Alright — I read the PDF, and I re-read the whole thread you and I just built inside it. Here's the **clean, undeniable signal**:

What you actually have today (not vibes)

You have a **"proof receipt" object model** plus a **strict verifier** that enforces the bindings that matter:

- **TraceSpecV1**: a content-addressed "what does this trace *mean*?" type tag (format id + schema ref + encoding + field) and a hash of it.
- **StatementV1**: a content-addressed "what claim are we proving?" that binds **policy hash + trace root + public inputs + anchors**, and whose hash is injected into the proof transcript so it can't be relabeled later.
- **Capsule hash + artifact payload hashes**: the bundle and its proof files are integrity-protected.
- **Policy registry inclusion (optional)**: you already support "policy is governance-pinned to a trusted Merkle root."
- **Authorship + ACL (optional)**: you already support "only authorized keys can publish 'official' results under this policy."
- **Probabilistic DA sampling audit**: deterministic sampling + Merkle inclusion checks against the committed trace root.

That's not a demo toy. That's a **verifier-gated artifact standard**.

The one honest gap

Your PDF says it directly: you're still **geom-shaped** and missing the "others can adopt without reading your mind" layer: a first-class **Trace Adapter API** and clearer "policy semantics with teeth."

So: you've built the **receipt format + enforcement engine**; now you need the **adoption surface**.

What the world is converging toward (and why your wedge is real)

The ZK world is loudly admitting benchmarking is messy and easy to mislead on:

- zk-Harness explicitly says comparisons are hard because many factors bias outcomes. (GitHub)

- Ethereum Foundation is actively benchmarking zkVMs for Ethereum and cares about *worst-case blocks* + *tooling* + *reliability* because of real-time proving constraints. (zkevm.ethereum.foundation)
- There's already a tooling ecosystem (zkvm-perf, zkvm-bench, a16z zkvm-benchmarks), but they largely output **results**, not cryptographically enforceable **receipts**. (GitHub)
- a16z has been pushing the idea that performance claims can be technically true yet misleading without context/standardized framing. (a16z crypto)

That's your opening: make the *unit of truth* a verifiable receipt object.

The painfully straightforward convergence

Converge on this product:

CapsuleBench = "benchmarks as receipts."

A benchmark result only counts if it comes with a capsule that verifies under a published policy.

That's wedge **(1) neutral leaderboard** + **(3) sales enablement** in one:

- Leaderboard forces the standard.
- Teams pay to show up credibly.

The game plan (what to build, in order)

1) Standardize the submission UX into ONE artifact

`capsulepack.tgz` is the only thing anyone ever uploads.

Your pipeline already emits most components; you add:

- auto hardware/os/toolchain manifests
- bundling into a single pack

Rule: *If it isn't inside the capsulepack, it didn't happen.*

This eliminates adoption friction.

2) Make policy real: Benchmark Policy v1 becomes the constitution

You already bind a policy hash; now you make it enforce something.

Minimum **Benchmark Policy v1** requirements:

- **tracks**: `baseline_no_accel` vs `optimized_declared`
- **required anchors**: runner git commit, docker digest, hardware manifest hash, toolchain hash, OS hash, workload input hash
- **measurement procedure**: warmups, repetitions, aggregation rule
- **required public outputs**: prover time (declared), peak RSS, proof size, verify time, workload params, result digest

This is directly aligned with what serious benchmark repos already try to do informally ("same hardware / same program / no accel/custom params") — you're just making it *enforced and portable*. (GitHub)

3) Ship the verifier-gated leaderboard (two tracks)

- **Community track**: must verify capsule + policy hash + anchors present.
- **Certified track**: must also have authorship sig + ACL allowlist + (optionally) policy registry inclusion proof.

Your verifier already has the machinery for policy registry inclusion and ACL checks.

4) Build the missing adoption surface: Trace Adapter API

This is the "make it not geom-only" step the PDF itself flags.

Define a tiny interface so a zkVM team implements ~5–8 functions and immediately gets:

- statement construction + transcript binding
- capsule emission
- capsulepack generation
- leaderboard submission compatibility

5) Wrap existing benchmark tools instead of fighting them

Support the existing ecosystem by making CapsuleBench an output format:

- integrate/wrap **zkvm-perf** style workflows (GitHub)
- support “apples-to-apples” harness runs like **zkvm-bench** (GitHub)
- allow community benchmark suites like **a16z zkvm-benchmarks** to emit receipts (GitHub)

This makes your standard look inevitable: you’re not replacing them — you’re giving them a cryptographic receipt layer.

