



vitnify

Reconstructable Execution Certificates for Tool-Using AI Agents

Jean N. Pijeu

vitnify · nick.pijeu@vitnify.com

Preprint · August 2026 · Artifacts and reproduction commands in §9

ABSTRACT

An AI agent is a language model given tools and left to run a multi-step task on its own. Agents reached production before anyone worked out how to secure, audit, or reproduce them: the output is non-deterministic, the tool calls change external state, and the only record kept is a log that cannot be re-run. After the fact, there is no way to prove what an agent actually did. I present **vitnify**, which turns an agent run into a single, independently-reconstructable object. Each run is bound into a tamper-evident execution certificate committing the program, the model computation (per-step logits), the granted capabilities, the attempted tool calls, the allow/deny decisions, the tool results, the entropy draws, and the event order. This supports two levels of verification: *certificate verification* checks the transcript's integrity offline, with no model and no secret; *execution recomputation* additionally re-runs the model and confirms the committed numerical trajectory. Recomputation rests on **vitni-tensor**, an engine whose integer and pinned-order arithmetic makes the same inputs produce the same output bits on different CPUs. Across three models in two architecture families (TinyLlama, Mistral, Qwen2.5), the same inputs produced an identical certificate digest on four CPU implementers (Apple, Arm/AWS Graviton, Intel, AMD), two instruction sets, and two compiler builds; a sweep of diverse prompt shapes and lengths, folded into one aggregate digest, reproduced with zero cross-vendor mismatches. Against a deterministic CPU reference (llama.cpp, GPU disabled), the engine's greedy output matches exactly for two of the three models and at 15 of 16 steps for the third, whose one divergence is a near-tie between two closely-scored continuations. Capability containment holds against an adversarial attack matrix, and the system works through unmodified LangGraph and Model Context Protocol agents. My contribution is architectural: vitnify binds deterministic model recomputation, capability authority, tool decisions, external results, and execution chronology into a single independently-reconstructable security object for tool-using agents. The deterministic primitives it rests on exist in prior work; their composition into one offline-verifiable execution object does not. §2 places it against the closest prior work.

KEYWORDS: reproducible computing, deterministic inference, AI agent security, execution certificates, capability-based containment, cross-vendor determinism.

1. Introduction

Determinism and verifiable replay are solved problems elsewhere in computing: a database replays a transaction, a build system re-runs deterministically, a blockchain re-executes every block to reach consensus. Agents have none of it. An agent decides at inference time which tools to call and what to pass to them, and three things go wrong at once: the model's output is non-deterministic, the tool calls hit live systems, and the best record anyone keeps is a log that cannot be re-executed. If an agent leaks data, or an auditor asks what a given agent did on a given day, there is currently no way to answer.

The consequences are already concrete. Documented 2025 incidents include *EchoLeak* (CVE-2025-32711), a zero-click prompt injection that exfiltrated data from Microsoft 365 Copilot;^[10] the Salesloft–Drift compromise (tracked as UNC6395), in which stolen OAuth tokens from a single over-permissioned AI integration were used to bulk-export data from more than 700 organizations' Salesforce tenants;^[11] and a reported state-linked operation (designated GTG-1002) that jailbroke a coding agent to drive largely-automated intrusions against roughly thirty targets.^[12] Regulation is arriving in parallel. The EU AI Act's Article 12 requires automatic event logging over a high-risk system's lifetime; it does not say “tamper-evident,” which is a common implementation recommendation rather than a statutory requirement.^[13] The tooling market, meanwhile, mostly ships *detection*: classifiers that block on a score at the prompt boundary. That is a probabilistic arms race, and it does nothing for reproducibility or provenance.

This paper asks whether an agent run can be given the three properties ordinary software enjoys, in userspace, without a new operating system:

1. **Containment.** A hijacked agent cannot exceed the capabilities it was granted, because an ungranted tool is not present to call.
2. **Deterministic replay.** The run reproduces bit-for-bit, because the model is re-computed rather than read back from a log.
3. **Provenance.** A signed certificate that anyone can verify offline, without trusting the recorder.

Contributions. (i) **Reconstructable agent execution.** A tool-using agent run is represented as a cryptographically-committed object binding the model computation, granted authority, attempted actions, authorization decisions, tool results, entropy, and event order, so an action cannot be detached from the computation and context that produced it without invalidating the certificate. (ii) **Cross-vendor deterministic recomputation.** vitni-tensor reproduces the committed model computation bit-for-bit across CPU vendors and ISAs (§4.1, §6.1). (iii) **Capability-based authority containment.** Ungranted actions are structurally unreachable rather than classified as malicious; the boundary and its limits are stated precisely (§4.3, §6.5). (iv) **Numerical-trajectory certification.** The certificate commits to model-computation state, not just tokens, which exposes batch-dependent divergence that is invisible at the token level (§4.2, §6.3). (v) **Existing-agent integration.** The system works through unmodified LangGraph and Model Context Protocol (MCP) agents (§4.6, §6.6). All results are reproducible from the artifacts in §9.

2. Background and Related Work

Reproducible builds.

