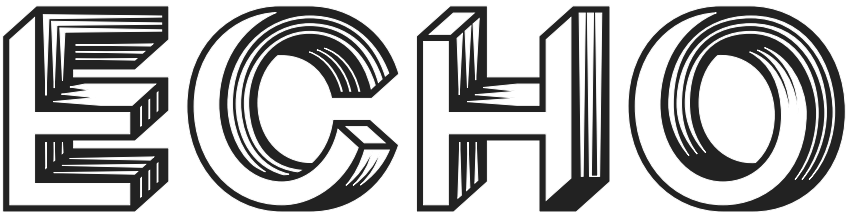

ECHO

Core Concepts

Version v0.1-draft
Commit: HEAD
December 4, 2025

James Ross & Echo Contributors

Source

This booklet is generated from the Echo repository documentation.

Foreword

Echo is a deterministic, multiverse-aware engine. This booklet walks you in with progressive layers: orient yourself, learn the core building blocks, then dive into math and operations. Each shelf can stand alone; together they form the full Echo field guide.

If you are new, start with the onboarding roadmap and glossary. If you build or extend Echo, keep the determinism contract and scheduler flow in view. Future work will deepen each part and add more diagrams as Echo evolves.

Contents

Part I

Core Concepts

Chapter 1

High-Level Architecture

Echo follows a strict **Hexagonal Architecture** (or Ports and Adapters pattern). The core domain logic sits in the center, isolated from the outside world.

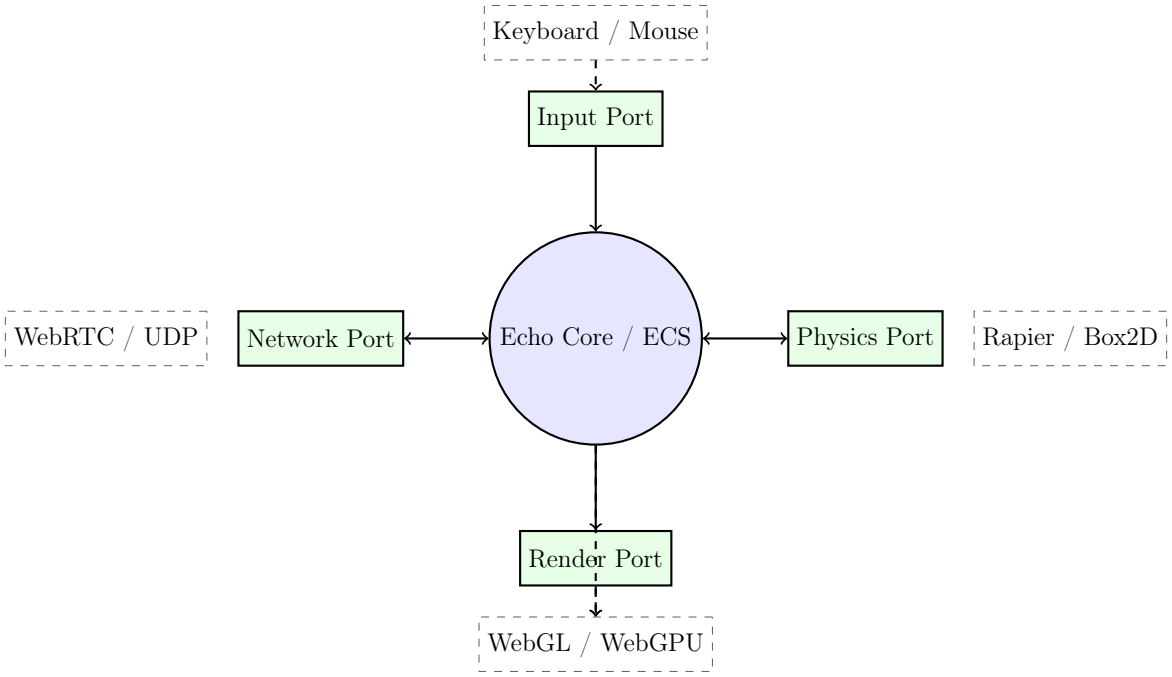

Figure 1.1: Echo's Hexagonal Architecture

1.1 Domain Layers

The architecture is stratified into clear layers:

1. **Core ECS:** The heart of the engine.
    - **Entities:** Numerical IDs managed by a high-watermark allocator.
    - **Components:** Data buckets. Storage is archetype-based (chunks sized for CPU cache lines) with Copy-on-Write (COW) support for branching.
    - **Systems:** Pure functions that transform data.

2. **World & Scene Management:** Handles the lifecycle of entities and the "Scene Graph".

3. **Time & Simulation (Timecube):** Manages the temporal axes:

   - **Chronos:** The monotonic tick counter (Sequence).
   - **Kairos:** The branch identifier (Possibility).
   - **Aion:** The narrative weight/entropy (Significance).

4. **Codex's Baby (Event Bus):** A deterministic event bus used for communication between systems and for bridging the gap between the Core and the Ports.

1.2 Ports & Adapters

Echo does not define *how* things happen, only *that* they happen.