What the adopter actually has to do

If they’re just submitting results

They should do **two commands**:

1. `capsule-bench run --workload X --track Y`
2. `capsule-bench submit --pack capsulepack.tgz`

That’s it. No manual hashing, no writing manifests.

If they’re a zkVM team integrating

They implement:

- `build_trace_spec()`
- `extract_public_outputs()`
- `build_statement()`
- `prove()` / `verify()` hooks (or a wrapper)
- optional `export_row_archive()` if you want DA audits

Everything else is your framework.

The “don’t get killed on category mismatch” positioning

Do **not** pitch as “we solved DA” or “we’re a Celestia competitor.” Your DA today is **audited retrievability via probabilistic sampling** — valuable and honest.

Your pitch is:

> **“We make benchmarking claims reproducible objects with cryptographic integrity and policy enforcement.”**

That harmonizes with the direction EF is going (tooling + worst-case + reliability) without pretending you’re the whole Ethereum proving pipeline. (zkevm.ethereum.foundation)

The single sentence to align the team on

Build CapsuleBench: a verifier-gated, policy-bound benchmark receipt standard (capsulepacks) + leaderboard + certified track, with adapters for zkVMs.

If you want, next I’ll do two very concrete things (no fluff):

1. Write the **TraceAdapter interface spec** (the minimal methods + exact input/output schemas).
2. Write the **capsulepack file layout + server validation order** as an implementable checklist that matches your existing `verify_capsule.py` logic.

Yes — and if you do it right, it actually *amplifies* the wedge (it makes the whole thing feel inevitable), **without weakening the cryptographic story**.

The key is: **the UI is “observable progress,” not “trusted truth.”** The *truth* is still the final capsule + verifier.

What “real-time” can mean (3 levels)

Level 1: Real-time progress (easy, low risk, high value)

Users see:

- current workload / phase (trace gen → commit → prove → verify)
- % progress + ETA-ish (based on steps done)
- live metrics: CPU/GPU util, RSS, throughput, proof size so far
- logs + warnings

This is just telemetry streaming over WebSocket/SSE. No cryptographic changes needed.

Level 2: Real-time *checkpoints* that are cryptographically tied (your sweet spot)

As the computation runs, the runner periodically emits **checkpoint commitments**:

- every `k` steps: compute a chunk root / partial Merkle root (or "rolling STC root")
- publish `(checkpoint_index, checkpoint_root, bytes_written, chunk_len, arity, …)`
- UI shows a timeline: "checkpoint 17 committed → root 0x…"
- server can even do lightweight verification of internal consistency (format, hashes)

This gives the user the visceral feeling of "I'm watching it become a receipt."

Level 3: "Streaming capsule" (hardest, but insane as a demo)

You treat the capsule as a living object:

- capsule draft exists from the beginning with fixed `TraceSpec` + policy + anchors
- statement hash is defined early (or per-epoch statements if you want multiple)
- as chunks commit, the "trace_root" isn't final yet, but you maintain a verifiable evolving commitment structure
- final step: seal capsule with final root + proof + signature

This is possible, but you need to be careful about *what's bound when*.

The simplest architecture that works

Runner (client-side)

- Runs workload + trace generation + commitments + proving
- Emits an **event stream**:
  - `RUN_STARTED`, `PHASE_CHANGED`, `CHECKPOINT_COMMITTED`, `PROOF_PROGRESS`, `RUN_FINISHED`
- Writes the final `capsulepack.tgz`

Relay/API (server-side)

- Receives events (WebSocket)
- Persists them for replay
- Optionally recomputes/validates hashes on received artifacts (sanity)

- When finished, runs the full `verify_capsule.py`-style verification and stamps the run **Verified / Certified / DA-Audited**

UI

- Live timeline + metrics + checkpoint roots
- “Trust indicator” that stays yellow until final verification turns it green
- A “Download capsulepack” button at the end

How to tie it cleanly into your capsule model

You already have the right conceptual anchors:

- **TraceSpec + policy + anchors** are known at time 0 → UI can display them immediately.
- **Commitments** can be emitted as they’re produced:
  - chunk roots, chunk indices, archive pointers
- **DA sampling** can even be shown as a live “audit” view:
  - server picks sample indices deterministically after capsule hash exists (end), *or*
  - if you want a live feel, you can do “preview sampling” during the run but only count the final audit after sealing.

