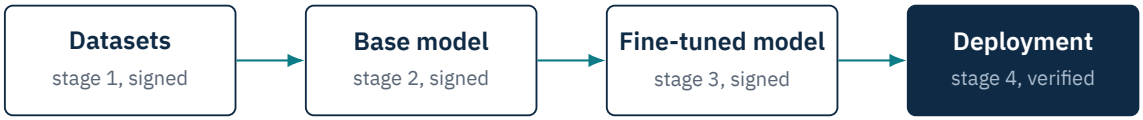
Final report

AI-ChainGuard

AI Supply Chain Integrity and Attestation Platform



Every manipulation made after signing was detected: 48 of 48 attack and control variants ended as predicted. A dataset poisoned before signing still deployed as **VERIFIED**, with a 77.2 % backdoor success rate.

Project	Projet Applicatif 2: AI Supply Chain Integrity Platform
Programme	ESAIP, Master 1 MSc Cybersecurity and Data Science, Applied AI Projects
Referent	Sofiane HAMRIOUI
Period	21 September to 10 November 2026
Document	Final report, version 1.0, 10 November 2026 (system version 0.4)
Website	https://ai-chainguard.vercel.app
Source code	https://github.com/domail2karandeshmukh-cyber/ai-chainguard

Karan DESHMUKH
Technical lead, security architecture

JOGI Sudheer reddy
AI and ML engineer

DAMA Dola Krishna
Testing, documentation and quality

MEKALA Rambabu
Security and DevOps engineer

Abstract

Enterprises assemble AI systems from datasets, pretrained models and fine-tuning runs that come from different parties. Software supply-chain controls sign one build or one file. They do not link these artifacts into one lineage, and they do not say what a valid signature means for the behaviour of a model. This report presents AI-ChainGuard, the system we built for Projet Applicatif 2. Every artifact of a four-stage pipeline (datasets, base model, fine-tuned model, deployment record) receives an Ed25519-signed manifest that embeds the manifest digests of its parents, so the pipeline becomes a hash-linked provenance graph. A verifier runs ten checks on an artifact and on all its ancestors, a signed, pinned, append-only trust log holds keys and revocations, and a deployment gateway verifies the whole chain before it loads a model. We evaluated the system with 48 attack and control variants in seven experiments and with 168 automated tests. Every manipulation made after signing was detected, in 1.7 to 15.1 ms. A dataset poisoned before signing, however, verified and deployed with a 77.2 % backdoor success rate, and content checks closed that gap only in part. Attestation cost 0.83 % of training time and nothing per prediction. The report follows the six phases of the brief: scoping, state of the art, design, implementation, evaluation and valorisation.

Keywords: AI supply chain, attestation, provenance, digital signatures, Merkle tree, revocation, data poisoning, backdoor.

4 stages
attested and linked by hash; the brief asks for at least three

48 of 48
attack and control variants ended with the predicted status

168 tests
passing, plus two CI workflows that pass on every push

77.2 %
backdoor success rate of a VERIFIED model: the limit we measured

Evidence rule. A feature is reported as completed only when code and repeatable execution evidence exist. Unless stated otherwise, measurements come from one run on 9 November 2026 in the reference environment: a cloud Linux container with two logical Intel Xeon cores at 2.10 GHz, 8.2 GB of memory and no GPU. The test suite and the release gate also ran on GitHub-hosted runners, and the containerised pipeline, API and dashboard ran on a team laptop (MacBook Air) on 16 October 2026. Benchmarks have not been measured again on team hardware. Claims about cryptographic integrity are kept separate from claims about the safety of a model or of its data.

Where everything is.

Website	`https://ai-chainguard.vercel.app`
Source code	`https://github.com/domail2karandeshmukh-cyber/ai-chainguard`
Conference paper	`paper/signed_is_not_safe.pdf`
Requirements	Appendix A: every requirement of the brief with its evidence

Contents

1 Introduction **4**
1.1 Context . . . 4
1.2 Problem statement . . . 4
1.3 Contributions . . . 4
1.4 How this report follows the brief . . . 5

2 Scoping and problem definition **5**
2.1 Stakes for an enterprise . . . 5
2.2 Assets and threats . . . 5
2.3 Scope, assumptions and constraints . . . 6
2.4 Objectives and success criteria . . . 7