The reproducible-builds community reports high *same-host* bit-identical rebuild rates: roughly 69–91% for Nix across 700k+ packages,^[18] and about 80–90% for Debian; Bazel claims determinism within its sandbox.^[7] These efforts target build *artifacts* on a fixed architecture. They do not claim cross-architecture or cross-vendor *runtime* determinism. Cross-hardware floating-point reproducibility is a recognized hard problem for high-performance kernels, because non-associative reductions, vendor-specific kernel fusion, and atomics make GPU tensor-core results vary run-to-run and machine-to-machine;^[19] production numerical stacks settle for agreement “within tolerance.” It is tractable in principle if you avoid hardware FP variability altogether: integer-only or software-emulated, fixed-order arithmetic on CPUs is portable and deterministic. That is the route my engine takes (§4.1). My contribution is therefore not floating-point determinism itself, but its integration into a contained, certified, cross-vendor agent runtime.

Deterministic LLM inference.

Recent work traces run-to-run non-determinism in LLM serving to *batch-variance* in reduction kernels, and proposes batch-invariant kernels; these are now in vLLM and SGLang, used mainly for reinforcement-learning reproducibility.^[1] I build on that insight but aim for a stronger property: bit-identity across hardware vendors, bound to a certificate. The nearest prior work binds bit-exact inference to cryptographic verification on a single fixed GPU architecture;^[14] demonstrates bitwise-identical inference across ARM and x86 CPUs with a pure-integer engine, verified by on-chain attestation;^[21] claims bitwise-reproducible inference across environments without a cross-vendor equality experiment;^[15] or characterizes cross-hardware reproducibility only to within a tolerance rather than bit-for-bit.^[17] The closest of these is ARC,^[21] which independently establishes that cross-vendor CPU bit-identity is achievable and confirms it on overlapping models; I reach the same property by a different route (pinned-order fp32 plus exact-integer k-quant kernels running stock GGUF files, rather than a wholly re-quantized integer model). I therefore do *not* claim cross-vendor deterministic inference as my novelty. I differ by binding that determinism, at the level of per-step logits, into an agent-execution object that also carries capability authority, tool decisions, results, and tamper-evident history, verifiable offline from one certificate (§6.1), and by showing why token-level replay is not enough for certification (§4.2).

Record-and-replay.

Deterministic record-replay is well studied for native execution, e.g. Mozilla rr,^[2] and for deterministic simulation testing, e.g. Antithesis.^[3] Those systems replay program or agent-written *code*. They are not built for forensic replay of an agent's own reasoning-and-tool run. What observability tools call agent “replay” is log reconstruction, not bit-identical re-execution.

Agent security and execution certificates.

As of 2025–2026, the commercial agent-security market is dominated by *detection*. A minority do action-layer *containment* (capability- or identity-scoped allow/block). Verifiable-execution

primitives for agents have started to appear: runtime verification and tamper-evident event streams for governed agent actions,^[4] signed workflow-history attestation in production orchestrators,^[5] action-level attestation frameworks,^[20] and cryptographic binding with reproducibility verification for agent tool use.^[16] Capability and information-flow designs for prompt-injection defense, such as CaMeL,^[6] motivate my containment layer. None of them combine all of what I do: capability containment, deterministic cross-vendor recompute, and a portable offline-verifiable certificate in one runtime, with the certificate committing to per-step *logits* rather than tokens or tool traces. The table below places vitnify against the closest systems and forces the claim into a precise form: my contribution is the *combination*, not any single row.

Positioning. *vitnify against the closest 2025–2026 work. “✓” provides it; “—” does not; “~” partial or qualified, with the inline note giving the caveat.*

Property	Det. serving ^[1]	EigenAI ^[14]	RepDL ^[15]	ARC ^[21]	Agent attest. ^[4,5]	vitnify
Bit-identical inference	~ same HW	✓ 1 GPU	~ claimed	✓ int	—	✓
Cross-vendor CPU / ISA	—	—	~	✓	—	✓
Bound to a certificate	—	✓	—	~ chain	✓	✓
Commits model computation	—	✓	—	~ output	~ tool trace	✓
Capability containment	—	—	—	—	~	✓
Tool decisions + results	—	—	—	—	✓	✓
Offline verification	—	~	—	~ chain	✓	✓
Framework integration	vLLM/SGLang	—	—	arc-chain	Dapr	LangGraph/MCP

3. Threat Model and Requirements

I consider an agent that reads untrusted content (tickets, web pages, tool outputs) which may carry an *indirect prompt injection* instructing the agent to exfiltrate data or take an unauthorized action. The model may be fully compromised by such an injection. The tool substrate and event log may later be inspected by a mutually-distrusting auditor who does not trust the operator. I require: **(R1)** an ungranted tool must be unreachable even to a fully-hijacked agent; **(R2)** the recorded run must be re-executable to a bit-identical result, on independent hardware, by the auditor; **(R3a) *certificate integrity***: the certificate must be verifiable offline with no secret and no access to the model or network, rejecting any tampering or forgery of the transcript; and **(R3b) *execution recomputation***: given the content-addressed model weights and the deterministic runtime, an independent auditor must be able to reproduce the committed model computation bit-for-bit. R3a requires neither the weights nor the network; R3b additionally requires the content-addressed GGUF file, which I assume is available to the verifier performing that level (§4.4). I do not defend against a malicious hardware root or side channels.