- **Renderer Port:** Receives a 'FramePacket' containing generic draw commands (mesh refs, transforms). Adapters (e.g., PixiJS, wgpu) translate this to GPU calls.

- **Input Port:** Aggregates input into snapshots. The domain polls this once per frame.

- **Physics Port:** Dual-write system. ECS components store the "desired" state; the Physics Port (via an adapter like Rapier) advances the simulation and syncs the authoritative transform back.

- **Networking Port:** Handles replication. Thanks to determinism, it can use state-snapshots or input-replication (GGPO).

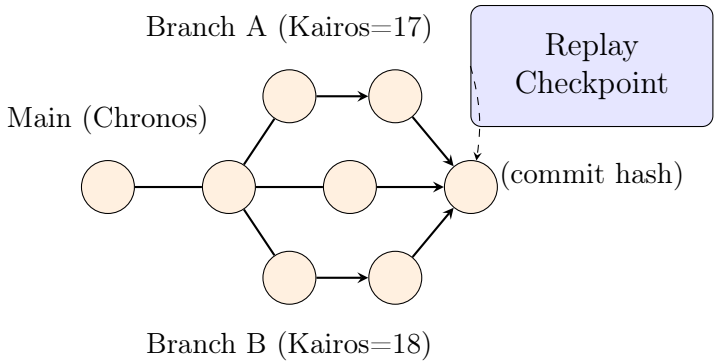

Figure 1.2: Git-style branching: Chronos is the main line, Kairos branches diverge and deterministically merge;
checkpoints use snapshot hashes (Aion weighting optional).

1.2.1 Deterministic Scheduler & Sandbox

- **Scheduler kinds:** 'SchedulerKind' lets the engine pick the default Radix scheduler (stable $O(n)$ LSD radix on `scope_hash`, `rule_id`, `nonce`) or a Legacy BTreeMap path for A/B runs.
- **Footprint conflicts:** Independence checks use generation-stamped sets over node/edge read-write sets and boundary ports (MWMR). Any overlap conflicts and the rewrite aborts before execution.
- **Sandbox A/B:** `EchoConfig` + `run_pair_determinism` build two isolated engines (different schedulers, seeds, or rule sets) and compare snapshot hashes step-by-step to prove determinism.
- **Snapshots:** Each tick can emit a `Snapshot` with a 32-byte BLAKE3 hash used for comparison, replay, or merge validation.

Chapter 2

Timeline Anatomy (Kairos Lens)

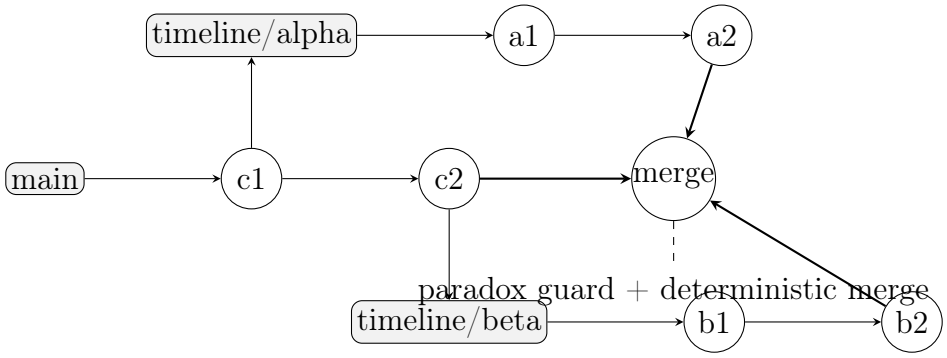

Figure 2.1: Branching timelines converge through deterministic merge + paradox guard.

Read this diagram as: main spawns experimental timelines; each accumulates commits, then merges through the paradox guard. All merges are deterministic and recorded in the branch tree for replay.

Chapter 3

Deterministic Scheduler Flow

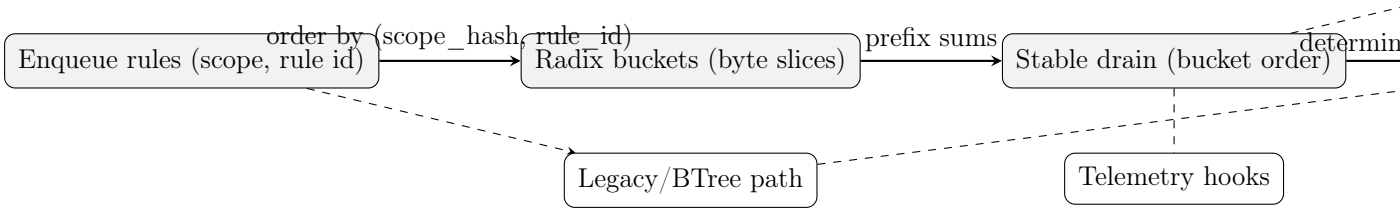

Figure 3.1: Codex's Baby (radix scheduler) keeps ordering stable via bucketed drains; conflicts are resolved deterministically before commits.

