Yeah, my bad — you're right to call that out. I just re-opened the **actual zip** and here's what's *really* going on, end-to-end, based on the code that's in there.

What this zip actually contains

It's a **capsule format + (demo) pipeline harness** for one concrete VM/AIR ("geom") that shows how you want capsules to be built and verified.

What's included:

- `scripts/run_geom_pipeline.py` — **producer**: runs Geometry → STC → STARK proof → (optional) Nova recursion → writes a `bef_capsule_v1` JSON/BIN capsule + an artifact manifest + row archive.
- `scripts/verify_capsule.py` — **verifier**: "accept/reject with stable error codes", verifying *hashes, statement-binding, proof validity, policy registry membership, signature/ACL, and DA sampling audit*.
- `bef_zk/spec/{trace.py,statement.py}` — defines your two key *adoption surface primitives*:
  - **TraceSpecV1** + `trace_spec_hash`
  - **StatementV1** + `statement_hash`
- `trace_statement_spec.md` + `README.md` — explains the intention.

What's **not** included (but referenced heavily): the actual `bef_zk` implementation (codec, DA providers, STC VC, AIR, prover/verifier, error codes, etc.). So the zip is a *"capsule interface + geom demo driver"* snapshot, not the whole repo.

The real architecture (as implemented)

1) Producer pipeline: `run_geom_pipeline.py`

This script does five big things:

A) Simulate a trace

- `simulate_trace(GEOM_PROGRAM, params, init)` produces rows.
- It also derives a **column schema**, hashes it, and sticks `sha256:<schema_hash>` into `record_schema_ref`.

B) Create TraceSpecV1

```
TraceSpecV1(

  spec_version="1.0",

  trace_format_id="GEOM_AIR_V1",

  record_schema_ref="sha256:<schema_hash>",

  encoding_id=<selected_encoding>,

  field_modulus_id="goldilocks_61",

)

trace_spec_hash = compute_trace_spec_hash(trace_spec)
```

So: the trace isn't "just bytes." It's *typed* by a content-addressed schema reference.

C) Prove (STARK) with statement binding
During proving, you build a `StatementV1` inside `_build_statement_hash(...)`, and critically:

- `policy_hash = sha256(policy_file_bytes)`
- `trace_root = row_commitment.params["root"]` (Merkle root over chunk roots)
- `public_inputs` are concrete geom values (`final_m11`, `final_m12`, etc.)
- `anchors` optionally include an external anchor ref (L1 block hash, dataset hash, whatever)

Then `statement_hash` is computed and **fed into the transcript** via `statement_hash_fn`.

That's the "no proof reuse under a different meaning" property you want.

D) Export STC row archive + chunk roots

- It extracts `chunk_handles` and `chunk_roots_hex` from proof metadata, then persists:
  - `row_archive/chunk_roots.json`
  - `row_archive/chunk_roots.bin`
  - plus `chunk_roots_digest` (sha256 over the `.bin`)
- It constructs:
  - `chunk_meta` (num_chunks, chunk_len, sizes)
  - `row_index_ref` (commitment root, arity, pointer to archive)

E) Emit a capsule
It emits `bef_capsule_v1` containing:

- `trace_spec` and `trace_spec_hash`
- `statement` and `statement_hash`
- `policy: {policy_id, policy_version, policy_path, policy_hash}`
- `da_policy`:
  - `verification_level: "probabilistic_da_sampling"`
  - provider = local filesystem archive root
  - `k_samples` defaults to `ceil(log(1/epsilon)/delta)` with hardcoded `(delta=0.1, epsilon=1e-6)`
- `proofs.geom`:
  - proof paths, sizes
  - `row_archive` metadata
  - `formats` (json/bin with `sha256_payload_hash` per format)
- optional `proofs.nova` block and `trace_commitment = nova_state`

Finally:

- computes `capsule_hash = compute_capsule_hash(capsule_without_capsule_hash)`
- optionally adds `authorship` signature (secp256k1 recoverable sig over capsule hash)

---

2) Verifier pipeline: `verify_capsule.py`

This is honestly the most “product-like” piece in the zip. It’s a strict accept/reject verifier with stable error codes.

It verifies in layers:

(1) Parse + schema check

- Must be dict
- `schema == "bef_capsule_v1"`

(2) Capsule hash integrity

- Requires `capsule_hash` field.
- Recomputes canonical hash after removing `capsule_hash`.
- Mismatch ⇒ reject.

(3) TraceSpec integrity

- Requires `trace_spec` object AND `trace_spec_hash`.
- Recomputes `compute_trace_spec_hash(TraceSpecV1.from_obj(...))`
- Any mismatch ⇒ `E053_PROOF_STATEMENT_MISMATCH`

(4) Optional policy registry pinning
If you provide `--policy`, `--policy-inclusion-proof`, and `--policy-registry-root`:

- checks `sha256(policy_file) == capsule.policy.policy_hash`
- verifies a Merkle proof that the policy hash is included under the trusted registry root

So you support both "self-asserted policy hash" and "policy hash pinned to a governance root."

(5) Proof artifact integrity

- Locates proof path from capsule.
- If the capsule includes `sha256_payload_hash` for that proof format, it recomputes payload hash (with a header-skip rule) and checks it.

(6) Binding: proof row commitment must match capsule metadata
This is that anti-tamper invariant you were talking about:

- compares root / chunk_len / arity between:
    - proof's internal row_commitment params
    - capsule's `chunk_meta` + `row_index_ref`

Mismatch ⇒ `E053_PROOF_STATEMENT_MISMATCH`

(7) Statement integrity + cross-binding

- Requires `statement` object and `statement_hash`.
- Recomputes statement hash, checks it matches.
- Also checks:
    - statement.trace_spec_hash equals capsule.trace_spec_hash
    - statement.policy_hash equals capsule policy hash
    - statement.trace_root equals capsule row_index_ref.commitment
    - anchors match capsule anchor (if present)

Then it calls `zk_verify_geom(..., statement_hash=bytes.fromhex(statement_hash))`.

So the verifier is forcing the **same statement hash** to be used in the STARK verifier.

(8) Optional Nova consistency check
If `nova_stats` exists, it checks:

- `nova_stats["nova_state"] == capsule["trace_commitment"]`
  Mismatch ⇒ `E055_NOVA_STATE_MISMATCH`

(9) Row index commitment verification
If row archive info exists:

- loads chunk_roots from inline/json/bin
- reconstructs k-ary Merkle root
- checks equals `row_index_ref.commitment`

So even before any DA sampling, it checks the row index root is consistent with the chunk roots list.