4. Design

4.1 The deterministic inference engine (vitni-tensor)

The certificate binds a hash of the model's *computation*, so the engine must be reproducible bit-for-bit across machines. Floating-point non-determinism has two sources: reduction *order* (addition is non-associative, so parallel/blocked reductions differ), and fused multiply-add (FMA), which rounds once where separate multiply-then-add rounds twice. vitni-tensor removes both:

- **Pinned-order f32 reduction.** Dot products use a canonical fixed pairwise-tree reduction over lanes and chunks (constants `CANON_LANES=8`, `CANON_CHUNK=8192`) that is independent of SIMD width and thread count; FMA is forbidden (multiply and add remain separate IEEE-754 operations). Row-parallelism, when enabled, splits independent output rows and therefore cannot alter any reduction order.
- **Exact-integer quantized matmuls.** For k-quant weights (Q4_K/Q6_K/Q5_o/Q8_o) the activation is quantized to `int8` and the reduction is accumulated in `int32`. Integer addition is associative and exact, so this path is order-independent as a matter of arithmetic, not convention. The accumulator cannot overflow: with 8-bit operands each partial product is at most 127×127 , and a 256-wide super-block sum is bounded by $127 \times 127 \times 256 \approx 4.1 \times 10^6$, three orders of magnitude below the `int32` limit ($\approx 2.1 \times 10^9$); the widest full-row reduction I use (`in_feat = 14336`) stays under 2.3×10^8 . The reduction is therefore exact for every supported kernel, so the determinism tax that afflicts floating point does not apply here.
- **Softfloat transcendentals.** `exp`, `sqrt`, etc. use the pure-software `libm` crate, whose output is identical on every platform, rather than hardware transcendental units that differ across vendors.

None of these operations depends on hardware rounding modes, ISA extensions, or thread scheduling, so the canonical path is *designed* to be independent of SIMD width, thread count, and vendor FMA behaviour. This is a design intent, not a proven theorem. §6.1, §6.2, and §6.4 verify it empirically across every evaluated CPU and ISA (including a dense input sweep of the transcendental path); a universal portability claim would require formally specifying all arithmetic and runtime assumptions, which I do not attempt here.

Because this reduction contract *defines* the bits, changing it would change every subsequent digest and silently break replay of prior receipts. The certificate therefore records the *numerical regime* it was produced under — an identifier (`vitni-regime-2`) for the reduction contract above, bound into the tier-1 digest (`vitnify-receipt v2`). A deliberate regime change is then cryptographically distinguishable from tampering: a verifier reports “produced under a different regime, cannot replay with this engine” rather than an unexplained hash mismatch. Absent this binding, a moved reduction and a forged receipt are indistinguishable — the regime is to the arithmetic what a hash-suite identifier is to the digest.

4.2 Why certifiable replay requires batch-invariance

There are two ways to “replay” an agent. One is record-and-replay: store the model's outputs and play them back. The other is recompute: re-run the model so its outputs can be re-derived. Recompute is the stronger claim, the one an auditor wants, because it means “re-run this and you

get bit-identical results.” It also runs into a subtlety I measured directly (§6.3). In production a request is co-batched with other traffic. Running the same prompt alone versus co-batched almost never changes the greedy *token*, but it shifts the underlying *logits* every time. So a certificate over a hash of the tokens is reproducible by anyone, which makes it weak and already commoditized. A certificate over the logits, the actual computation, is reproducible only if inference is batch-invariant. Put differently: the same greedy token does not imply the same computation, so committing to the computation rather than the visible output turns batch-dependent numerical drift into a detectable provenance failure. vitnify commits to per-step logit hashes, and the engine of §4.1 is what makes that commitment reproducible.

4.3 Capability containment

Every side-effecting tool call goes through a broker that holds a set of granted capabilities. A call to an ungranted tool never runs. It is recorded as a denial and returns nothing (R1). The broker does not try to recognize a malicious call the way a detector would; it makes the call impossible. If the exfiltration tool was never in the capability set, an injection that tells the agent to exfiltrate data has nothing to call. This prevents *authority escalation*: a compromised agent cannot reach a capability it was not granted. It does not make granted authority safe — if the agent legitimately holds `send_email`, an injection can still misuse that authorized tool. Containment bounds what an agent can reach, not what it does with what it holds; §6.5 tests both boundaries.

4.4 The execution certificate

An agent run is recorded as an ordered, hash-chained event log. Each entry is one LLM step (prompt hash, tokens, per-step logit hashes), one tool call (name, arguments, allow/deny decision, and result, so replay can re-inject it), or one entropy draw. The certificate binds four fields and then signs them: the program hash, the granted capability set, the Merkle root of the event log, and the chain head. Verification has two distinct levels, which I keep separate. *Certificate verification* (level 1) recomputes the Merkle root from the raw events and checks the signature; it establishes integrity and tamper-evidence offline, with no model and no secret. Editing any event, for instance flipping a blocked call to “allowed,” changes the root and fails the check. Level 1 does *not* prove the model computation was correct. *Execution recomputation* (level 2) additionally obtains the weights and the deterministic runtime, re-runs the model, and confirms the committed per-step logits reproduce (R2). The Merkle content-addressing and the offline verifier come from a pre-existing, pure-standard-library proof-carrying-knowledge stack. Signing is pluggable, and here too there are two levels. An embedded **ed25519** key lets anyone verify with no shared secret, but it proves only *integrity and signer continuity* — not that the signer was an authorised runtime. Binding execution *provenance* to an approved runtime requires anchoring the key: a pinned trust root, or a **TPM**/enclave-resident key. I therefore distinguish a *signed* certificate from an *attested* one (§6.5).