Important: **don’t pretend the live view is the proof.** It’s just “watching the sausage get made.”

What the adopter has to do to support this

If you ship `capsule-bench`, adopters don’t build anything. They just run it.

If a zkVM team integrates via your Trace Adapter, the only extra requirement is:

- implement a callback hook like `on_progress(event)` (or emit events to stdout in a known JSONL format)
- optionally expose “checkpoints” (every k steps, emit chunk roots / partial commitments)

That’s it.

Watch-outs (so you don’t accidentally undermine trust)

- **Don’t stream raw trace rows** by default (too big + privacy + easy to misinterpret).
- Treat streamed metrics as **self-reported telemetry** until the run is sealed and verified.
- If you stream checkpoint roots, label them as “intermediate commitments” (nice to watch, but only final root is what the statement binds).

If you want the “painfully straightforward” product angle

“We’re the first benchmark leaderboard where you can literally watch the proof being born, and then the verifier seals it.”

If you want, I’ll sketch:

- the exact event schema (JSON) for the WebSocket stream,
- what checkpoints you should emit (minimal set),
- and how to map it onto your current capsule fields without redesigning the cryptography.

Absolutely. Here’s a **drop-in, exact event schema** you can stream over WebSocket (or SSE), the **minimal checkpoints** that give “I can literally watch the proof being born,” and the **1:1 mapping** into your existing capsule fields — no crypto redesign.

---

1) WebSocket event schema (JSON)

Envelope (every message)

Every event shares the same outer shape so the UI + server can be dumb and reliable.

```
{
  "schema": "bef_capsule_stream_v1",
  "v": 1,
  "run_id": "run_2025_12_18T14_33_12Z_7b3c",
  "seq": 42,
  "ts_ms": 1766077992123,
  "source": "runner",
  "type": "phase",
  "data": {}
}
```

Envelope fields

- `schema`: constant string
- `v`: integer version
- `run_id`: unique per run (used for reconnect/resume)
- `seq`: strictly increasing sequence number (server can detect drops)
- `ts_ms`: event time in ms
- `source`: `"runner"` or `"server"`
- `type`: one of the event types below
- `data`: event-specific payload

Core event types (runner → server)

A) `run_started`

Sent once, first.


```
{
  "schema": "bef_capsule_stream_v1",
  "v": 1,
  "run_id": "run_...",
  "seq": 1,
  "ts_ms": 1766077900000,
  "source": "runner",
  "type": "run_started",
  "data": {
    "client": {
      "runner_name": "capsule-bench",
      "runner_version": "0.3.0",
      "backend_id": "GEOM_AIR_V1",
      "backend_version": "1.0.0"
    }
  }
}
```

B) `phase`

Used for UX clarity; lightweight and frequent.

```
{
  "type": "phase",
  "data": {
    "phase_id": "trace_generate",
    "phase_label": "Generating trace",
    "progress": { "done": 1200, "total": 2000000, "unit": "steps" }
  }
}
```

Recommended `phase_id` values:

- `init`
- `trace_generate`
- `trace_encode`
- `commit_chunks`
- `finalize_row_root`
- `build_statement`
- `prove`
- `verify_local`
- `pack`
- `seal_capsule`
- `done`

C) `spec_locked` ✅ (first *cryptographically meaningful* checkpoint)

Emitted when TraceSpec/policy/anchors are finalized (before proving).

```
{
  "type": "spec_locked",
  "data": {
    "trace_spec": {
      "spec_version": "1.0",
      "trace_format_id": "GEOM_AIR_V1",
      "record_schema_ref": "sha256:8a1f...",
      "encoding_id": "stc_row_v1",
      "field_modulus_id": "goldilocks_61"
    },
    "trace_spec_hash_hex": "1c9f...ab",
    "policy_id": "bef_bench_policy_v1",
    "policy_version": "1.0.0",
```

```
    "policy_hash_hex": "5e2a...91",
    "anchors": {
      "workload_id": "merkle_build_open_v1",
      "track_id": "baseline_no_accel",
      "runner_repo": "github:org/capsule-bench",
      "git_commit": "8c3d1f4",
      "docker_image_digest": "sha256:1a2b...",
      "hardware_manifest_hash": "sha256:9d4e...",
      "toolchain_hash": "sha256:aa00...",
      "os_fingerprint_hash": "sha256:bb11...",
      "workload_input_hash": "sha256:cc22..."
    }
  }
}
```