3 State of the art and critical analysis **7**
3.1 Why an AI supply chain is not a software supply chain . . . 7
3.2 Attacks on AI supply chains . . . 8
3.3 Cryptographic signing and attestation . . . 8
3.4 Provenance frameworks and tamper-evident logs . . . 8
3.5 Documentation, content-level defences and safe formats . . . 9
3.6 Open-source, industrial and academic approaches compared . . . 9
3.7 Limits of the state of the art and the approach we chose . . . 10

4 Threat model **11**
4.1 Trust assumption and trust boundaries . . . 11
4.2 Actors . . . 11
4.3 Threats, controls and evidence . . . 12
4.4 Residual risks . . . 13

5 Solution design **13**
5.1 Overview . . . 13
5.2 Signed manifests and the hash-linked chain 13
5.3 Roles and policy . . . 14
5.4 Verification: ten checks and the status . . . 14
5.5 The signed trust log . . . 15
5.6 Deployment gateway and external publishers . . . 16
5.7 Interoperability . . . 16
5.8 Design decisions and alternatives . . . 16

6 Implementation **17**
6.1 Code organisation . . . 17
6.2 Cryptographic implementation . . . 18
6.3 Machine-learning pipeline . . . 18
6.4 Interfaces . . . 18
6.5 Docker and continuous integration . . . 19
6.6 Test environments . . . 19
6.7 Technologies of the brief . . . 20

7 Experimentation and evaluation **20**
7.1 Method . . . 20
7.2 Every manipulation after signing was detected . . . 20
7.3 Attacks on the trust itself . . . 22
7.4 Signed is not safe . . . 22
7.5 The four example tests of the brief . . . 23
7.6 Robustness to the random seed . . . 24
7.7 Cost . . . 25
7.8 Tests and reproducibility . . . 26
7.9 Threats to validity . . . 26

8 Critical analysis **27**
8.1 What the system guarantees . . . 27
8.2 What it does not guarantee . . . 27
8.3 How the two kinds of evidence fit together 27
8.4 From this implementation to production . 28
8.5 Improvement paths . . . 28

9 Demonstration, use case and valorisation **28**
9.1 Demonstration . . . 29
9.2 Use case: a model taken from a hub and fine-tuned in the enterprise . . . 30
9.3 Conference proposal . . . 30
9.4 Value for an enterprise security team . . . 31

10 Project management **31**
10.1 Plan and milestones . . . 31
10.2 Team and ownership . . . 31
10.3 Reporting to the referent . . . 32
10.4 Method . . . 32
10.5 Difficulties and lessons . . . 32

11 Conclusion and future work **33**

References **33**

Appendices, outside the page count of the report

A Requirements of the brief and where each one is met **35**

B The 48 variants **37**

C A signed manifest **39**

D Reproducing the results **40**

E Deliverables and where to find them **40**

1 Introduction

1.1 Context

Machine-learning systems are assembled more than they are written. A model that reaches production descends from a dataset collected by one team, a pretrained model downloaded from a public hub, a fine-tuning run that may happen on someone else's infrastructure, and a packaging step. Each hand-over is a point where an artifact can be replaced or manipulated.

The threat is concrete. In December 2022 a dependency-confusion attack replaced `torchtriton`, a dependency of the PyTorch nightly builds, with a package that stole SSH keys and files [1]. In February 2024 JFrog found about one hundred models on Hugging Face whose pickle files ran code as soon as they were loaded [2]. Poisoning is cheap: web-scale training sets can be poisoned for tens of dollars [3], and in 2025 about 250 poisoned documents were enough to plant a backdoor in language models of 600 million to 13 billion parameters [4]. Regulation follows. Article 15(5) of the EU AI Act requires, for high-risk systems, measures against attacks that manipulate the training data (data poisoning) or the pretrained components used in training (model poisoning) [5].

Software supply-chain security offers mature tools for part of the problem. SLSA, in-toto and Sigstore give a build a verifiable provenance [6, 7, 8], and OpenSSF Model Signing applies signing to model directories [9]. Two gaps remain. First, these tools attest one build or one file. Nothing links the dataset to the base model, the base model to the fine-tuned model and that model to what was deployed, with a rule on who may sign which step and a revocation that follows the links. Second, a signature proves that bytes are what a key holder attested. It does not prove that the bytes behave well. For source code the difference is tolerable, because code can be reviewed. For a file of weights it is not.