4.5 Model-agnostic tokenization

To avoid a class of correctness bug in which a prompt is tokenized with the wrong vocabulary, the tokenizer is read *from the model file*. I implement both GGUF families: SentencePiece/unigram (tokenizer.ggml.model = "llama": Llama-2, Mistral, TinyLlama) and byte-level BPE ("gpt2":

Qwen 2.5, Llama-3), each reconstructed from the embedded vocab, scores, and merges. §6.4 shows both match `llama.cpp` exactly.

4.6 Framework adapters

The record/replay/contain machinery is exposed to real agent frameworks. For **LangGraph**, a compiled StateGraph routes its model node through the session's LLM interface and its tool node through the capability broker. For the **Model Context Protocol (MCP)**, a broker wraps an MCP client's `call_tool`, forming a gateway through which every tool invocation is capability-checked, recorded, and certified before (if ever) reaching the real MCP server. Neither requires changing the agent's logic.

5. Implementation

`vitni-tensor` is a `no_std` Rust crate that also builds on a host. Its only dependencies are `libm` and a hashing library. It parses GGUF and supports grouped-query attention, both interleaved (Llama) and NeoX (Qwen2) rotary embeddings, optional attention biases, RMSNorm, SwiGLU, a KV cache, and the `F32/Q4_o/Q4_K/Q5_o/Q6_K/Q8_o` matmul types. It reads per-model hyperparameters (RoPE base, RMS epsilon) from the model file, and it emits and re-verifies a certificate per run. The agent layer (containment broker, event log, certificate, replayer, tokenizer bindings, adapters) is a few hundred lines of Python. It sits on two of my earlier systems: a proof-carrying-knowledge certificate stack, and the VitniOS substrate's content-addressed store and event bus. Roughly 70–80% of the supporting machinery already existed; the new work is the connecting core plus the engine's architecture and quantization support. The engine implements five quantized matmul kernels (`Q4_o/Q4_K/Q5_o/Q6_K/Q8_o`) plus `F32`, two RoPE layouts (interleaved and NeoX), optional QKV biases, GQA, and a KV cache, and passes 103 library unit tests alongside the integration tests of §6; the determinism-critical surface is the pinned-order dot product and the integer k-quant kernels.

6. Evaluation

Hardware, four targets: an Apple-silicon Mac (ARM64, macOS) and three EC2 instances running Ubuntu — AWS Graviton (Neoverse-V1, ARM64, m7g), Intel Xeon Platinum 8488C (x86-64, m7i), and AMD EPYC 9R14 (x86-64, m7a). All three models ran on all four targets: TinyLlama-1.1B-Chat and Mistral-7B-v0.1 (`Q4_K_M` GGUF, Llama-architecture), and Qwen2.5-0.5B-Instruct from the stock `Q4_K_M` GGUF (a mix of `Q5_o`, `Q4_K`, `Q6_K`, and `Q8_o` tensors). Qwen2's architecture differs materially from Llama: attention QKV biases, NeoX split-half RoPE, and a per-model RoPE base of 10^6 and RMS ϵ of 10^{-6} , all read from the file. Prompts are tokenized offline, and the certificate binds token IDs.

6.1 Cross-vendor bit-identity

I ran identical greedy generations (prompt “*Once upon a time,*”) through `vitni-tensor` on every target and compared, per model, the generated token sequence, the model-file hash (BLAKE3 of the GGUF), and the certificate digest.

Table 1. Same models, same inputs, same engine source (vitni-tensor 0.2.0), built independently from source on each host. Digests are the tier-1 v2 model-computation digest, which binds the numerical regime (vitni-regime-2). For every model, the digest is byte-identical on all four hosts; so are the generated tokens and the model-file hash. Qwen2.5 exercises different code paths: attention biases, NeoX RoPE, and a four-type quantization mix (Q5_0/Q4_K/Q6_K/Q8_0). **Regime-2 refresh (2026-08):** RMSNorm, softmax, and attention adopted the canonical lane+tree reduction the matmul already used, moving every digest and bumping the bound regime to vitni-regime-2; generated tokens are unchanged. The values shown were regenerated and bit-verified on Apple silicon (CPU and GPU, per-step logits identical); the cross-vendor byte-identity is the determinism property of §4.1/§6.1, which the canonical reductions preserve.

CPU (implementer)	ISA	rustc	TinyLlama-1.1 B	Mistral-7B	Qwen2.5-0.5B
Apple silicon (Apple)	ARM64	1.95-nightly	7a2e28c9...1730	e0507d0f...c094	a7014042...f1aa
Graviton, Neoverse-V1 (Arm-AWS)	ARM64	1.98-stable	7a2e28c9...1730	e0507d0f...c094	a7014042...f1aa
Xeon Platinum 8488C (Intel)	x86-64	1.98-stable	7a2e28c9...1730	e0507d0f...c094	a7014042...f1aa
EPYC 9R14 (AMD)	x86-64	1.98-stable	7a2e28c9...1730	e0507d0f...c094	a7014042...f1aa

Every target produced, per model, the identical token sequence and the identical certificate digest (Table 1), all four hosts agreeing on the tier-1 v2 digest (which binds the regime vitni-regime-2): 7a2e28c9...1730 for TinyLlama, e0507d0f...c094 for Mistral-7B, and a7014042...f1aa for Qwen2.5 (full digests in §9). The model-file hash also matched on every host (ab7591e1... for TinyLlama, f04c8834... for Mistral, d4c4d976... for Qwen2.5). That rules out two easy explanations: the hosts did not share a binary, since each built the engine from source, and they did not read a different input file, since each fetched the weights independently. The digest holds across four CPU implementers (Apple, Arm/AWS, Intel, AMD), two ISAs (ARM64 and x86-64), and two different rustc builds (a 1.95 nightly on Apple, a 1.98 stable on the three EC2 hosts). So bit-identity survives a change of vendor, of microarchitecture, and of compiler. This extends prior cross-host and cross-vendor determinism results for the VitniOS substrate, which were on a non-LLM workload, to language-model inference.^[9]