D) `chunk_committed` ✅ (minimal live "receipt being born" timeline)

Emitted every time you commit a chunk root.

```
{
  "type": "chunk_committed",
  "data": {
    "chunk_index": 17,
    "chunk_len": 1024,
    "chunk_root_hex": "a3f2...9b",
    "chunk_bytes": 65536,
    "num_chunks_estimate": 2048
  }
}
```

Optional but useful:

- `chunk_handle` if you have a retrieval handle URI
- `chunk_root_proof_path` if you already compute inclusion path (usually later)

E) `chunk_roots_digest` ✅ (locks the list of chunk roots)

Emitted once when the full chunk-roots list is finalized and digested.

```
{
  "type": "chunk_roots_digest",
  "data": {
    "num_chunks": 2048,
    "chunk_roots_digest_hex": "9b72...e1",
    "chunk_roots_format": "bin",
    "chunk_roots_bytes": 65536
  }
}
```

F) `row_root_finalized` ✅ (this is the commitment the statement binds)

Emitted once.


```
{
  "type": "row_root_finalized",
  "data": {
    "row_root_hex": "44aa...10",
    "arity": 16,
    "chunk_len": 1024,
    "row_index_ref": {
      "scheme": "stc_merkle_v1",
      "commitment": "44aa...10",
      "arity": 16,
      "chunk_len": 1024,
      "archive_ref": "file:row_archive/"
    },
    "chunk_meta": {
      "num_chunks": 2048,
      "chunk_len": 1024
    }
  }
}
```

G) `statement_locked` ✅ (the thing injected into the transcript)

Emitted when StatementV1 + statement_hash are computed, before `prove`.


```
{
  "type": "statement_locked",
```

```
  "data": {
    "statement": {
      "statement_version": "1.0",
      "trace_spec_hash_hex": "1c9f...ab",
      "policy_hash_hex": "5e2a...91",
      "trace_root_hex": "44aa...10",
      "public_inputs": {
        "result_digest_hex": "0f9d...aa",
        "workload_size_params": { "leaves_pow2": 20, "openings": 1024 }
      },
      "anchors": { "workload_id": "merkle_build_open_v1", "track_id": "baseline_no_accel" }
    },
    "statement_hash_hex": "e88c...02"
  }
}
```

H) `prove_progress`

Periodic.

```
{
  "type": "prove_progress",
  "data": {
    "stage": "fri_layers",
    "progress": { "done": 12, "total": 24, "unit": "layers" },
    "metrics": {
      "rss_mb": 8421,
      "gpu_util_pct": 91,
      "cpu_util_pct": 65,
      "throughput_items_per_sec": 183000
    }
  }
}
```

I) `proof_artifact` ✅ (content-addressed proof output)

Emitted when proof file is written and hashed.

```
{
  "type": "proof_artifact",
```

```
  "data": {
    "proof_id": "geom",
    "format": "bin",
    "path": "proofs/geom.proof.bin",
    "size_bytes": 733271,
    "sha256_payload_hash_hex": "c1a5...7d",
    "vk_id": "geom_vk_v1",
    "stats": {
      "prover_time_ms": 53210,
      "verify_time_ms": 18,
      "peak_rss_mb": 9120
    }
  }
}
```

J) `capsule_sealed` ✅ (final object hash)

Emitted once when capsule JSON is complete and `capsule_hash` is computed.

```
{
  "type": "capsule_sealed",
  "data": {
    "capsule_path": "capsule.json",
    "capsule_hash_hex": "7d3e...99"
  }
}
```

K) `signature_attached` (only for certified track)

```
{
  "type": "signature_attached",
  "data": {
    "scheme": "secp256k1_recoverable",
    "pubkey_hex": "02ab...ff",
    "signature_hex": "1b9c...88"
  }
}
```

L) `pack_ready`

```
{
  "type": "pack_ready",
  "data": {
    "pack_path": "out/capsulepack.tgz",
    "pack_sha256_hex": "aa77...00",
    "bytes": 19827312
  }
}
```

Server → runner events (optional but powerful)