(10) Authorship + ACL

- If ACL exists, authorship becomes required.
- It verifies recoverable secp256k1 signature recovers claimed pubkey.
- Then checks signer pubkey is authorized for the policy_id (status active).

(11) DA audit (probabilistic sampling)
If `da_policy` is present:

- deterministically samples indices using a seed derived from:
  - capsule_hash
  - anchor_ref
  - policy_id, policy_version
- fetches sampled chunks from provider
- recomputes each sampled chunk root and verifies its Merkle inclusion proof up to the row root

This is why you call it "probabilistic_da_sampling."

What I think — where this stands overall

Based purely on what's in the zip:

This is strong as a *capsule contract*

You've got a coherent object model:

- **TraceSpec** = "what does this trace mean?"
- **Statement** = "what claim are we proving, bound to policy/root/inputs/anchors?"
- **Capsule hash** = "this entire bundle is integrity-protected"
- **Proof payload hashes** = "artifacts are content-addressed too"
- **Policy registry inclusion** = "governance pinning is possible"
- **Authorship/ACL** = "who's allowed to produce capsules under this policy"
- **DA sampling** = "retrievability checks are enforceable"

That's a *real* systems design. It's not vibes.

But it's still a demo-shaped system

Because:

- Only one AIR (`GEOM_AIR_V1`) is concretely wired.
- The "adoption surface" for *other traces* is implied, not implemented in this snapshot.
- DA is implemented as **sampling + local archive provider**, not a full general DA layer story.

My honest placement

- **"Proof object / verification contract" maturity:** high (you're doing the right bindings and checks)
- **"General-purpose platform / others can adopt without you" maturity:** medium-low (Trace Adapter API is the missing headliner)
- **"Market wedge clarity" maturity:** medium (you've built the primitive; wedge depends on packaging and target)

If you want the cleanest one-liner for what this is **today**:

"A policy-bound proof capsule format + verifier that prevents proof reuse under different semantics and supports deterministic DA sampling audits."

That's already a *category*.

The 3 biggest "what's missing next" signals *from the zip itself*

These are not generic advice — they fall straight out of the code:

1. **Trace Adapter API (first-class)**
   Right now, the pipeline hardcodes how to:

- define schema
- define TraceSpec
- define public inputs
- define Statement
- invoke prover/verifier

To make it adoptable, you need a standard interface like:

- `TraceAdapter.build_trace_spec()`
- `TraceAdapter.extract_public_inputs()`
- `TraceAdapter.statement_from_commitment(...)`
- `TraceAdapter.program_id / air_id / vm_id`

2. **Policy semantics need teeth**
   You already hash the policy file and optionally pin it to a registry — great.
   But "policy" will only matter if you define the **minimum required checks** that a verifier must enforce (beyond inclusion).

Right now policy is mostly "bound," not "interpreted," except via ACL and DA parameters.

3. **DA story: name it precisely**
   Your implementation is: deterministic sampling of chunks from a known archive + Merkle proof checking.
   That's valuable, but it's not "all DA problems solved." It's **audited retrievability** under a sampling policy.

That's fine — it's actually a wedge if you position it right.

If you want, I can do the next step in the same "look at the zip and be concrete" mode:

- I'll write you the **exact wedge options that match what this system already enforces**, not speculative ones.
- And I'll map each wedge to the *fields already in the capsule* (policy_id, anchors, da_policy, ACL), so the pitch sounds inevitable.

Alright. Here are wedge options that are **compatible with what your zip already enforces today** (TraceSpec/Statement binding, policy hash + optional registry, authorship/ACL, deterministic DA sampling, artifact hashing), plus exactly how you'd pitch + implement each **using the fields you already have**.

I'm going to be a little ruthless about "what you can honestly claim *now*" vs what would be overclaiming.

Wedge 1 — Verifiable backtests / “proof-carrying alpha reports”

Who buys first: trading teams, quant pods, prop-ish shops, crypto funds, even internal risk/compliance.
Pain: “Your backtest is a PDF + trust me.” No provenance, no reproducibility, no audit trail, easy to cherry-pick.

What you deliver: a capsule that proves:

- the *exact* backtest code/program variant ran (TraceSpec ties to schema/program id)
- on the *exact* dataset snapshot (anchor)
- under an agreed policy (policy hash + optional registry pinning)
- and the raw trace/dataset chunks are retrievable with a quantifiable sampling guarantee (da_policy)

How your current capsule fields map

- `anchors`: use `dataset_hash`, `data_vendor_attestation`, `universe_hash`, `time_range`, or even “exchange snapshot hash”
- `policy_hash`: encodes “no lookahead, slippage model, fees model, train/test split rules, max leverage, etc.”
- `policy_id` / `policy_version`: “strategy_eval_policy/v3”
- `public_inputs`: final metrics (CAGR, Sharpe, max drawdown, turnover) + key intermediate checkpoints
- `authorship` + `ACL`: only approved “research runners” can produce capsules that count
- `da_policy.verification_level="probabilistic_da_sampling"`: “audited retrievability” of the raw trace or row archive

Why this is a good wedge *for your current system*
You don’t need “L1 rollup DA.” You need “audited reproducibility.” Your DA sampling is *already enough* to say: “if you challenge me, I can produce the underlying rows/chunks, and you can verify inclusion to the committed root.”

Skeptic pushback you’ll get (and how to answer)

- “Sampling isn’t full availability.” ✅ Correct — don’t oversell. Call it **retrievability audit** with configurable (δ, ε) policy.
- “What if you cherry-pick a dataset?” That’s exactly why `anchors` exists. Make dataset anchoring non-optional in the policy.

Wedge 2 — zkML evaluation receipts (model claims that can't be faked)

Who buys first: ML infra teams, model marketplaces, "model provider" startups, AI safety/compliance groups.
Pain: benchmark claims are easy to lie about; provenance is weak; dataset leakage is rampant.

What you deliver: "This model achieved X on dataset Y under eval policy Z, and here's a cryptographic artifact that binds it."

Field mapping

- `anchors`: `dataset_hash`, `benchmark_suite_version`, `promptset_hash`, `model_weights_hash`
- `policy_hash`: evaluation policy (no internet, deterministic seeds, allowed preprocessing, max context length, etc.)
- `public_inputs`: metrics + confusion matrices / aggregate stats
- `TraceSpecV1`: schema reference for trace rows (e.g., per-example logits hashes / loss summaries)
- `DA sampling`: lets an auditor spot-check example-level trace chunks without you dumping everything publicly

Why it fits
Your design is basically a "receipt object" with stronger policy semantics than typical zk receipts.