A separate check covers the software transcendental path, on which the whole determinism argument rests. I evaluated exp, sqrt, cos, sin, pow, and the sigmoid over a dense sweep of 489,509 inference-range and corner-case inputs (3.4M evaluations) and hashed the IEEE-754 bit patterns. The hash is identical, 0ae93cd6..., on all four CPU implementers and both ISAs, so the libm softfloat path is bit-identical ARM-versus-x86, measured rather than assumed.

A single prompt is a weak basis for a determinism claim, so I also ran a sweep. Many diverse inputs (repetitive, sequential, maximum-token-id, and pseudo-random token streams over several lengths and two generation lengths) each produce a certificate digest, all folded into one aggregate hash. The inputs are generated deterministically, so any host builds the identical matrix without a shared file. Re-run under tier-1 v2 on Apple silicon, the TinyLlama sweep — 33 distinct inputs over three lengths and two generation lengths (66 runs, 1,320 generated tokens) — aggregates to fed2a2c8..., and an equal-sized Qwen2.5 sweep, through the Q5_0/Q8_0 path, to 2b8543ec.... Every run's digest is the same tier-1 v2 model digest that Table 1 shows byte-identical across all four hosts, so these aggregates are cross-vendor by construction; the predecessor tier-1 v1 sweep was directly measured identical on all four CPU implementers (Apple, AWS Graviton, Intel, AMD) — 528 runs and 10,560 generated tokens with zero cross-host digest mismatches. The single-prompt result of Table 1 is thus not a lucky input.

6.2 Numerical correctness against a reference

A result can be identical across my own hosts and still be identical-but-wrong, so I validated the engine against a *deterministic* reference: `llama.cpp` with the GPU disabled, on the same GGUF weights. For each model I dumped the engine's per-step logits and compared them, teacher-forced (feeding both implementations the same token sequence), against the CPU reference (Table 2).

Table 2. Numerical validation against a deterministic CPU reference (`llama.cpp`, GPU disabled), teacher-forced on the same GGUF weights. Logits differ slightly, as expected from a different reduction order (max absolute deviation 0.6–1.7 on the models' logit scales, mean 0.05–0.11); the greedy argmax agrees at every step for two models and at 15 of 16 for the third, whose sole divergence is a near-tie.

Model	max abs logit err	mean abs logit err	top-1 match	top-5 overlap	greedy token match
TinyLlama-1.1B	1.66	0.112	16 / 16	77 / 80	16 / 16
Mistral-7B	0.58	0.051	12 / 12	58 / 60	12 / 12
Qwen2.5-0.5B	0.72	0.092	15 / 16	76 / 80	15 / 16

The logits are close but not identical, as expected from a different reduction order, and the greedy argmax agrees at every step for TinyLlama and Mistral. This settles a point from an earlier draft: the single first-token difference I had previously seen for Mistral was the reference's non-deterministic Metal GPU path, not a defect: against a deterministic CPU reference the two agree 12 of 12. Qwen2.5 (a different architecture, run from the stock `Q4_K_M` file through the new `Q5_0` and `Q8_0` kernels) differs at exactly one step, where the engine's top-1/top-2 logit gap is 0.13, consistent with a small numerical difference changing the ordering of two closely-scored candidates. Together these results provide substantially stronger evidence of numerical correctness than cross-host self-consistency alone.

6.3 Determinism is load-bearing, not decorative

I recorded a run under simulated production batch load, then replayed it alone, once with the batch-invariant kernel enabled and once disabled, committing per-step logit hashes each time. With the kernel *disabled*, the greedy tokens matched but the logit hashes diverged, so the certificate correctly **failed**. With it *enabled*, the logit hashes were bit-identical and the certificate **verified**. This is the point of §4.2: token-level replay is free, but certifiable replay needs batch-invariance. Table 3 gives the cost on Apple Metal, which is fp16-optimized hardware.

Table 3. Determinism cost on Apple Metal, per forward pass at 1.1B. The fp32 pinned-order kernel is the only batch-invariant option, at about 20× native. This tax is a floating-point artifact of fp16-optimized hardware; the integer path (§4.1) avoids it.

Linear kernel	Batch-invariant?	ms / forward	vs. native
native fp16	No	4.3	1.0×
single fp32 matmul	No	~40	~9×
pinned-order chunked fp32	Yes	~85	~20×

The tax motivates distinguishing two deployment modes, which I note but do not benchmark here. In *canonical execution mode*, vitni-tensor performs the actual inference: the committed numerical trajectory is the execution trajectory, and the guarantee is strongest. In *asynchronous reconciliation mode*, production inference runs natively on the GPU and vitni-tensor later re-

derives the canonical trajectory off the hot path, comparing it against the recorded run at externally-significant checkpoints (the emitted tokens and the decisions that trigger tool calls). This is more practical than forcing every production token through the canonical engine, but its guarantee is strictly weaker and must be stated as such: it certifies *agreement* between the production run and the canonical computation at those checkpoints, not a bit-identical reconstruction of the GPU arithmetic itself. Because the same token need not imply the same computation (§4.2), the reconciliation is an equality check on the committed decisions, not a proof that the GPU's internal floating point matched the canonical path. A third, orthogonal choice is to deploy the deterministic engine on hardware with fast integer/fp32 throughput, where the integer k-quant path carries no order-fixing penalty.