M) `server_received`

```
{
  "source": "server",
  "type": "server_received",
  "data": { "last_seq": 88 }
}
```

N) `server_verified`

```
{
  "source": "server",
  "type": "server_verified",
  "data": {
    "status": "verified",
    "verify_report": {
      "capsule_hash_ok": true,
      "trace_spec_hash_ok": true,
      "statement_hash_ok": true,
      "proof_ok": true,
      "acl_ok": true,
      "policy_inclusion_ok": true
    }
  }
}
```

O) `da_audit_result`

```
{
  "source": "server",
  "type": "da_audit_result",
  "data": {
    "verification_level": "probabilistic_da_sampling",
    "k_samples": 64,
    "sample_indices": [3, 91, 144, 1000],
    "passed": true
  }
}
```


---

2) Minimal checkpoints to emit (the set that matters)

If you want the smallest set that still feels “real-time computation + cryptographic inevitability,” emit just these:

1. **`spec_locked`**
   Locks TraceSpec + policy hash + anchors.
2. **`chunk_committed`** (repeated)
   Shows the commitment timeline growing.
3. **`row_root_finalized`**
   This is the trace root that everything binds to.
4. **`statement_locked`**
   This is what gets injected into the prover transcript.
5. **`proof_artifact`**
   Proof file exists, is hashed, and has sizes + measured stats.
6. **`capsule_sealed`**
   Capsule hash computed (final “receipt id”).
7. **`signature_attached`** (optional, certified track)

Everything else (`phase`, `prove_progress`) is nice UX but not essential.

---

3) Mapping events → your current capsule fields (no crypto redesign)

This is the clean "wiring diagram": every checkpoint event corresponds to fields you already have.

`spec_locked` → capsule fields

- `data.trace_spec` → `capsule.trace_spec`
- `data.trace_spec_hash_hex` → `capsule.trace_spec_hash`
- `data.policy_id` / `policy_version` / `policy_hash_hex` → `capsule.policy.{policy_id, policy_version, policy_hash}`
- `data.anchors` → `capsule.statement.anchors` **and** (optionally) mirrored at top-level `capsule.anchor_ref` if you use that pattern

Important: This doesn't change cryptography. It just makes explicit that TraceSpec/policy/anchors are determined *before* proving.

`chunk_committed` / `chunk_roots_digest` / `row_root_finalized` → capsule fields

- chunk roots become your `row_archive/chunk_roots.(bin|json)`
- `chunk_roots_digest_hex` → `capsule.proofs.geom.row_archive.chunk_roots_digest`
- `row_root_hex` → `capsule.statement.trace_root` (and `capsule.row_index_ref.commitment`)
- `arity`, `chunk_len`, `num_chunks` → `capsule.chunk_meta` + `capsule.row_index_ref`

This is exactly what your verifier already cross-checks (root/len/arity consistency). You're just streaming it as it happens.

`statement_locked` → capsule fields (and transcript)

- `data.statement` → `capsule.statement`
- `data.statement_hash_hex` → `capsule.statement_hash`
- Prover uses that same `statement_hash` as transcript binding input (unchanged)

This is the "anti-relabel" core. The UI can show it immediately once computed.

`proof_artifact` → capsule fields

- `proof_id "geom"` → `capsule.proofs.geom`
- `sha256_payload_hash_hex` → `capsule.proofs.geom.formats[bin].sha256_payload_hash` (or your existing location)
- `size_bytes` → `capsule.proofs.geom.formats[bin].size_bytes`
- `stats.*` → `capsule.proofs.geom.stats` (or `public_inputs` if you prefer; I'd keep measured stats next to the proof, and keep "result digest + params" in public_inputs)

No crypto redesign — you're already hashing payloads and storing sizes.

`capsule_sealed` → capsule fields

- `capsule_hash_hex` → `capsule.capsule_hash`

Your verifier already recomputes this by removing the field and hashing canonical form. Streaming it is just UX.

`signature_attached` → capsule fields

- attach `capsule.authorship = {scheme, pubkey, signature}`

Then the server can enforce `ACL` and "Certified track" rules using what you already designed.

Implementation note: make drops/reconnect painless

Because the runner emits `seq`, the server can support:

- resume: “send me from seq N”
- or runner can periodically emit a `snapshot` event (optional) that contains the current state (trace_spec + policy + last chunk root + etc.)

