manual intervention

Decision Rule:
```
if breach_step in [2, 3, 4, 5, 6, 7] and timeline_hash_ok:
    → fast_path_rollback()
elif breach_step == 8 or hash_mismatch:
    → slow_path_rollback()
else:
    → slow_path_rollback()  # safest default
```

Silent failures are breaches. All errors must be logged and handled explicitly.

PERFORMANCE GUARANTEES

Determinism (G1)
Given identical inputs at Step 1, Steps 2-11 produce identical outputs.

Reversibility (G2)
After Step 6, inverse Deltas exist to undo mutations.

AP Compliance (G3)
Steps 5 + 8 ensure no non-compliant state ever becomes canonical.

Temporal Fences (P3)
If any step exceeds time budget:
- Insert explicit fence marker in Timeline
- Alert subsystems to degraded mode
- Continue with best-effort (never skip validation)

INTEGRATION POINTS

With ZON4D Temporal Law:
- Step 3: Temporal Ordering (Section 31 - Syncpoints)
- Step 9: Anchor Update (Section 32 - Hashing)
- Section 34: Snapshot Consolidation (post-Tick merge protocol)

With AP Rule Engine:
- Step 5: Preflight validation
- Step 8: Finalization validation
- Section 24: Oracle Law arbitration hooks

With Performer Engine:
- Step 11: SceneTrack scheduling (Section 22)
- Dialogue Tracks (Section 13)
- Audio Envelopes (Section 19)
- Visemes (Section 18)

COMPLIANCE CRITERIA

A Runtime Loop implementation is **Enginal-compliant** if:

1. It executes all 11 steps in order
2. It enforces all Enginebility Contract clauses (G1-G4, P1-P4, R1-R4)
3. It handles breaches per Section 5 of the Contract
4. It produces deterministic output given deterministic input
5. It logs all decisions for replay/debugging

Non-compliant implementations may produce output but are not Enginality runtimes.

NEXT STEPS FOR IMPLEMENTATION

1. **Python Prototype** - Implement loop in `core/enginality/runtime_loop.py`
2. **Godot Integration** - GDScript runtime in `godot/addons/engain/enginality/`
3. **Compliance Tests** - Generate test suite validating all 11 steps
4. **Breach Simulation** - Test failure modes (malformed Deltas, AP rejections, hash collisions)
5. **Performance Profiling** - Measure step timing, optimize bottlenecks

---

Loop Ends

CHANGELOG - v0.1 Implementation Tightening

Date: Dec 6, 2025
Changes: Added Implementation Requirements section (§1-§5) based on code-readiness review

What was added:

1. **Delta Structure (§1)** - Concrete field definitions, future reference policy, causality chain limits
2. **Performance Bounds (§2)** - 1,024 Delta per-Tick limit, bucketed conflict detection (not $O(n^2)$)
3. **AP Time Budgets (§3)** - 5ms preflight, 10ms finalization, explicit timeout policies
4. **Domain Classification (§4)** - CRITICAL vs non-critical domains, degradation policies
5. **Dual Rollback Paths (§5)** - Fast path (inverse Deltas) vs slow path (anchor restore)

Steps Updated:
- Step 3: References concrete Delta structure, causality validation
- Step 4: Performance bounds, entity/field bucketing
- Step 5: Time budget enforcement, timeout handling

- Step 7: Domain criticality classification
- Step 8: Finalization time budget
- Breach Handling: Dual rollback path decision logic

Flags for Cross-Check:

⚠️ **Delta `temporal_index` type:** Spec says "int or rational" - needs alignment with ZON4D Temporal Law Section 31 (Temporal Syncpoints). Confirm whether rational/float is allowed or if it must be discrete int.

⚠️ **AP Rule Engine interface:** Time budgets assume AP can be queried with timeouts. Verify Section 20 (AP Hooks on ZON4D) and Section 24 (Oracle Law) support timeout-based evaluation.

⚠️ **Domain criticality:** Default classification (Spatial + AP Rule = CRITICAL) should be validated against existing ZON4D specs. Narrative might need to be CRITICAL depending on story-critical flags.

⚠️ **Conflict detection bucketing:** Assumes ZON4D paths are hierarchical and can be bucketed by `entity_ref`. Verify this aligns with ZON4D path structure spec.

⚠️ **Inverse Delta computation:** Step 6 requires ZON4D Kernel to compute inverse Deltas. Confirm Sections 39 (Temporal Deltas & Patch Propagation) defines inverse delta semantics.

Status: Ready for implementation prototype (Steps 2+3+6 first pass)