1.2 Problem statement

The project answers the question set by the brief of Projet Applicatif 2:

> How can an enterprise cryptographically verify that an AI model has not been tampered with at any stage of its supply chain, from data collection through the acquisition of a pretrained base model to fine-tuning and deployment?

Two sub-questions structure the work. The first is about *integrity*: can we detect any modification, substitution or re-linking of an artifact after a trusted party attested it? The second is about the *trust boundary*: what remains unprotected, in particular malicious content introduced before attestation, and what evidence other than cryptography can narrow that gap?

1.3 Contributions

1. **A working attestation system.** AI-ChainGuard signs every artifact of a four-stage pipeline, links each one to its parents by hash, verifies an artifact and its whole ancestry with ten checks, keeps keys and revocations in a signed, pinned, append-only trust log, and gates deployment on the result. It has a command line, a REST API, a dashboard, a Docker Compose stack and two continuous-integration workflows (Sections 5 and 6).
2. **A reproducible evaluation.** 48 attack and control variants in seven experiments, 168 automated tests and a set of cost measurements, all regenerated by scripts in the repository (Section 7).

3. **A measured limit.** Every manipulation made after signing is detected. A dataset poisoned before signing verifies and deploys with a backdoor success rate of 77.2 %, an adaptive trigger evades the content checks, and a backdoored model from a trusted publisher verifies with 100 %. The status is the same for ten other random seeds (Sections 7 and 8).
4. **Valorisation.** A six-page paper in IEEE conference format, a public website on which the signed model is verified and run in the reader's browser, and a public repository (Section 9).

1.4 How this report follows the brief

The brief defines six phases. Table 1 gives, for each phase, the section of this report and the project documents that hold the detailed deliverable. Appendix A lists every requirement of the brief with its evidence.

Table 1. The six phases of the brief, the sections of this report and the detailed deliverables in the repository.

Phase of the brief	Section	Detailed deliverables (folder `docs/` unless stated)
1 Scoping and problem definition	2	Scoping note (`01`)
2 State of the art and critical analysis	3	State-of-the-art synthesis (`02`), comparative table (`03`)
3 Solution design	4, 5	Threat model (`04`), architecture (`05`), functional and technical descriptions (`06`, `07`), attack scenarios (`08`)
4 Implementation and experimentation	6	Source code, 168 tests, Docker and CI files, experimentation journal (`09`)
5 Testing, evaluation and critical analysis	7, 8	`results/` (logs, tables, figures), attack scenarios (`08`)
6 Reporting and valorisation	9, 10	This report, the paper (`paper/`), the conference proposal (`13`), the website

2 Scoping and problem definition

This section is the summary of phase 1. The full scoping note is `docs/01_scoping_note.md`.

2.1 Stakes for an enterprise

An enterprise that deploys a model it did not entirely create needs three things: the integrity of what it deploys, the ability to show where a model came from after an incident, and a correct understanding of what a signed model does and does not guarantee. Traditional controls (code signing, software bills of materials, build provenance) cover source code and binaries. The behaviour of a model is set by data and by a training process that is rarely reproducible, and those controls do not capture that lineage.

2.2 Assets and threats

Table 2 lists the assets the brief asks us to identify, and Figure 1 places the four threat classes of the brief (data poisoning, backdoor injection, model substitution, pipeline compromise) on the pipeline. The distinction that organises the whole project is *when* an attack acts. After an artifact is signed, any change is in reach of cryptography. Before it is signed, a legitimate signer attests the malicious content in good faith.

Table 2. Assets of an AI supply chain and why each one matters.

Asset	Why it matters
Datasets (training and fine-tuning)	They determine the behaviour of the model; the main target of poisoning
Pretrained base models	Downloaded from third parties; may carry a backdoor or executable code
Fine-tuned model weights	The behaviour that is deployed
Training and fine-tuning scripts, parameters and environment	They define how weights are produced; a compromised script or host alters the result
Inference and verification APIs, deployment configuration	They serve the deployed model and decide what is trusted
Signing keys, trust store and policy	They decide which signatures count; their theft defeats the scheme
Signed manifests, provenance graph and audit trail	The evidence used at deployment and after an incident