If you want, I can also give you:

- a **single “state snapshot” event** format for reconnect,
- and a “capsulepack layout” that includes `events.jsonl` so anyone can replay the UI timeline from the pack itself (huge for credibility).

Got you. Here are the two pieces, written so you can implement them straight away.

---

1) Single “state snapshot” event (for reconnect/resume)

When to send it

- Runner sends `state_snapshot`:
  - on connect (immediately after `run_started`), **or**
  - whenever the server requests it, **or**
  - periodically (e.g., every 2–5 seconds) if you want robust recovery.

What it must do

It must let the server/UI reconstruct the *current run state* **without replaying all past events**, and then continue from `last_seq`.

Snapshot event schema


```
{
  "schema": "bef_capsule_stream_v1",
```

```
"v": 1,
"run_id": "run_2025_12_18T14_33_12Z_7b3c",
"seq": 120,
"ts_ms": 1766078123456,
"source": "runner",
"type": "state_snapshot",
"data": {
  "last_event_seq": 120,

  "phase": {
    "phase_id": "prove",
    "phase_label": "Proving",
    "progress": { "done": 12, "total": 24, "unit": "fri_layers" }
  },

  "locked": {
    "trace_spec_hash_hex": "1c9f...ab",
    "policy_hash_hex": "5e2a...91",
    "statement_hash_hex": "e88c...02",
    "row_root_hex": "44aa...10",
    "capsule_hash_hex": null
  },

  "trace_spec": {
    "spec_version": "1.0",
    "trace_format_id": "GEOM_AIR_V1",
    "record_schema_ref": "sha256:8a1f...",
    "encoding_id": "stc_row_v1",
    "field_modulus_id": "goldilocks_61"
  },

  "policy": {
    "policy_id": "bef_bench_policy_v1",
    "policy_version": "1.0.0"
  },

  "anchors": {
    "workload_id": "merkle_build_open_v1",
    "track_id": "baseline_no_accel",
    "runner_repo": "github:org/capsule-bench",
    "git_commit": "8c3d1f4",
    "docker_image_digest": "sha256:1a2b...",
    "hardware_manifest_hash": "sha256:9d4e...",
    "toolchain_hash": "sha256:aa00...",
```

```
    "os_fingerprint_hash": "sha256:bb11...",
    "workload_input_hash": "sha256:cc22..."
   },

   "commitment_state": {
    "chunk_len": 1024,
    "arity": 16,
    "num_chunks_estimate": 2048,
    "num_chunks_committed": 311,
    "last_chunk_index": 310,
    "last_chunk_root_hex": "a3f2...9b",
    "chunk_roots_digest_hex": null
   },

   "proof_state": {
    "proof_id": "geom",
    "format": "bin",
    "path": null,
    "size_bytes": null,
    "sha256_payload_hash_hex": null,
    "prover_stats": {
     "prover_time_ms_so_far": 21123,
     "rss_mb": 8421,
     "gpu_util_pct": 91,
     "cpu_util_pct": 65
    }
   },

   "local_verify_state": {
    "attempted": false,
    "passed": null,
    "error_code": null
   },

   "authorship": {
    "enabled": false,
    "pubkey_hex": null
   }
  }
}
```


Server-side behavior (simple + robust)

- Server keeps `last_seq_received` per `run_id`.
- On reconnect, server sends:

{ "source":"server", "type":"resume_request", "data": { "from_seq": 87, "want_snapshot": true } }

- Runner responds with:
  - a `state_snapshot`, then
  - subsequent events starting at `seq = from_seq + 1` (or just continue monotonic; server can ignore duplicates as long as it has seq tracking).

Why this works

- Snapshot tells the UI **where you are** (phase + progress).
- Snapshot tells the server **what is already locked** (trace_spec/policy/statement/row root).
- Snapshot tells the server what's already committed (chunk count + last root).
- Snapshot avoids replay storms.