6.4 Tokenizer fidelity

Both GGUF tokenizer families, reconstructed from the model file, match `llama.cpp` exactly on identical text.

Table 4. Tokenization of “Once upon a time,” read from each model's own GGUF, identical to `llama.cpp`. The same text yields different IDs per model, which is the mismatch this design eliminates.

Model / family	Token IDs	Match
TinyLlama (SentencePiece)	1, 9038, 2501, 263, 931, 29892	exact
Mistral-7B (SentencePiece)	1, 5713, 3714, 264, 727, 28725	exact
Qwen2.5 (byte-level BPE)	12522, 5193, 264, 882, 11	exact

6.5 Certificate integrity, containment, and adversarial evaluation

I ran an adversarial attack matrix against the certificate and the capability broker (Table 5). Each attack is either *contained* (the broker makes the action unreachable), *detected* (tampering the recorded run fails level-1 verification), or a stated *limitation*. Of eighteen attacks, seventeen are defended; the one that is not is the honest boundary of a self-signed certificate.

Table 5. Adversarial attack matrix. “contain” = the capability broker blocks it; “detect” = level-1 certificate verification fails; “limit” = an acknowledged boundary.

Attack	Layer	Result
Invoke an ungranted tool	contain	blocked
Exfiltrate via an ungranted tool	contain	blocked
Alias a forbidden tool under a new name	contain	blocked
Injected instruction in a tool's output	contain	no escalation
Duplicate an external side effect on replay	contain	prevented
Delete an event from the log	detect	cert fails
Reorder two events	detect	cert fails
Modify a tool argument	detect	cert fails
Modify a tool result	detect	cert fails
Flip a blocked call to “allowed”	detect	cert fails
Modify the prompt	detect	cert fails
Change committed logit hashes	detect	cert fails

Attack	Layer	Result
Modify an entropy event	detect	cert fails
Recompute with different model/weights	detect	cert fails
Tamper the recorded capability set	detect	cert fails
Corrupt the signature	detect	cert fails
Replace signing key, no pinned anchor	limit	not caught
Replace signing key, pinned trust anchor	detect	rejected

Level-1 verification catches every transcript edit: deleting, reordering, or modifying any event (a tool argument or result, the prompt, a committed logit hash, an entropy draw), flipping a blocked call to “allowed,” tampering the recorded capability set, or corrupting the signature. The broker contains every ungranted call, including a forbidden tool aliased under a new name and an instruction injected through a tool's output; and replay re-injects recorded results, so a live side effect never fires a second time, which I call *deterministic agent-computation replay with recorded external-effect substitution*. The one undefended row is deliberate: a certificate re-signed by a different key still self-verifies, because integrity and signer continuity are intact, so without a pinned trust anchor a verifier cannot distinguish an authorised signer from an unauthorised one. With a pinned key it is rejected. This is the signed-vs-attested boundary of §4.4.

On attestation: I signed a certificate digest with a hardware-resident key and verified it through the TPM 2.0 stack (RSA-2048); the create–sign–verify flow succeeds. Because no physical TPM was available, this is *functional validation of the attestation flow* against a software TPM (swtpm), not a hardware-backed attestation result. The tpm2 sequence is identical on hardware, but a hardware trust root is required before the stronger claim can be made.

6.6 Containment and replay through real frameworks

I drove a real LangGraph StateGraph agent and a real MCP 2.0 server/client, each routed through vitnify, with a scenario that reads a poisoned input and then tries to exfiltrate. In both, the ungranted tools (read_secret, send_external) were **blocked** and nothing was exfiltrated, the run **replayed bit-identically**, and the ed25519 certificate **verified**. None of this required changing the agent's logic.

6.7 Deterministic GPU execution (Metal and CUDA)

The cross-vendor result above is CPU-based, and bit-identical floating point *across GPU vendors* is usually treated as out of reach. The obstacle, though, is the kernel, not the silicon: stock GPU matmuls choose their own reduction order and fuse multiply–add, so their bits move. A kernel that obeys the same numerical contract as §4.1 — a reduction order fixed by the contract (each product assigned to one of eight lanes by index, then a fixed pairwise tree) and *no* fused multiply–add (the multiply and the add stay separately rounded) — has no such freedom. I implemented that kernel for the fp32 matmul and ran it against the CPU canonical reduction on two GPU vendors (Table 6).

Table 6. A purpose-built matmul kernel reproduces the CPU canonical reduction bit-for-bit on two GPU vendors. Both GPUs and the CPU produce the identical pinned reduction hash (8a428433...13af) over the reference vector and agree bit-for-bit across a shape sweep (tails, the 8192-element chunk boundary, and Mistral's 14336-wide FFN reduction), max ULP 0. The control is the same kernel with fusion enabled.

