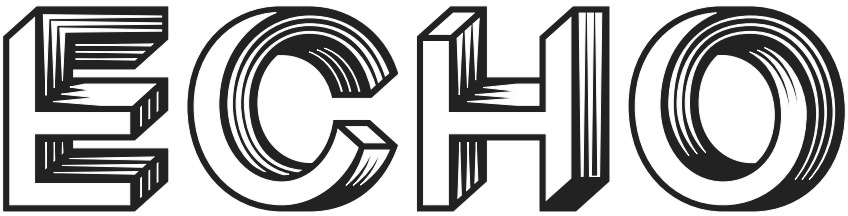

ECHO

Echo Editor Tools

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

I Echo Editor Tools **1**

1 Echo Editor Constellation of Services **3**

1.1 Echo Session Service and RMG Viewer Sync . . . 4

1.1.1 Roles . . . 4

1.1.2 Wire Types (from `echo-graph` / `echo-session-proto`) . . . 4

1.1.3 Protocol Invariants . . . 5

1.1.4 Viewer Apply Loop (gapless) . . . 5

1.1.5 Service Emit Loop . . . 5

1.1.6 Failure Handling . . . 5

1.1.7 Why Structural Ops (not rule replay) . . . 6

1.1.8 Next Steps . . . 6

1.2 Echo RMG Viewer: State Machine & UI . . . 6

1.2.1 Overview . . . 6

1.2.2 Top-Level States . . . 6

1.2.3 Transitions . . . 7

1.2.4 State Machine Diagram . . . 7

1.2.5 Screen Layouts (Wireframes) . . . 7

1.2.6 RMG Ring Layout . . . 8

1.2.7 Publish / Subscribe semantics . . . 8

1.2.8 Notes . . . 9

Part I

Echo Editor Tools

Chapter 1

Echo Editor Constellation of Services

Echo's tooling isn't a single monolithic editor. Instead, we run a constellation of small, purpose-built tools around a shared *Editor Service Hub*. Each tool speaks three channels: RMG state (diffs/snapshots), notifications/toasts, and commands/edits. The hub maintains session authority and causal ordering so every participant sees a consistent, deterministic timeline.

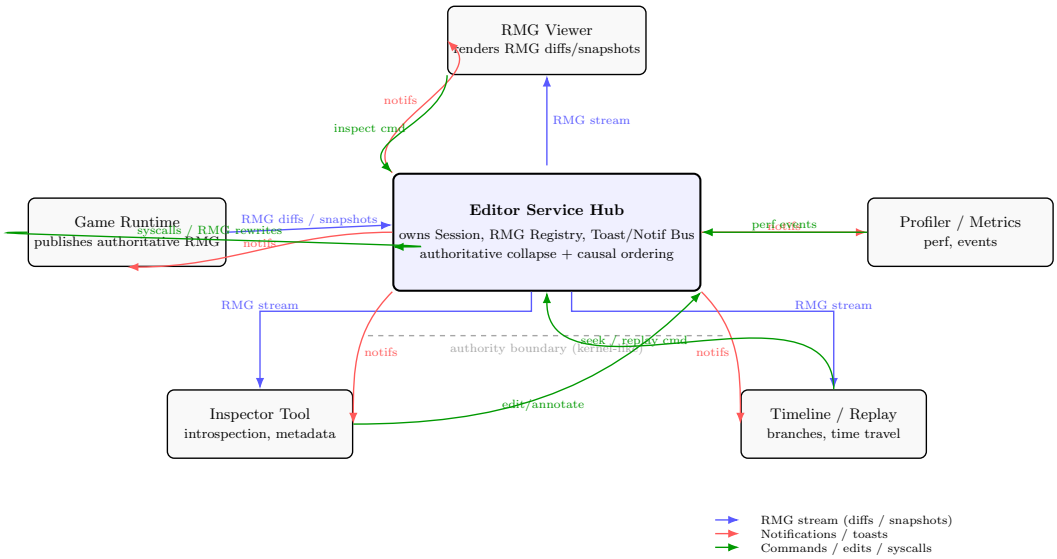

Figure 1.1: Editor constellation: the hub owns sessions and RMG registry; tools subscribe to RMG streams, receive notifications, and send commands.

Roles

- **Editor Service Hub**: authoritative session, RMG registry, notification bus; enforces causal order and collapse rules.
- **Publishers** (e.g., Game Runtime): stream RMG diffs/snapshots into the hub.
- **Consumers** (Viewer, Inspector, Timeline, Profiler): subscribe to chosen RMG streams; render, inspect, or replay; emit commands back.

Interfaces (hex-friendly)

- RMG Stream Port: ordered diffs + on-demand snapshots per RMG id.
- Notification Port: broadcast info/warn/error events (toasts/logs) scoped to session or RMG.
- Command Port: edits, inspections, timeline seeks, profiler signals; acknowledged with deterministic acks.