Hard truth / constraint
This wedge is *only credible* if you can make the trace adapter story clean for "any evaluation pipeline." Right now zip is geom-shaped. But this wedge doesn't require new cryptography — it requires better adapter UX.

Wedge 3 — "Policy-bound compute attestations" for regulated workflows

Who buys first: fintech, enterprise, gov-ish orgs, internal audit teams.
Pain: proving you followed process controls ("we ran the approved pipeline on approved data") is expensive and mostly paperwork.

What you deliver: a capsule that *is* the audit artifact: hash-bound policy + authorized signer + anchored inputs + retrievability checks.

Field mapping

- `policy_registry_root + inclusion_proof`: this becomes huge here — it's "the policy was approved by governance."
- `authorship/ACL`: policy-bound signing authority ("only these keys can attest")
- `anchors`: change-ticket id, build artifact hash, config hash, dataset snapshot hash
- `statement_hash` in transcript: stops replaying an old proof under a new "approved" policy

Why it's real
Unlike "build a rollup," this wedge sells into people who already pay for audit/compliance, and you give them cryptographic receipts instead of screenshots.

Skeptic pushback

- "We can do this with Sigstore / attestations." Yes — so the differentiation must be: **you're attesting to the computation's correctness, not just that someone signed a build.** Your statement/proof binding is the value.

Wedge 4 — "Capsules as verifiable research artifacts" for multi-party collaboration

Who buys first: any team collaborating across trust boundaries (joint research, open competitions, bounty-based R&D, distributed teams).
Pain: people don't trust each other's experiment claims; reproducibility is painful.

What you deliver: a capsule as the unit of collaboration:

- one object to share
- deterministic verification
- stable error codes
- can be pinned to a registry or signed

Field mapping

- `artifact_manifest.json`: content-addressed index of "what files does this capsule refer to"
- `anchors`: commit hash, repo tag, docker image hash
- `policy_hash`: "how experiments must be run"

- `authorship`: identities

Why it's a good "fast demo wedge"
You can ship this without integrating into chains or DA networks. It's a software distribution wedge.

Wedge 5 — "Bridge / oracle-grade provenance layer" (careful: only if you narrow scope)

Who buys first: oracle teams, bridge teams, on-chain protocols that need trustworthy off-chain computation summaries.
Pain: on-chain contracts depend on off-chain processes (indexing, pricing, risk) with weak attestations.

What you can honestly do now
You can produce a capsule proving "off-chain computation X produced value Y under policy Z and anchored to block B."

Field mapping

- `anchors`: L1 block hash / finalized checkpoint
- `public_inputs`: the oracle values being posted on-chain
- `policy_hash`: exactly what computation is allowed
- `authorship/ACL`: only authorized operators can produce capsules accepted by the contract

What you should NOT overclaim yet
This does not automatically become a decentralized oracle network. It becomes **auditable operator attestation** with cryptographic correctness and retrievability audit.

Wedge 6 — "Proof-carrying benchmarks" for ZK / systems performance

Who buys first: ZK infra companies, provers, hardware teams, researchers.
Pain: performance claims ("we proved X in Y ms") are unverifiable and cherry-picked.

What you deliver
A capsule whose `public_inputs` include benchmark results, and whose trace proves the benchmark workload ran as specified.

Field mapping

- `policy_hash`: benchmark rules
- `TraceSpec`: schema of performance trace
- `anchors`: hardware spec hash / environment hash

This is a surprisingly good wedge if you want credibility fast in ZK circles — but it's not usually a big business.

Which wedge is "best" given what you have right now?

If we're being brutally practical:

Best immediate wedge (fastest to product):

Wedge 1 (Verifiable backtests) + Wedge 3 (Policy-bound attestations)
Because:

- your policy registry + ACL + statement binding is *already* tailor-made for "compliance-like" claims
- "DA sampling" is plenty for **audited retrievability** (spot checks)
- you don't need to solve global DA or build a chain

Most "platform-y" wedge (bigger upside, slower):

Wedge 2 (zkML eval receipts)
Huge market narrative, but requires the Trace Adapter API to be clean so others can plug in pipelines.

How to pitch the wedge in one sentence (per option)

- **Backtests:** “A cryptographic backtest receipt that binds dataset snapshot + evaluation policy + metrics, and supports spot-checkable retrievability of the underlying run.”
- **zkML eval:** “Verifiable benchmark receipts: model claims you can audit, tied to an exact dataset + eval policy.”
- **Compliance:** “A policy-bound computation attestation that’s governance-pinnable and signer-restricted.”
- **Oracle/bridge:** “Correctness proofs for off-chain compute outputs, anchored to chain state and bound to operator authorization.”

Two concrete GTM experiments you can run *without adding new crypto*

Experiment A — “Challengeable backtest capsule”

1. Pick 1 strategy + 1 dataset snapshot.
2. Write a policy file that forbids lookahead/leakage and requires dataset anchoring.
3. Emit capsule with:
   - `anchors: dataset_hash + time range + vendor`
   - `public_inputs: key metrics`
   - `da_policy` configured to a strong-sounding but honest epsilon/delta
4. Invite someone to challenge: “Ask for chunk i, we’ll produce it, you verify inclusion.”

That demo sells your system better than any theory.

Experiment B — “Governed policy registry”

1. Create a tiny policy registry root (Merkle over allowed policies).
2. Ship capsules that include an inclusion proof to that root.
3. Now you can say: “Only governance-approved evaluation policies are valid.”

That’s the difference between “cool proof” and “enterprise/protocol-ready artifact.”

One last “don’t fool yourselves” note

Your system’s sharp edge is **semantic binding + governance + auditability**.

If you try to wedge as “data availability network” or “rollup competitor” too early, people will compare you to Celestia/EigenDA/etc. and you’ll lose on category mismatch. But if you wedge as **policy-bound proof capsules for off-chain computation**, your current design looks *purpose-built*.

If you tell me which world you want to win first—**finance backtests, zkML evals, or compliance attestations**—I’ll write:

- the exact capsule fields you should require in policy,
- the demo storyline,
- and the 5-slide wedge pitch (no fluff, just inevitability).

Wedge 6 is actually *sneaky good* — not as a “huge revenue day 1” product, but as a **credibility + adoption Trojan horse** that perfectly matches what your capsule system already does well.