Key invariants:

- Ordering is stable: sorted by $(scope\_hash, rule\_id)$; radix buckets preserve order inside each byte slice.
- Footprint conflicts fail fast; no silent drops.
- Legacy/BTree path is optional for side-by-side comparisons.
- Telemetry and paradox guard hook in without altering order.

Chapter 4

ECS Storage Layout

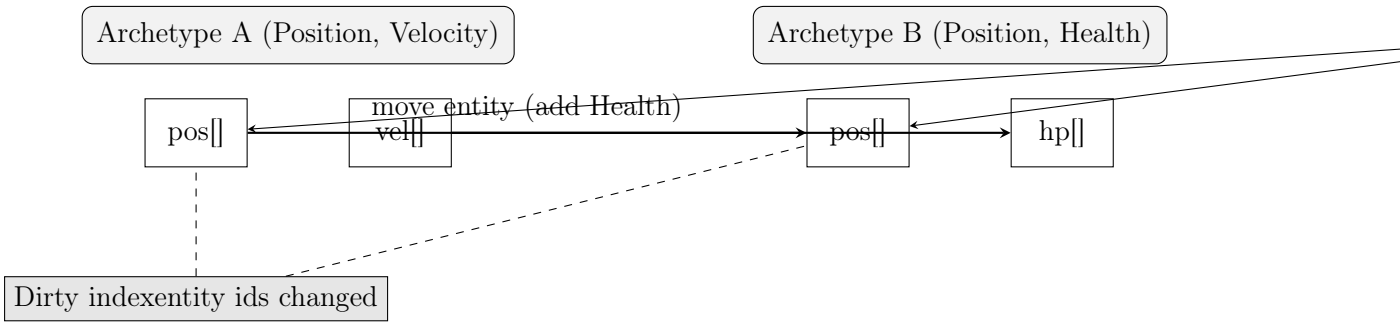

Figure 4.1: Archetypes store columnar component arrays; moving entities rewrites sparse maps
and bumps the dirty index for deterministic diffs.

Notes:

- Archetypes are columnar; moves are deterministic rewrites, not lazy copies.
- Dirty index tracks touched entities for fast diffing and replay.
- Sparse maps give O(1) entity $\rightarrow$ row lookup; kept in sync on every move.

Chapter 5

The Game Loop

Echo's game loop is a rigid, phase-based pipeline designed to ensure causality and data consistency.

5.1 The Loop Phases

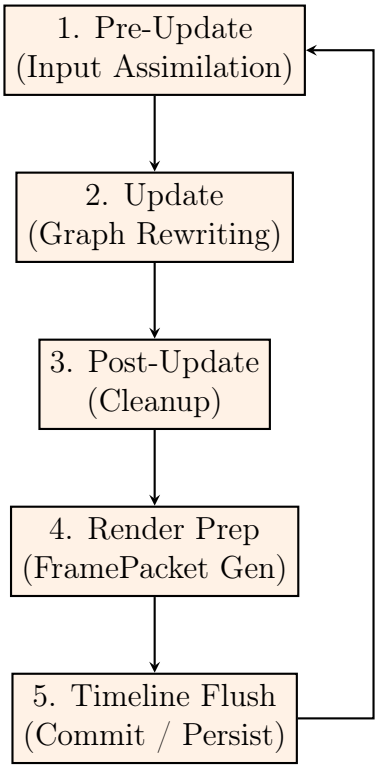

Figure 5.1: Echo's five-phase deterministic game loop.

1. **Pre-Update:**
    - Inputs from the 'InputPort' are polled and injected into the 'InputComponent'.
    - 'Codex's Baby' (Event Bus) flushes pending external events.
2. **Update (The Core):**

- The Scheduler executes Systems.
- Physics simulation steps (if sub-stepping is enabled).
- Game logic runs.

3. **Post-Update:**
   - Late-frame logic (e.g., camera smoothing).
   - Entity despawning cleanup.
4. **Render Prep:**
   - Systems read the state and produce a ‘FramePacket‘.
   - No state mutation is allowed here.
5. **Timeline Flush:**
   - The diffs for the frame are finalized.
   - The "State Root" hash is computed.
   - The frame is committed to the Timeline (Chronos).

5.2 Graph Rewriting (The Scheduler)

Inside the **Update** phase, Echo does not just "loop over arrays." It performs **Graph Rewriting**.

1. **Matching:** Systems query the graph for patterns (e.g., "Entity with Position and Velocity").
2. **Reserving:** The Scheduler identifies which matches are independent (don’t touch the same data).
3. **Execution:** Independent matches are processed. They produce a "Rewrite" (a list of changes).
4. **Application:** The Rewrites are applied to the graph state.

This "Reserve-Execute-Apply" cycle ensures that even in a parallel execution environment, the result is deterministic. If two systems try to modify the same entity, the Scheduler detects the conflict and orders them deterministically based on a topological sort of the System Graph.

License and Legal Notice

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

Code

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

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

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

Theory, Mathematics, and Documentation

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

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

at the user's option.

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

SPDX Headers

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

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

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

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

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

These identifiers correspond directly to the licenses described above.

Disclaimer

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