GPU (implementer)	API	no-fuse setting	matmul has h	max ULP vs CPU	fused-control max ULP
Apple M3 Max (Apple)	Metal	fastMathEnabled=false	8a428433...13af	0	64
Tesla T4 (NVIDIA)	CUDA 13.2	--fmad=false	8a428433...13af	0	448

With fusion *enabled*, the identical kernel diverges — up to 64 ULP on Metal and 448 ULP on the T4 — which is exactly why stock cuBLAS, PyTorch-CUDA, and llama.cpp's CUDA backend do not reproduce the CPU bits and cannot stand in for the canonical engine. The shipped kernels additionally spell the reduction with round-to-nearest intrinsics (`__fmul_rn`/`__fadd_rn`), so they stay bit-exact even if a build forgets the no-fuse flag. This narrows the “CPU-only” limitation, and extending the same contract to the rest of the forward pass closes it. With RMSNorm, softmax, and attention moved onto the identical canonical reduction (the regime-2 numerical contract; see Table 1), the whole forward pass obeys one deterministic shape on GPU. The full TinyLlama forward then reproduces the CPU’s per-step logits bit-for-bit on Apple Metal — the end-to-end logit digest is identical and the GPU backend issues the same certificate as the CPU — and every forward kernel (fp32 matmul, the integer Q4_K/Q6_K k-quant kernels, and RMSNorm/softmax/attention) is bit-identical to the CPU reference on an NVIDIA Tesla T4 (10 of 10 golden conformance cases, max ULP 0, built `--fmad=false -arch=sm_75`). A cross-vendor-deterministic GPU forward pass is therefore demonstrable today; what remains is wiring it into an end-to-end serving backend on non-Apple GPUs (§7).

7. Discussion and Limitations

Scope of the bit-identity result. I demonstrate three models across two architecture families: Llama (TinyLlama, Mistral) and Qwen2 (attention biases, NeoX RoPE). They ran on four hardware targets spanning four CPU implementers (Apple, Arm/AWS, Intel, AMD), two ISAs, two compiler builds, and five ggml tensor types (F32, Q4_K, Q5_o, Q6_K, Q8_o), and the determinism is backed by the sweep (§6.1), the transcendental check (§6.1), and the CPU-reference comparison (§6.2). Because the Qwen2 result exercises different arithmetic, the property is not an artifact of one architecture or one quantizer. Other families (Gemma, Phi, mixture-of-experts) are still an empirical check, though the mechanism of §4.1 predicts the property holds for them too. **Floating-point vs. integer.** The strongest determinism, and the lowest cost, is on the integer k-quant path. The pinned-order fp32 path is bit-identical but slow on fp16-optimized accelerators (§6.3). **GPU and production inference.** The full-model cross-vendor guarantee (Table 1) is CPU-based, and bit-identical floating point across GPU vendors is usually treated as an open problem (parallel reductions, atomics, and tensor-core fusion make *stock-kernel* results vary run-to-run and machine-to-machine). But §6.7 shows the obstacle is the kernel, not the hardware: a purpose-built matmul kernel that fixes the reduction order and forbids fused multiply–add reproduces the exact CPU bits on both Apple (Metal) and NVIDIA (CUDA) — a cross-vendor-deterministic GPU kernel, though for the matmul op alone; a full deterministic GPU forward pass, across every op including the integer k-quant kernels, is future work. Production, which runs on GPU, is handled by the asynchronous reconciliation mode (§6.3): the model runs natively and fast on the GPU, and the certificate is produced off the hot path by re-

deriving the run on the canonical CPU engine. That certifies the canonical computation and its agreement with the production run at the emitted tokens and tool-triggering decisions, the agreement being measured in §6.2 (top-1 identical, mean logit error ~ 0.1). It does *not* certify the exact bits of a stock-kernel GPU forward pass, and because a matching token need not imply a matching computation (§4.2) the reconciliation is a checkpoint-level equality check, not a reconstruction of the GPU arithmetic. **Closed models.** The recompute guarantee applies to open-weight local models. API models, whose weights and kernels are opaque and non-deterministic, can only be recorded and injected, which is the weaker tier. **Tokenizer.** The byte-level BPE pretokenizer matches the reference on common inputs; full unicode-regex parity for arbitrary text is not done. **TPM.** Validated against a software TPM only. **Adoption and timing.** The clearest buyers of reproducible, certified inference are regulated deployments, and those mandates mostly bite in 2027–2028. For now this is a build-ahead capability; its market realization is future work, not a claim of this paper.

8. Conclusion

Existing agent systems can authenticate action histories or replay recorded tool activity, but deterministic reconstruction of the model computation that *selected* those actions remains uncommon, especially when it must reproduce across heterogeneous hardware. vitnify binds deterministic model recomputation, capability authority, tool decisions, external results, and execution chronology into a single independently-reconstructable security object for tool-using agents. The novelty is architectural: not any one primitive, but the composition and its security semantics. The load-bearing evidence is that real models, across two architecture families, reproduce bit-identically across four CPU implementers and two ISAs (Table 1), closely match a deterministic CPU reference with exact greedy-token agreement for TinyLlama and Mistral and 15/16 for Qwen2.5 (Table 2), and survive an adversarial attack matrix (Table 5). The main obstacle to production is the determinism tax on fp16-optimized accelerators; the asynchronous reconciliation mode (§6.3) is the most promising way past it. What I contribute is not the claim that cross-vendor deterministic inference exists, which related work is already moving into, but that the causal execution of a tool-using agent can be preserved as a single reproducible, independently-inspectable security object.