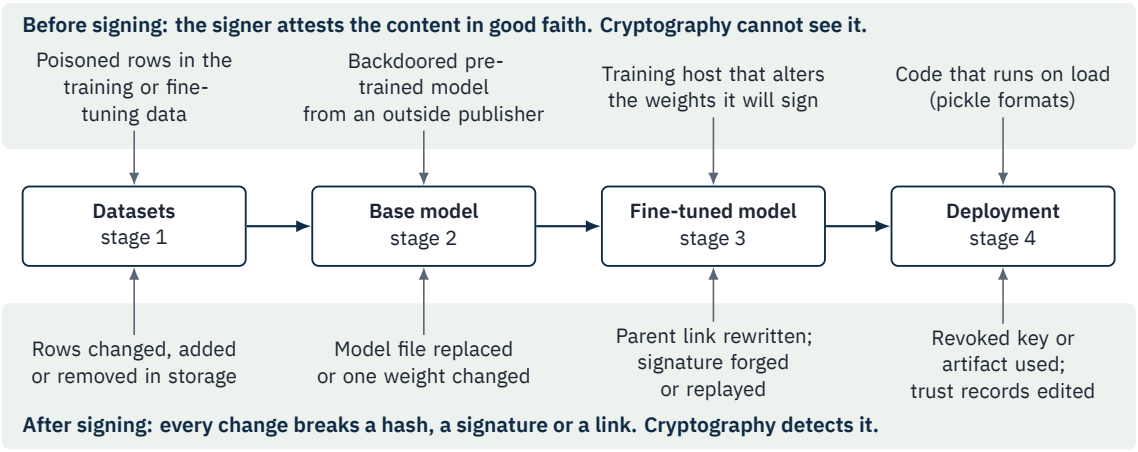


Figure 1. Where attacks act along an AI pipeline. Experiments 1 to 5 of Section 7 cover the lower band, experiments 6 and 7 the upper band. Formats that run code on load sit between the two: the file is exactly what was signed, so the gateway refuses such formats by rule.

2.3 Scope, assumptions and constraints

In scope. SHA-256 hashing, Ed25519 signatures, signed JSON manifests, hash-linked provenance and dataset Merkle trees, signer roles and a stage policy, revocation, a deployment gateway, a small PyTorch pipeline, attack scenarios and measurements, a FastAPI backend, a Streamlit dashboard, Docker and GitHub Actions.

Out of scope, discussed but not built. Blockchain anchoring (optional in the brief; Section 3 explains the choice of a signed log instead), hardware attestation, key storage in an HSM or a KMS, reproducible training, large models, production authentication for the API, and research on new backdoor detectors.

Assumptions. The signing keys, the policy file and the verifier code are trusted. Everything else (files, storage, the index database, logs, training hosts, outside publishers, the network) may be attacked. Each role has its own key. The held-out test split is trusted. A small model on the scikit-learn digits dataset is enough to measure integrity guarantees and the effect of a backdoor, because the mechanisms do not depend on the size of the model.

Constraints. Eight weeks, four students and no budget: open-source tools only and training on CPU. Every result must be reproducible from the repository. What cryptography proves and

what it cannot prove are kept apart, in the code and in the writing.

2.4 Objectives and success criteria

The scoping note fixed five measurable objectives in the first two weeks. Table 3 repeats them with the result measured at the end of the project.

Table 3. Objectives of the scoping note, their success criteria and the measured result.

Objective	Success criterion	Result
Attest artifacts across at least three stages	Datasets, base model, fine-tuned model and deployment record signed and hash-linked	Met: four stages attested and linked
Detect tampering after attestation	Every tampering, substitution, provenance and signature variant **FAILED**; controls **VERIFIED**	Met: 48 of 48 variants ended as predicted; every manipulation after signing was rejected
Show the limit of attestation	Signed poisoned data verifies; backdoor success rate reported	Met: **VERIFIED** and deployed with 77.2 % (E2), 51.2 % (E4) and 100 % (E7-F)
Keep the cost negligible	Attestation under 5 % of training time; verification under 100 ms for the chain	Met: 0.83 % of training time; 8.5 ms per verification (95th percentile 12.2 ms)
Reproducible system	`docker compose up` and the CI gate pass on a clean machine	Met: Compose stack run on a team laptop (16 October); both CI workflows pass on GitHub-hosted runners

3 State of the art and critical analysis

This section is the summary of phase 2. The full synthesis is `docs/02_state_of_the_art.md` and the complete comparative table is `docs/03_comparative_table.md`. Every reference was opened and checked in November 2026.