The honest premise

Right now, zk performance “benchmarks” are mostly:

- blog posts by teams with incentives (RISC Zero vs SP1, etc.) (RISC Zero)
- some community attempts at standardization (ZK Bench / zk-Harness / zkvm-perf) (ZK Bench)
- occasional broader meta-reports (e.g., Fenbushi’s 2025 benchmarking overview) (Fenbushi)

The missing thing is: **a result format that’s cryptographically bound to “what ran,” under “what rules,” on “what inputs,” with “what artifacts,” and optionally “who is authorized to claim it.”**
That is *literally* what your capsule schema is.

What wedge 6 becomes in your world

“Proof-carrying benchmarks” = a benchmark result that is only accepted if it comes with a capsule that verifies.

Not “trust me bro, we ran it.”
More like: “Here’s the exact workload definition + policy + inputs + proof + artifact hashes + audit hooks.”

This matches the benchmarking standards vibe the ZKProof community has talked about for years: benchmark frameworks and recommended practices, plus qualitative properties/assumptions that need to be declared. (ZKProof)

Why wedge 6 fits your zip *today*

Your current capsule/verifier machinery already supports the essential ingredients:

1. **Workload identity (TraceSpecV1)**
- `trace_format_id`, `record_schema_ref`, `encoding_id`, `field_modulus_id`
- This is your “benchmark definition fingerprint.”
2. **Claim binding (StatementV1 in transcript)**
- binds trace root + public outputs + anchors + policy hash
- stops re-labeling the same proof as a different benchmark.
3. **Rules of the benchmark (policy hash + optional registry inclusion)**
- you can publish “Benchmark Policy vN” and pin it via a registry root.
- now people can’t silently change methodology.
4. **Artifact integrity**
- proof payload hashes let you content-address the proof files.
5. **DA sampling**
- for benchmarks, this is *perfect* as “auditability”: you can spot-check logs/chunks without demanding everyone publish massive traces.
6. **Authorship + ACL**
- lets you do “certified runs”: only keys in the policy ACL can publish “official” benchmark capsules.

The key limitation (and how to position it honestly)

You **cannot** cryptographically prove wall-clock time in a trustless way from pure software proofs alone.

So don’t sell “we prove runtime.” Sell:

✅ What you can prove

- the computation/workload was executed correctly
- under declared constraints (policy)
- with declared inputs/versions (anchors)
- and the artifact set is integrity-protected

- plus you can include measurements *as signed telemetry* and make them auditable/reproducible.

How the ecosystem already handles this

Even existing benchmark frameworks emphasize how many "factors affect performance" and the difficulty of direct comparisons. (ZK Bench)
And tools like zkvm-perf are basically trying to standardize the *process* around running benchmarks. (GitHub)
Your wedge is: **standardize the process + make it cryptographically enforceable.**

The clean "product shape" for wedge 6

If you do wedge 6, the product is basically:

1) A benchmark suite spec

- "workloads" = named TraceSpecs (and/or "program IDs")
- "policies" = methodology: compiler flags, CPU pinning, dataset sizes, warmups, GPU model constraints, etc.
- "required anchors" = git commit hash, docker image digest, prover version, hardware manifest

2) A runner that emits capsules

- wraps existing tools (zkvm-perf, zk-bench style harnesses) but outputs **your capsule** at the end (ZK Bench)

3) A verifier + leaderboard that only accepts verified capsules

- no capsule, no leaderboard entry
- "certified track" requires ACL signing

4) Optional: "certified lab" program

- keys represent labs/hardware providers
- ACL restricts who can publish certified results

How to pick workloads so this wedge lands

You want workloads that match how teams already publish comparisons:

- “real-world workload” style comparisons (SP1 and RISC Zero both benchmark specific workloads and blocks) (Succinct)
- “microbenchmarks” (hashes, FFT/MSM-ish components, recursion steps, etc.)
- “proof system stack” benchmarks (zk-Harness explicitly aims to cover the stack) (GitHub)

The Fenbushi benchmarking article suggests the market is actively paying attention to cross-zkVM performance comparisons right now. (Fenbushi)
So you’d be riding an existing wave, not inventing demand.

Where wedge 6 stands commercially

This is the real trade:

What wedge 6 is ***great*** for

- instant credibility in the ZK ecosystem
- forcing a standard “capsule interface” into people’s workflows
- becoming the neutral layer everyone references

What wedge 6 is ***not*** great for (alone)

- big revenue early, unless you attach it to:
    - certification
    - enterprise procurement (help teams choose provers/hardware)
    - “benchmarking as a service” for teams shipping prover networks / rollups

---

If you want the tightest possible pitch for wedge 6

> “We built a cryptographically verifiable benchmark artifact format. Benchmarks become receipts: workload definition + policy + input anchoring + proof + artifact hashes + optional certified signer. No more unverifiable performance claims.”

…and the subtext is: *once people adopt capsules for benchmarks, adopting capsules for everything else becomes natural.*

Concrete next step (no new cryptography required)

If you say “yes wedge 6,” the fastest win is:

1. Ship `capsule-bench` CLI:
- runs a benchmark workload (even if initially only your GEOM demo + 1–2 microbenchmarks)
- emits capsule + artifacts
- verifies locally
2. Ship a tiny website that:
- uploads capsule
- verifies capsule
- posts result to a leaderboard
3. Add a **policy registry root** and a **certified track**:
- community track: anyone can post
- certified track: must be signed by keys in ACL

That’s enough to be “a thing” people can point at.

If you want, tell me which direction you want wedge 6 to be:

1. **ZKVM leaderboard (neutral ecosystem credibility)**
2. **Enterprise “certified benchmark reports” (procurement/compliance angle)**
3. **Prover-network sales enablement (help teams market performance with receipts)**

…and I’ll write the exact capsule field requirements + the policy schema + the demo storyline you should run with.

Yeah — **1 + 3 is the right combo** *because they reinforce each other*:

- **(1) Neutral leaderboard** gives you legitimacy + attention + a standard everyone references.
- **(3) Sales enablement for prover networks / zkVM teams** turns that legitimacy into distribution + money (teams *want* to market performance claims, but the space is noisy and trust is low).

And the best part: this is already where the ecosystem is moving. There are multiple active benchmark efforts (tooling + repos + reports), but very few “**cryptographically-bound benchmark receipts**” as the unit of truth:

- Tooling exists (e.g., **zkvm-perf**) (GitHub)
- Other benchmark harnesses exist (**zkvm-bench**, **zk-Harness**) (GitHub)

- People are explicitly trying to do standardized comparisons (Fenbushi's 2025 report; Ethereum Foundation zkEVM benchmarking post) (Fenbushi)

The wedge statement

"We make zkVM benchmarks *receipt-based*."
No receipt (capsule) → it doesn't count. And "receipt" doesn't just mean logs; it means **(TraceSpec + Statement + policy hash + artifact hashes + optional signer/ACL + audit hooks)**.