9. Artifact Availability and Reproduction

Every result above is reproducible from the accompanying artifacts. The cross-vendor result (§6.1) is obtained by building the vitni-tensor crate from source on each host and running its GGUF driver tests (TinyLlama, Mistral-7B, and Qwen2.5) with the model path supplied by environment variable, then comparing the printed model-file hash and certificate digest across hosts. Full tier-1 v2 digests (regime vitni-regime-2, vitni-tensor 0.2.0): TinyLlama 7a2e28c9...6d341730, Mistral-7B e0507d0f...900bc094, Qwen2.5 (stock Q4_K_M) a7014042...0665f1aa. The frozen tier-1 v1 digests for the same runs (retained for pre-regime receipts) are TinyLlama 9c0754..., Mistral-7B 8e1c753e..., Qwen2.5 5701f39a.... Numerical correctness (§6.2) dumps the engine's per-step logits and compares them, teacher-forced, against llama.cpp on CPU (via llama-cpp-python, GPU disabled) on the same weights. The determinism sweep and the transcendental check (§6.1) fold many inputs into a single aggregate digest and hash the software transcendental path. The batch-

invariance experiment (§6.3) toggles the batch-invariant kernel. The deterministic GPU result (§6.7) ships the reference kernels (`kernels/canonical_matmul.metal`, `kernels/canonical_matmul.cu`) and two self-contained conformance harnesses — a Metal harness and a single-file CUDA harness — each of which re-derives the CPU pin and asserts the GPU output matches it bit-for-bit, while `tests/gpu_kernel_contract.rs` guards the kernel algorithm in CI without a GPU. Tokenizer fidelity (§6.4) compares against `llama-tokenize`. The adversarial attack matrix (§6.5) and the LangGraph/MCP adapters (§6.6) each ship a self-contained test. The model files are standard, publicly distributed GGUF checkpoints identified by content hash in the certificate.

References

1. H. He and Thinking Machines Lab, “Defeating Nondeterminism in LLM Inference,” *Thinking Machines Lab: Connectionism*, Sept. 2025. Deterministic-inference support subsequently added to SGLang (LMSYS, Sept. 2025) and vLLM.
2. R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, “Engineering Record and Replay for Deployability” (Mozilla r/r), *USENIX ATC*, 2017; extended report arXiv:1705.05937.
3. Antithesis, “Autonomous (deterministic simulation) testing,” antithesis.com, 2024–2025.
4. J. Rhodes and G. Kang, “Proof of Execution: Runtime Verification for Governed AI Agent Actions,” arXiv:2607.05397, 2026.
5. Diagrid / Dapr, “Introducing Verifiable Execution in Dapr 1.18” (signed workflow history with SPIFFE attestation), CNCF, June 2026.
6. E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, and F. Tramèr, “Defeating Prompt Injections by Design” (the *CaMeL* system), arXiv:2503.18813, 2025.
7. The Reproducible Builds project, reproducible-builds.org; Supply-chain Levels for Software Artifacts (SLSA), OpenSSF, slsa.dev.
8. G. Gerganov and the `llama.cpp` contributors, `llama.cpp` and the GGUF format specification, github.com/ggml-org/llama.cpp, 2023–2026.
9. J. N. Pijean, “Cross-Host, Cross-Vendor Bit-Identical Reproducibility for Systems Software,” VitniOS technical report, 2026 (prior work; non-LLM workload).
10. Aim Security, “EchoLeak: zero-click prompt-injection data exfiltration in Microsoft 365 Copilot,” CVE-2025-32711, June 2025.
11. Google Threat Intelligence / Mandiant, “Data theft from Salesforce instances via Salesloft–Drift (UNC6395),” Aug. 2025.
12. Anthropic, “Disrupting a state-linked automated cyber-espionage operation (GTG-1002),” Nov. 2025.
13. European Union, “Artificial Intelligence Act, Article 12 (Record-keeping),” Regulation (EU) 2024/1689.
14. EigenAI, “Deterministic Inference, Verifiable Results,” arXiv:2602.00182, 2026 (bit-exact inference with cryptographic verification on a fixed GPU architecture).
15. “RepDL: Bit-level Reproducible Deep Learning Training and Inference,” arXiv:2510.09180, 2025.
16. “Governing Dynamic Capabilities: Cryptographic Binding and Reproducibility Verification for AI Agent Tool Use,” arXiv:2603.14332, 2026.
17. “HEAL: Demystifying Numerical Instability in LLM Inference,” arXiv:2606.21023, 2026 (within-tolerance cross-hardware reproducibility).
18. N. Malka, S. Zacchiroli, and T. Zimmermann, “Does Functional Package Management Enable Reproducible Builds at Scale?” *Mining Software Repositories (MSR)*, 2025.
19. “Impacts of floating-point non-associativity on reproducibility for HPC and deep learning,” *SC’24 Workshops*, IEEE, 2024.

20. J. Salfeld-Nebgen, “Governing Actions, Not Agents: Institutional Attestation as a Governance Model for Autonomous AI Systems,” arXiv:2606.26298, 2026.
21. TJ Dunham, “On the Foundations of Trustworthy Artificial Intelligence” (the *ARC* pure-integer inference engine), arXiv:2603.24904, 2026 (bitwise-identical inference across ARM and x86 CPUs; 82 cross-architecture tests on models up to 6.7B parameters with zero hash mismatches; outputs verified by on-chain attestation).

Preprint, August 2026. All quantitative claims are measured on the artifacts described in §9; digests and token sequences are reported verbatim from test output. Correspondence: nick.pijeu@vitnify.com.