Why this shape It keeps UI and runtime concerns decoupled, supports multi-tool workflows (viewer + inspector + profiler simultaneously), and preserves determinism via a single authority boundary. Every tool remains a hexagonal adapter; the hub is just another adapter hosting the shared core.

1.1 Echo Session Service and RMG Viewer Sync

Echo's editor constellation streams RMG state from the engine to tools over a gapless, deterministic protocol. This section documents the session service, the wire format, and how the RMG Viewer consumes snapshots/diffs to stay in lockstep.

1.1.1 Roles

- **Engine / Session Service**: authoritative owner of the RMG; emits snapshots and diffs over a Unix socket; guarantees ordered, gapless epochs.
- **RMG Viewer**: connects to the service, receives frames, applies structural ops, renders the graph; treats any epoch gap/hash mismatch as a protocol error.
- **Other tools**: inspectors, profilers, etc., use the same frames and can render or analyze the graph identically.

1.1.2 Wire Types (from `echo-graph` / `echo-session-proto`)

- **EpochId**: monotonically increasing tick id.
- **RmgFrame**: either `Snapshot` or `Diff`.
- **Snapshot**: `{ epoch, graph: RenderGraph, state_hash? }` (first frame after connect).
- **Diff**: `{ from_epoch, to_epoch, ops: Vec<RmgOp>, state_hash? }` with `to = from + 1` in live streams.
- **RmgOp**: structural graph mutations (Add/Update/Remove node/edge with payload patches).
- **Notification**: info/warn/error scoped to Global/Session/RMG/Local; carried alongside RmgFrame.

1.1.3 Protocol Invariants

- First frame must be a Snapshot.
- Live stream is gapless: `diff.from_epoch == local_epoch`, `diff.to_epoch == local_epoch + 1`. Any violation = protocol error + disconnect.
- Optional state hashes are computed over canonical RenderGraph (sorted, CBOR) with blake3; mismatch = error.
- Late join/reconnect: service sends a fresh Snapshot (or Snapshot + diffs if it can resume); viewer resets to the Snapshot epoch.

1.1.4 Viewer Apply Loop (gapless)

```
Snapshot(s):
  wire_graph = s.graph
  epoch = s.epoch
  scene = scene_from_wire(wire_graph)
  if hash provided -> verify
  screen = View

Diff(d):
  assert d.from == epoch
  assert d.to   == epoch + 1
  for op in d.ops: wire_graph.apply_op(op)
  epoch = d.to
  if hash provided -> verify
  scene = scene_from_wire(wire_graph)
```

1.1.5 Service Emit Loop

1. On client connect: send **Snapshot** of current epoch, then begin diffs.
2. On each tick: compute structural ops from epoch k to $k + 1$, emit **Diff** with `state_hash` of post-state.
3. Ordering is the transport contract: one socket, ordered CBOR frames, length-prefixed.

1.1.6 Failure Handling

- Any gap or hash mismatch: surface an error toast/banner; drop the connection; prompt reconnect.
- Socket read/parse failure: drop connection; return to Title screen.

1.1.7 Why Structural Ops (not rule replay)

- Viewer stays lightweight and version-tolerant; no need to ship rule code.
- Determinism is preserved because ops are applied in recorded order with hashes to verify.

1.1.8 Next Steps

- Engine emitter: send real RenderGraph snapshots/diffs using `echo-graph` types.
- Viewer: improve `scene_from_wire` to honor payload-driven layout/appearance.
- Add resume-from-epoch handshake (`RmgHello`) to support reconnect without full snapshot when logs are retained.

1.2 Echo RMG Viewer: State Machine & UI

The RMG Viewer is a thin, rendering-first client. It connects to the Echo Session Host, subscribes to one or more RMG streams, and presents a minimal HUD over a 3D scene. This section fixes the viewer's states, transitions, and on-screen layout so the implementation and the design stay in lockstep.

1.2.1 Overview

- Full-screen startup flow (Title → Connecting → Viewer).
- Optional Settings overlay reused between Title and Viewer.
- Viewer HUD: Menu button, toasts stack, perf/controls/stats blocks, persistent watermark/version.
- Multiple RMGs shown on a ring around the camera; arrow keys rotate the ring to bring one RMG to the front.
- Menu overlay offers: Settings, Publish Local RMG, Subscribe to RMG, Back.

1.2.2 Top-Level States

Start Ephemeral entry; immediately transitions to Title.

Title Wordmark + version + primary menu (Connect / Settings / Exit).

Connecting Full-screen status with boot-like log lines while the session client handshakes.

Viewer 3D scene + HUD. Overlays (Menu, Settings, RMG Directory) sit atop this state.