That's exactly what your capsule design already supports.

Product shape for (1) Neutral leaderboard

What you ship

A public leaderboard that only accepts verified capsules.

Tracks

1. **Community track**: anyone can submit a capsule, must verify.
2. **Certified track**: capsule must include `authorship` and pass `ACL` checks for a given `policy_id` (your verifier already supports this pattern).

Why this works
Most benchmarking repos can *run tests*; your differentiator is **a verification gate** and a portable artifact format.

Bench workloads
You don't need 50 benchmarks. You need **5–8** that are:

- widely understandable,
- hard to game,
- representative (CPU-heavy, GPU-heavy, recursion-ish, memory-ish).

You can even *wrap* existing benchmark suites rather than inventing everything:

- zkvm-perf exists specifically as a benchmarking tool for zkVMs (GitHub)
- zkvm-bench provides an "apples-to-apples" comparison setup across multiple zkVMs (GitHub)

Your move is: **output a capsule at the end**.

Product shape for (3) Sales enablement (the money printer)

Here's what zkVM/prover-network teams want *badly*:

- "Here are our numbers."
- "They're reproducible."
- "They're fair."
- "Here's the raw methodology."
- "Here's a badge / proof it's legit."

This is already playing out in the wild: teams publish benchmark claims (SP1 posts benchmark articles; RISC Zero posts "zkVM 1.0" performance claims) (Succinct) — but readers always ask "under what settings, what hardware, what version, what shortcuts?"

What you sell

Verified Benchmark Packs:

- a capsule bundle + signed certification + a shareable "Benchmark Receipt" page
- optional "Run in our lab" service (you control hardware + policy registry)
- "Competitive comparison report" (capsules for multiple systems under identical policy)

Fenbushi's 2025 benchmarking report is basically evidence that *market attention exists* for standardized zkVM comparisons (Fenbushi). You're productizing that into receipts.

How your *existing* capsule fields become the standard (no handwaving)

To make this wedge crisp, define these as **required** in the benchmark policy (so verifiers enforce them):

Required anchors (anti-gaming)

- `git_commit` (benchmark runner + workload)
- `docker_image_digest` (or build hash)
- `hardware_manifest_hash` (CPU/GPU model, RAM, OS/kernel, driver versions)
- `workload_input_hash` (fixed input set)

- optionally `cloud_instance_type` (when running in cloud)

Required public inputs (what shows up on leaderboard)

- proof size
- prover time *as declared measurement* (not cryptographically proven, but policy-governed)
- peak memory
- cycles/steps (workload size parameter)
- verification time

Required policy fields

- warmup rules
- repetitions + aggregation rule (median/mean)
- allowed flags (or explicitly disallow compiler accel/custom params unless stated)
- whether GPU is allowed, which GPU class
- whether recursion is included

Your verifier already binds `policy_hash` into the statement hash, so the proof can't be "relabelled" under a different policy later.

The "credibility lock": Policy registry + Certified Keys

This is how you prevent the leaderboard from becoming a circus:

- Publish a **policy registry root** (Merkle root of approved policies).
- Submissions must include inclusion proofs for "Certified" track.
- Certified runs require signature + ACL authorization per policy.

This is exactly the kind of governance/provenance story serious benchmarking needs, and it maps to what your verifier already supports.

Practical rollout plan (fast)

v0 (2 weeks energy)

- 3 workloads
- community track only

- upload capsule → verify → show results

v1 (adds the wedge "teeth")

- policy registry + certified track
- fixed runner repo + docker image
- "compare two zkVMs under identical policy" template

v2 (turn into distribution + revenue)

- "Verified Benchmark Pack" for zkVM teams (they embed your badge)
- optional hosted runs (your hardware)
- enterprise/protocol-facing reports ("pick a prover network")

The one pitfall to avoid

Don't claim you "prove wall-clock time."
What you *can* claim is stronger and cleaner:

"We make benchmarks reproducible and policy-bound, with cryptographic integrity over the artifacts and a verifier-gated submission format."

That's a very defensible line, and it's aligned with the direction of current benchmarking efforts (standardized harnesses + cross-system comparisons) (GitHub).

If you say "go," I'll write you:

1. a **Benchmark Policy v1 JSON schema** (the exact fields + rules),
2. the **capsule submission contract** (what a run must include),
3. the **leaderboard categories** (so it doesn't devolve into apples vs oranges).

Yeah — **1 + 3 is the cleanest "inevitable" wedge**, and you already have the right primitives in your capsule system to make it real. Let me lay out the *actual* product thesis in a way that's hard to argue with.

The core insight

Benchmarking in ZK is messy for two reasons:

1. **Comparability is structurally hard** (too many knobs; comparisons can unintentionally favor one system). zk-Harness literally starts from this premise: lots of zkSNARK libraries exist, but "various factors make it difficult to compare different proof systems without favoring some approaches over others." (GitHub)
2. The ecosystem *increasingly needs benchmarking as infrastructure*, not as blog posts. The Ethereum Foundation's zkEVM team is explicitly benchmarking zkVMs to evaluate **real-time proving** and worst-case performance, because failing RTP threatens liveness/finality (their framing, not yours). (zkEVM)

So the wedge isn't "we benchmark zkVMs." Lots of repos do. (zkvm-perf, brevis' zkvm-bench, a16z's repo, etc.) (GitHub)
The wedge is:

Benchmarks become "receipts."

A benchmark result is only accepted if it comes with a **verifiable capsule** that binds:

- *what ran* (TraceSpec / program identity)
- *under what rules* (policy hash + optional policy registry inclusion)
- *on what inputs* (anchors)
- *with what artifacts* (proof + payload hashes)
- *who is allowed to claim "certified"* (authorship + ACL)
- *with audit hooks* (your deterministic sampling retrievability check)

This turns benchmarking from "content" into **a protocol**.

Why 1 + 3 works as a pair

(1) Neutral leaderboard = legitimacy + standard adoption

If you become the place where results are posted **only if they verify**, you set the format.

(3) Sales enablement = distribution + money

Teams *want* to publish performance claims, and they also *want* a credible way to say "fair apples-to-apples." Brevis' repo explicitly emphasizes same hardware, same Rust program, and no compiler acceleration/custom params to keep things fair. (GitHub)
You can sell them **"Verified Benchmark Packs"** that:

- produce capsules under a strict policy
- show up on the neutral leaderboard
- come with a shareable "Receipt Page" + badge

The leaderboard makes the receipts matter; the teams fund the receipts and push them everywhere.