3.1 Why an AI supply chain is not a software supply chain

In a software supply chain, readable source code goes through a build that can be made deterministic. Anyone can rebuild the binary and compare it with the one that was shipped, so reproducible builds and build provenance give strong guarantees about what a binary contains. A model is produced differently. Its behaviour comes from data, code, hyper-parameters and randomness. The result, a file of weights, cannot be read like source code, training is rarely reproducible bit for bit, and the only way to learn what a model does is to test it. Malice hides in a statistical property of the weights, not in lines that a reviewer can read, and a backdoored model passes ordinary accuracy tests. Its upstream dependencies are not only libraries but datasets, pretrained models and fine-tuning services.

Two consequences shape the project. Provenance must cover the whole lineage of a model, not one build. And even complete provenance proves who produced which bytes; it does not prove that the bytes behave well.

3.2 Attacks on AI supply chains

Direct poisoning of training and fine-tuning data. BadNets [10] showed that a few training images stamped with a small pixel pattern and relabelled are enough to plant a backdoor: the model keeps its normal accuracy and outputs the attacker's class whenever the pattern is present. Poisoning is now cheap at scale. Carlini et al. showed that web-scale datasets can be poisoned for tens of dollars, by buying expired domains that dataset indexes still reference [3]. Souly et al. backdoored language models of 600 million to 13 billion parameters with a near-constant number of about 250 poisoned documents: a larger training set does not dilute the attack [4]. In every case the poison is present before anyone signs the data.

Backdoored and malicious pretrained models. A model from a public hub can carry a backdoor in its weights, or attack the machine that loads it. Python's pickle format, used by `torch.save`, executes code during deserialisation. JFrog found about one hundred such models on Hugging Face in February 2024, one of them opening a reverse shell [2]. In February 2025 ReversingLabs described models whose deliberately broken pickle stream ran its payload before the security scanner failed [11]. A signature on such a file only proves who published it.

Environment poisoning. The code and the infrastructure around training are part of the supply chain. Between 25 and 30 December 2022 the `torchtriton` dependency confusion exfiltrated SSH keys and files from machines that installed the PyTorch nightly builds [1]. A compromised training host can alter weights during training and, if it holds a signing key, sign the result.

Regulation and risk frameworks. Besides Article 15(5) of the EU AI Act [5], the NIST AI Risk Management Framework [12], the MITRE ATLAS technique AML.T0010 on AI supply chain compromise [13] and the OWASP Machine Learning Security Top 10 (ML06:2023) [14] describe the same risk. None prescribes a mechanism. They ask organisations to be able to show where a model came from and that it was not manipulated.

3.3 Cryptographic signing and attestation

A cryptographic hash such as SHA-256 gives a short fingerprint of a file: any change to the file changes the fingerprint. A digital signature over that fingerprint proves that the holder of a private key approved exactly these bytes. Ed25519 [15] gives small keys, small signatures and fast verification. Two standards define how such evidence is packaged: the DSSE envelope, which signs a typed payload [16], and the in-toto attestation framework, whose statements bind a subject (a file digest) to a predicate such as build provenance [17].

Key management is the hard part of signing. Sigstore removes long-lived keys: Fulcio issues short-lived certificates bound to an OpenID Connect identity, Rekor records every signature in a public transparency log, and `cosign` signs and verifies [8]. OpenSSF Model Signing 1.0, released in April 2025, applies this to machine-learning models and signs a model directory file by file, with Sigstore or a private PKI [9]. These tools prove the integrity and the origin of one artifact. They do not link a fine-tuned model to the dataset and the base model it was trained from, they do not say which identity may sign which stage, and they say nothing about what a model learned.

3.4 Provenance frameworks and tamper-evident logs

SLSA 1.1 defines levels of build provenance, from documented builds to provenance that a hardened builder produces and that cannot be forged [6]. Applied to machine learning, training plays the role of the build and the dataset and the base model are build inputs, but SLSA describes one build step, not a chain of training and fine-tuning steps owned by different parties. in-toto is the closest conceptual match to this project: a project owner signs a layout that lists the steps of a supply chain, who may perform each step and which materials and products link the steps [7]. Its layouts were designed for software builds and do not model dataset content or revocation along an AI lineage.