Error Optional full-screen error (e.g., desync) with a path back to Title.

1.2.3 Transitions

- Start → Title on launch.
- Title → Connecting when Connect is pressed.
- Title → Title after Settings (Save/Back) or if connection fails/cancelled.
- Title → Exit when Exit is pressed.
- Connecting → Viewer on successful hello + directory fetch.
- Connecting → Title on failure or cancel.
- Viewer → Title on explicit disconnect.
- Any → Error on fatal protocol/stream violation; Error → Title via Back.

1.2.4 State Machine Diagram

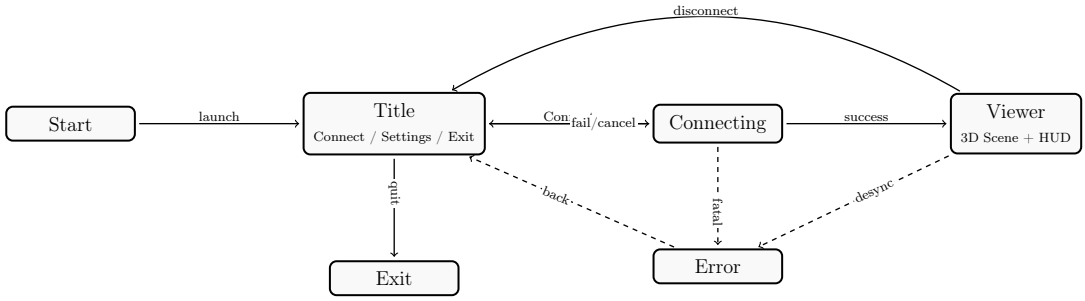

Figure 1.2: Top-level RMG Viewer state machine.

1.2.5 Screen Layouts (Wireframes)

All layouts are schematic; exact styling is left to implementation.

Title Full-screen with wordmark/version and vertical menu (Connect, Settings, Exit). Selecting Connect shows host/port fields + Connect button in-place; Settings shows app settings with Save/Back; Exit quits.

Connecting Cleared screen with wordmark/version bottom-right and a vertically stacked log, e.g.:

- Connecting...
- Connected. Sending hello...
- Session started.
- Requesting RMG directory...
- RMG indexes received.

Transitions to Viewer on success.

Viewer Full-screen 3D scene with HUD anchors:

- **Menu** button (top-left) opens overlay: Settings / Publish Local RMG / Subscribe to RMG / Back.
- **Toasts** stack (top-right).
- **Perf** block (bottom-left) showing FPS/timings (toggled in Settings).
- **Controls** block (bottom-left) showing WASD/QE, mouse look, right-drag spin, wheel zoom, arrow-keys cycle RMGs.
- **Stats** block (bottom-center/right) showing RMG id/epoch, subscription status.
- **Watermark/version** always bottom-right.

Overlay logic: Menu, Settings, and RMG Directory sit atop Viewer; closing them restores the HUD.

1.2.6 RMG Ring Layout

RMGs (local + subscribed) are placed on a horizontal ring of radius R around the camera. For n RMGs, positions are

$$x_i = R\cos\theta_i, \quad z_i = R\sin\theta_i, \quad \theta_i = \frac{2\pi i}{n} + \theta_{\text{offset}}.$$

Arrow keys adjust θ_{offset} to rotate the ring so a chosen RMG sits in front of the camera. Adding subscriptions increases n and redistributes positions evenly.

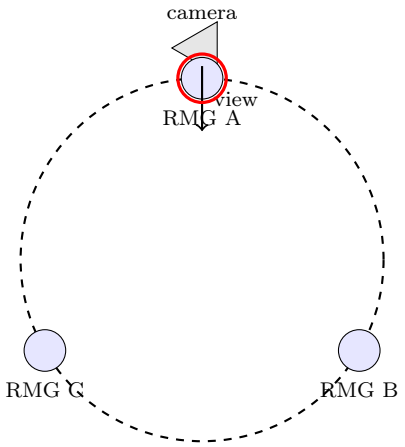

Figure 1.3: Overhead ring layout; arrow keys rotate which RMG is active.

1.2.7 Publish / Subscribe semantics

- **Publish Local RMG**: register as the sole producer for a chosen RmgId and stream Snapshot → gapless Diff to the host.
- **Subscribe to RMG**: open the RMG Directory overlay (listing host-known RmgIds); selecting one sends SubscribeRmg and adds that RMG to the ring; arrow keys cycle which RMG is in front.

1.2.8 Notes

- Error handling: on desync/hash mismatch, drop to Error or Title with a clear reconnect path; keep the watermark visible.
- All overlays reuse the same settings form (host/port, rmg id, HUD toggles) to avoid divergent UIs.

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`.