The key: be honest about what's proven vs measured

Your capsule can't "prove wall-clock time" in a trustless way by itself. That's fine.

What you *can* claim (and it's strong):

- Correctness of the benchmark workload execution/proof (your ZK verifier)
- Integrity of declared measurements (policy-bound + signed + anchored)
- Reproducibility constraints (docker/git/hardware anchors)
- Auditability (spot-check retrievability via your sampling mechanism)

This is aligned with EF's viewpoint that benchmarking is about defining constraints/fixtures and iterating measurements in a loop—not one-off conclusions. (zkEVM)

What you should actually build (minimum viable but "real")

1) A "Benchmark Policy v1" that verifiers enforce

Your *policy file* becomes the constitution. It should require:

Required anchors (anti-gaming)

- `benchmark_suite_id` + `suite_version`
- `workload_id` + `workload_input_hash`
- `git_commit` of the runner/workload
- `docker_image_digest` (or build hash)
- `hardware_manifest_hash` (CPU/GPU model + RAM + OS/kernel + driver versions)
- optional: `cloud_instance_type` (Brevis uses a specific instance type in their apples-to-apples setup; policies can standardize that.) (GitHub)

Required measurement procedure

- warmups
- repetitions
- aggregation rule (median, trimmed mean)

- fixed flags allowed / disallowed (e.g., “no compiler acceleration” track vs “optimized” track)

Required public outputs

- prove time, peak RSS, proof size, verify time, and the workload size parameter(s)

You already have the mechanism to bind the policy hash into the *statement hash in the transcript* (so results can’t be relabeled under a different methodology afterward).

2) A runner CLI that emits capsules (wrapping existing ecosystems)

You don’t need to fight existing tools. You *wrap them* and emit your receipt at the end.

- `zkvm-perf` already positions itself as a benchmarking tool for zkVM implementations with automated and manual modes. (GitHub)
- brevis’ `zkvm-bench` gives an explicit “apples-to-apples” template (same hardware, same Rust program, no accel). (GitHub)
- a16z’s repo is another community focal point for installing/running multiple zkVMs side-by-side. (GitHub)

Your move: **standardize output into capsules** and make the verifier the gate.

3) The leaderboard (two tracks at first)

Community track

- Must verify capsule + policy hash + anchors.

Certified track

- Requires `authorship` signature and ACL allowlist for that `policy_id`.
- Optionally requires policy registry inclusion proof (so “Certified Policy v1” is pinned to a registry root).

That one design choice prevents the leaderboard from becoming a marketing spam wall.

The leaderboard categories that avoid apples-vs-oranges

This is where most benchmark efforts fail. You need strict, obvious buckets:

1. **No-Accel / Baseline**

    - matches the “same program, no compiler accel/custom params” ethos that brevis calls out. (GitHub)
2. **Optimized (declared knobs allowed)**
    - allowed acceleration, but all knobs must be declared in anchors + policy.
3. **Recursion / aggregation track**
    - optional; only comparable if methodology is pinned.
4. **Ethereum block proving track (longer-term)**
    - aligns with EF’s active benchmarking pipeline for zkVMs for Ethereum, including worst-case fixtures and RTP framing. (zkEVM)

Why “now” is actually good timing

EF is actively building a benchmarking pipeline and explicitly cares about worst-case, reliability, and RTP constraints for mainnet readiness. (zkEVM)
That means: the ecosystem is going to converge on benchmark standards anyway. The only question is whether the “unit of truth” is:

- spreadsheets + blog posts, or
- **verifiable receipts**

You have the ingredients to make it the second.

If you want the sharpest one-sentence positioning

“We turn zkVM benchmarks into verifiable receipts—policy-bound, input-anchored, artifact-hashed, and optionally certified—so performance claims become reproducible objects, not marketing.”

If you want, next message I’ll draft:

- a concrete **Benchmark Policy v1 JSON** (fields + required anchors + tracks),
- a **capsule submission contract** (what your server checks before accepting),
- and the first **5 workloads** that are “high signal” and hard to game.

Yep. Here are **5 workloads** that (a) are *high-signal*, (b) map to real-world ZK/blockchain use, and (c) are hard to “game” without it being obvious. Then I’ll give you a **Benchmark Policy v1 JSON** + a **capsule submission contract** your server can enforce.

The 5 workloads

1) Keccak-256 streaming throughput (fixed bytes, fixed schedule)

Why it's high-signal: Keccak is everywhere (Ethereum, Merkleization, transcripts). It stresses pure CPU + memory bandwidth in a very "honest" way.

Workload definition

- Input sizes: `1 MiB`, `16 MiB`, `64 MiB`
- Operation: `keccak256(buffer)` repeated `R` times with a chaining rule (next buffer = previous digest expanded deterministically) so you can't short-circuit
- Output (public): final digest + bytes processed

Hard-to-game knob

- Inputs are *deterministically generated* from `suite_seed` + `size` (not user-provided)

---

2) Merkle tree build + random openings (hash-heavy + random access)

Why high-signal: This matches your world (commitments, chunk roots). It also hits cache/memory patterns.

Workload definition

- Leaves: `N = 2^20` leaves of 32 bytes (or 2^18 if too heavy initially)
- Hash: Keccak (or SHA-256 if you want non-EVM bias; you can have two variants)
- Task A: build root
- Task B: verify `Q=1024` inclusion proofs at random indices derived from seed
- Output (public): root + verification accumulator hash

Hard-to-game knob

- Indices come from seed; proofs must be consistent with the built tree

---

3) Signature verification batch (secp256k1 ECDSA verify)

Why high-signal: Big-int-ish arithmetic shows up; also extremely relevant to crypto systems.

Workload definition

- Verify `B=1024` signatures over fixed messages

- Messages, pubkeys, signatures are deterministically derived from seed (or taken from a canonical corpus with a published hash)
- Output (public): count of valid signatures + final accumulator digest

Hard-to-game knob

- The dataset is pinned by `workload_input_hash` (anchor), so nobody swaps "easy" signatures

---

4) Bytecode interpreter micro-EVM (control-flow + memory + hashing)

Why high-signal: This approximates "real computation" for zkVMs without requiring full Ethereum block proving.