---

2) Capsulepack layout with `events.jsonl` (replayable timeline)

This is *exactly* the "huge for credibility" move. Anyone can:

- unpack the capsulepack
- open the timeline file
- replay the run in a UI deterministically
- and then run `verify_capsule.py` to confirm it sealed correctly

Canonical `capsulepack` directory structure

```
capsulepack/
  capsule.json
  policy.json
  policy_inclusion_proof.json        (optional, certified track)
  policy_registry_root.json          (optional, or server-known)
  proofs/
    geom.proof.bin
    geom.proof.json                  (optional alternate format)
```

```
  geom.vk.bin                    (optional)
 row_archive/
  chunk_roots.bin
  chunk_roots.json               (optional)
  chunk_roots_digest.txt
  chunks/                        (optional: only if bundling data)
    chunk_000000.bin
    chunk_000001.bin
    ...
  samples/                       (recommended minimal DA bundle)
    sample_indices.json
    chunk_000091.bin
    chunk_001440.bin
    ...
 manifests/
  hardware_manifest.json
  os_fingerprint.json
  toolchain_manifest.json
  env_manifest.json              (optional: env vars, driver info)
 measurements/
  measurements.json              (times/memory + method)
 events/
  events.jsonl                   (append-only event log)
  events.sha256                  (sha256 of events.jsonl)
  events.meta.json               (optional: rotation info)
 logs/
  runner.log                     (optional)
 pack_meta.json
```

What goes into `pack_meta.json`

This ties everything together and makes packs self-describing.

```
{
  "schema": "bef_capsulepack_v1",
  "v": 1,
  "run_id": "run_2025_12_18T14_33_12Z_7b3c",
  "created_ts_ms": 1766078129999,
  "capsule_hash_hex": "7d3e...99",
  "policy_hash_hex": "5e2a...91",
  "statement_hash_hex": "e88c...02",
  "trace_spec_hash_hex": "1c9f...ab",
  "contents": {
```

```
    "events_jsonl": "events/events.jsonl",
    "proof_primary": "proofs/geom.proof.bin",
    "row_archive": "row_archive/",
    "manifests": "manifests/",
    "measurements": "measurements/measurements.json"
  }
}
```

`events.jsonl` format (append-only)

Each line is exactly one event, identical to the WS schema you already stream.

Example lines:

{"schema":"bef_capsule_stream_v1","v":1,"run_id":"run_...","seq":1,"ts_ms":...,"source":"runner","type":"run_started","data":{...}}
{"schema":"bef_capsule_stream_v1","v":1,"run_id":"run_...","seq":2,"ts_ms":...,"source":"runner","type":"spec_locked","data":{...}}
{"schema":"bef_capsule_stream_v1","v":1,"run_id":"run_...","seq":3,"ts_ms":...,"source":"runner","type":"chunk_committed","data":{...}}
...
{"schema":"bef_capsule_stream_v1","v":1,"run_id":"run_...","seq":88,"ts_ms":...,"source":"runner","type":"capsule_sealed","data":{...}}

Minimal events to guarantee a meaningful replay

For replay credibility, ensure `events.jsonl` always contains at least:

- `run_started`
- `spec_locked`
- every `chunk_committed` **or** (if too large) periodic `chunk_committed` + a final `chunk_roots_digest` + `row_root_finalized`
- `statement_locked`
- `proof_artifact`
- `capsule_sealed`
- optional `signature_attached`

How to prevent `events.jsonl` from being “editable theater”

Two clean options:

Option A (simple, good enough): hash it

- Write `events/events.sha256` = sha256(events.jsonl)
- Include that hash inside the capsule as an **anchor**:
  - `anchors.events_hash = "sha256:<...>"`
    This means the *proof* binds to the event log hash (because anchors are bound in StatementV1 → statement_hash → transcript).

No crypto redesign; just “anchor the log hash.”

Option B (stronger): rolling hash chain

Each event includes:

- `prev_event_hash_hex`
- `event_hash_hex = sha256(prev_hash || canonical_event_bytes)`

Still no redesign: it’s purely in telemetry, and you can store the final chain head in `anchors.event_chain_head`.

I’d start with Option A. It’s clean and sufficient.

Putting it together: “replay + verify” story

A third party can:

1. unpack `capsulepack.tgz`
2. load `events/events.jsonl` and animate the run
3. check `sha256(events.jsonl)` matches `anchors.events_hash`
4. run verifier on `capsule.json` + proof artifacts
5. see the run seal into a verified capsule hash

That is **painfully straightforward credibility**.

If you want, I can also write:

- the exact **`events_hash` anchor naming convention** and where it lives in your current `StatementV1` object,
- and a “rotation strategy” (e.g., `events_000.jsonl`, `events_001.jsonl`) so gigantic runs don’t create massive single files.