Workload definition

- A tiny fixed instruction set: `ADD, MUL, XOR, SHL, SHR, MLOAD, MSTORE, JUMP, JUMPI, KECCAK_CHUNK`
- Execute `S` steps over a memory buffer with a deterministic program generated from seed
- Output (public): final state hash + step count

Hard-to-game knob

- Program is seeded and must match `program_hash` / `workload_input_hash`

---

5) Memory bandwidth + branching stress ("STREAM + branchy loop")

Why high-signal: It exposes VM overhead, memory model efficiency, and interpreter/JIT differences. Also reveals "cheating" quickly.

Workload definition

- STREAM triad style: `a[i] = b[i] + scalar*c[i]` over large arrays
- Plus a branchy pass: conditional swaps / thresholding using a seeded threshold vector
- Output (public): checksum hash of arrays + bytes touched

Hard-to-game knob

- Checksums are hashed, not just summed, to avoid algebraic shortcutting

---

Benchmark Policy v1 JSON (concrete)

This is **the policy file** you hash and bind into the statement transcript. It's written to be *machine-checkable* by your verifier/server.


```
{
  "schema": "bef_benchmark_policy_v1",
  "policy_id": "bef_bench_policy_v1",
  "policy_version": "1.0.0",
  "suite": {
    "suite_id": "bef-zkvm-leaderboard",
    "suite_version": "2025-12",
    "suite_seed_hex": "9b3b7a2c1e4d...deadbeef",
    "workload_registry_hash": "sha256:REPLACE_WITH_HASH_OF_WORKLOADS_JSON"
  },

  "tracks": [
    {
      "track_id": "baseline_no_accel",
      "display_name": "Baseline (no accel)",
      "rules": {
        "forbid_custom_precompiles": true,
        "forbid_handwritten_asm": true,
        "forbid_jit": false,
        "forbid_gpu": true,
        "allowed_optimizations": ["-O2"],
        "forbidden_optimizations": ["profile-guided", "link-time-opt"],
        "require_deterministic_build": true
      }
    },
    {
      "track_id": "optimized_declared",
      "display_name": "Optimized (declared)",
      "rules": {
        "forbid_custom_precompiles": false,
        "forbid_handwritten_asm": false,
        "forbid_jit": false,
        "forbid_gpu": false,
        "require_full_knob_disclosure": true,
        "require_deterministic_build": true
      }
    }
  ],
```

"required_anchors": {
  "git_commit": true,
  "runner_repo": true,
  "docker_image_digest": true,
  "hardware_manifest_hash": true,
  "workload_input_hash": true,
  "toolchain_hash": true,
  "os_fingerprint_hash": true
},

"required_public_outputs": {
  "prover_time_ms": true,
  "peak_rss_mb": true,
  "proof_size_bytes": true,
  "verify_time_ms": true,
  "workload_size_params": true,
  "result_digest_hex": true
},

"measurement_procedure": {
  "warmup_runs": 3,
  "measured_runs": 10,
  "aggregation": "median",
  "clock_source": "monotonic",
  "pin_cpu_affinity": true,
  "disable_turbo_if_possible": true,
  "notes": "If turbo cannot be disabled, report it in hardware manifest."
},

"workloads": [
  {
    "workload_id": "keccak_stream_v1",
    "params": { "sizes_bytes": [1048576, 16777216, 67108864], "rounds": 4 }
  },
  {
    "workload_id": "merkle_build_open_v1",
    "params": { "leaves_pow2": 20, "openings": 1024, "hash": "keccak256" }
  },
  {
    "workload_id": "ecdsa_verify_batch_v1",
    "params": { "curve": "secp256k1", "batch": 1024 }
  },
  {

```
      "workload_id": "micro_evm_interp_v1",
      "params": { "steps": 2000000, "mem_bytes": 16777216 }
    },
    {
      "workload_id": "mem_stream_branch_v1",
      "params": { "n_elems": 33554432, "passes": 3 }
    }
  ],

  "da_policy": {
    "verification_level": "probabilistic_da_sampling",
    "delta": 0.1,
    "epsilon": 1e-6,
    "min_k_samples": 64
  },

  "certification": {
    "require_signature_for_certified_track": true,
    "acl_schema": "bef_acl_v1",
    "policy_registry": {
      "enabled": true,
      "registry_root_hash_type": "sha256",
      "require_inclusion_proof_for_certified_track": true
    }
  }
}
```


Companion anchor formats (what the capsule must include)

Your capsule should include these keys under `anchors` (or a dedicated `benchmark` block—either is fine as long as the statement binds it):

- `git_commit`: commit of runner/workload repo
- `runner_repo`: canonical repo identifier
- `docker_image_digest`: immutable image digest
- `hardware_manifest_hash`: sha256 of a JSON hardware manifest
- `toolchain_hash`: sha256 of toolchain manifest (rustc/cargo/clang versions, etc.)
- `os_fingerprint_hash`: sha256 of OS/kernel/driver manifest
- `workload_input_hash`: sha256 of generated inputs (or canonical dataset tarball digest)
- `track_id`: one of the policy's tracks
- `workload_id`: one of the policy's workloads

Capsule submission contract (server-side checks)

This is what your leaderboard server enforces **before** it accepts a run.

A) Structural + integrity checks

1. **Parse + schema**
- capsule must be a dict
- `schema == "bef_capsule_v1"`
2. **Canonical hash**
- recompute capsule hash (after removing `capsule_hash`)
- must equal `capsule.capsule_hash`
3. **TraceSpec binding**
- must contain `trace_spec` + `trace_spec_hash`
- recompute and match
4. **Statement binding**
- must contain `statement` + `statement_hash`
- recompute and match
- statement must bind:
    - `trace_spec_hash`
    - `policy_hash`
    - `trace_root` (row commitment root)
    - anchors (or benchmark block)
    - public outputs
5. **Proof artifact integrity**
- if capsule includes `sha256_payload_hash` for proof formats, recompute and match
- proof must verify under the verifier using the provided statement hash

B) Policy + track enforcement

6. **Policy hash match**
- server computes `sha256(policy_file_bytes)` and must equal capsule `policy_hash`
7. **Certified track gating**
- if submission is for `certified`:

    - require policy registry inclusion proof against a server-configured `policy_registry_root`
    - require `authorship` signature

8. **ACL enforcement (certified)**

- require ACL file (or server-known ACL registry)
- signer pubkey must be authorized for this `policy_id`

9. **Track rule checks**

- enforce track constraints:
    - baseline track forbids GPU + custom precompiles + etc.
    - optimized track requires knob disclosure

These are enforced by requiring anchors/manifests that declare them + rejecting missing fields.

C) Anchor requirements (anti-gaming / reproducibility)

10. **Required anchors present**

- must contain all keys listed in `required_anchors`
- `workload_id` and `track_id` must be in policy

11. **Hardware manifest hash matches**

- user uploads `hardware_manifest.json`
- server hashes it and matches `hardware_manifest_hash`

12. **Toolchain + OS fingerprint hashes match**

- same idea: require uploaded manifests, hash them, match anchors

13. **Workload input hash matches**

- either:
    - inputs are deterministically generated from policy seed → server recomputes expected hash, OR
    - inputs are uploaded as a blob/tarball → server hashes and matches `workload_input_hash`

D) Output requirements (leaderboard fields)

14. **Required public outputs present**

- must contain all metrics keys required by policy
- check types/ranges (e.g., non-negative, reasonable bounds)

15. **Consistency checks**

- `proof_size_bytes` must match actual proof file size
- `verify_time_ms` must be from a declared verifier version (anchored)
- `workload_size_params` must match policy workload params

E) DA / retrievability audit (what you already support)

16. **DA policy sanity**
- must include `verification_level="probabilistic_da_sampling"`
- enforce `k_samples >= min_k_samples`

17. **Server-performed sampling**
- server derives audit seed deterministically (from capsule hash + anchors per your design)
- server requests sampled chunks from the declared provider (or from uploaded row archive)
- server verifies Merkle inclusion up to `trace_root`
- failures reject the submission (or mark as "DA failed")

Recommended "day 1" simplification (still honest)

To ship quickly without weakening integrity:

- Require submissions to upload a **bundle**:
  - capsule
  - proof artifact(s)
  - row archive (or chunk roots + sampled chunks)
  - hardware/toolchain/os manifests
- Server stores those, then runs verification + sampling.

You can later support remote DA providers.

If you want, next I can also generate:

- a `workloads.json` registry file (the canonical definition you hash into `workload_registry_hash`)
- example `hardware_manifest.json`, `toolchain_manifest.json`, and `os_fingerprint.json`
- a "badge rules" spec: when a run gets ✅ Verified / ✅ Certified / ✅ DA-Audited

Yeah — **the pipeline can (and should) automatically gather + shape the whole submission UX**. In fact, *your current geom pipeline already does ~60–70% of it* (it computes hashes, writes the row archive, emits capsule + proof artifacts). What's missing is basically: **(a) auto-manifests + (b) bundling + (c) a clean adapter interface so it's not geom-only.**

Here's the clean breakdown: what the system can auto-do vs what an adopter has to do.

Can the pipeline form the "UX shape" automatically?

✅ Yes, for everything that is "machine-observable"

Your runner can automatically:

- compute `policy_hash`
- compute `trace_spec_hash` and `statement_hash`
- compute `capsule_hash`
- compute payload hashes for proof files
- export `row_archive/` + chunk roots + metadata
- run local verification
- produce a single **bundle** (capsulepack) ready to upload

And it can automatically **collect**:

- hardware fingerprint (CPU/GPU/RAM)
- OS/kernel + driver versions
- toolchain versions (rustc/cargo, clang, etc.)
- proof sizes
- verification time (measured)
- prover time + peak RSS (measured)

None of that requires humans.

⚠️ What it cannot "magically know"

It can't infer:

- what *your* computation semantics are (that's the Trace Adapter's job)
- what anchors are "the right ones" for your domain (policy decides this)
- whether you "used forbidden optimizations" unless you require disclosure and/or run in a controlled environment (docker + pinned runner)

So you don't prove "no JIT." You enforce it by:

- (baseline track) running inside a standardized container/harness you provide, and
- requiring manifests + disclosure + reproducible build.

What does an adopter have to do?

This depends on what “adopter” means: **(A) someone submitting a benchmark run** vs **(B) a team integrating a new VM/computation into the capsule framework**.

A) A *benchmark submitter* (most users)

If you ship `capsule-bench`, the submitter does basically:

1. **Install runner**

cargo install capsule-bench   # or curl | sh

2. **Run**

capsule-bench run --workload merkle_build_open_v1 --track baseline_no_accel

3. **Submit**

capsule-bench submit --pack out/capsulepack.tgz

That’s it. They never hand-write hashes, never craft manifests.

If they want **Certified track**, add:

capsule-bench run ... --sign --key ~/.bef/keys/secp256k1.pem

B) A *system integrator* (zkVM/prover team adding support)

This is the real “adoption surface” question. For them, the work is:

1) Implement a Trace Adapter (the only real integration cost)

They provide a small interface like:

- `trace_format_id` (e.g. `SP1_VM_V1`, `RISC0_VM_V2`, `MYVM_AIR_V1`)
- `record_schema_ref` (hash of the row schema they output)
- `encode_trace(rows) -> stc_rows` (or whatever your STC ingestion format is)
- `extract_public_outputs(run) -> dict`
- `anchors_required(run) -> dict` (git commit, docker digest, workload input hash, etc.)

- optional `da_export(trace) -> row_archive`

Everything else (hashing, statement building, capsule emit, bundle) is your framework.

2) Plug into your runner wrapper

They essentially implement:

capsule-bench run --backend myvm

3) (Optional) Provide a verification artifact path

So your verifier can call:

- `zk_verify_myvm(proof, statement_hash, vk, …)`

That's it. They don't have to understand your capsule internals beyond a schema.

The "capsulepack" UX shape you should standardize

Make the output of **every run** identical:

```
capsulepack/
  capsule.json
  policy.json            (or policy_id + inclusion proof)
  proofs/
    geom.proof.bin
    geom.vk.bin          (if needed)
  row_archive/
    chunk_roots.bin
    chunk_roots_digest.txt
    ... optionally sampled chunks for DA
  manifests/
    hardware.json
    os.json
    toolchain.json
  measurements.json      (prover_time_ms, peak_rss_mb, verify_time_ms, etc.)
  logs/
    run.log
```

A submitter just uploads **one tarball**. Your server does:

- unpack
- verify capsule hash + statement/trace spec hash
- verify proof
- verify required anchors/manifests
- do DA sampling audit (either from uploaded archive or remote provider)

So: what is the adopter *actually* responsible for?

If they're just submitting

- run one command
- upload one file
- optionally sign with a key

If they're integrating a new backend/computation

They are responsible for **exactly one thing**:

> Provide a deterministic way to produce a trace + public outputs + anchors in a known schema (Trace Adapter), so your framework can bind and verify it.

Everything else should be framework-provided.

If you want, I can write the **exact TraceAdapter interface** as a minimal Python/TypeScript spec (like 15–25 lines of required methods + expected outputs), and the **capsulepack schema** + the server-side validation order — so "adopter effort" becomes a checklist you can hand to any zkVM team.