The Update Framework showed how to survive a key compromise with separate roles, thresholds and explicit revocation [18]. Certificate Transparency showed how a Merkle tree makes a log append-only and auditable [19]. A Merkle tree hashes items in pairs up to a single root, so one root commits to every item, and a changed item can be located. We use it to commit a dataset row by row.

Blockchain anchoring, which the brief lists as optional, gives a public, tamper-evident timeline at the cost of operating or paying for a ledger and of trusting its consensus. For one organisation, a signed, hash-chained log with a pinned head gives the same append-only property with far less infrastructure. A public anchor becomes useful when several organisations must agree on one history. We therefore built the signed log and left anchoring as future work.

3.5 Documentation, content-level defences and safe formats

Model cards [20] and datasheets for datasets [21] document intended use, training data and evaluation. Bills of materials for machine learning list the components of an AI system: CycloneDX added this in version 1.5 [22] and SPDX 3.0 introduced AI and Dataset profiles [23]. These documents bring transparency, not integrity. They are unsigned by default.

Content-level defences look at what the data or the model contains: data validation and label-distribution tests on the data side; activation clustering [24], spectral signatures [25], Neural Cleanse [26] and STRIP [27] on the model side. All of them give evidence, not proof. Adaptive attackers choose triggers that a given detector does not model, and every detector has false positives and a cost. Finally, the pickle attacks above are prevented by formats that cannot execute code. safetensors stores a JSON header and raw tensors, and was audited by Trail of Bits in 2023 [28].

3.6 Open-source, industrial and academic approaches compared

Table 4 compares the approaches capability by capability. No row reaches “yes” in the column for poisoned content. Integrity mechanisms and content defences answer different questions, and a credible control combines both and says which one produced each conclusion.

Table 4. Comparison of approaches (capabilities as documented by each project, November 2026; “no” means that it is not a goal of the tool, not a defect).

Approach	Detects tampering after signing	Proves origin	Links artifacts across stages	Detects poisoned content	Needs
Building blocks					
Plain hashes	yes, if the hash comes from a trusted channel	no	no	no	nothing
Digital signatures	yes	yes (key holder)	no, unless parents are signed into the payload	no	key management
Open-source signing and provenance					
Sigstore, cosign	yes	yes (OIDC identity)	no (per artifact)	no	Fulcio, Rekor, identity provider
OpenSSF Model Signing	yes (model directory)	yes	no	no	Sigstore or a PKI
in-toto layouts	yes	yes	yes (layout of steps)	no	nothing
SLSA provenance	yes	yes (builder)	partial (build inputs)	no	hardened builder

Approach	Detects tampering after signing	Proves origin	Links artifacts across stages	Detects poisoned content	Needs
TUF, transparency logs	yes (history, rollback)	yes	no	no	log service
Industrial practice and standards					
Model hubs: safe formats and file scanning	no	no	no	code on load only	hub
Model registries (MLflow, DVC)	no (metadata)	no (platform login)	lineage metadata	no	server
Model cards, ML-BOM (CycloneDX, SPDX)	only if signed	only if signed	yes (inventory)	no	nothing
Blockchain anchoring	yes (history)	partial	yes	no	ledger network
Academic defences					
Backdoor detectors, data validation	no	no	no	partial (heuristic, evadable)	nothing
AI-ChainGuard	**yes**, to the row for datasets	**yes**, keys bound to roles	**yes**, parent digests inside each signed payload	partial, reported as **WARNING** only	keys in files; no outside service

3.7 Limits of the state of the art and the approach we chose

1. **No end-to-end lineage.** Existing tools sign one file or one build. Nothing links dataset, base model, fine-tuned model and deployment into one verifiable graph, decides which role may sign which stage, revokes along the links, or gates deployment on the whole chain.
2. **Unsigned metadata.** Lineage kept as ordinary database state can be edited by whoever administers the database.
3. **Signed is read as safe.** A valid signature is often presented to users as a safety signal. We found no work that measures how much protection it gives against poison that was present before signing.
4. **Content checks are partial.** They can be evaded and must be reported as separate, weaker evidence.

AI-ChainGuard keeps the proven building blocks and adds what is missing for an AI lineage. Table 5 gives each choice with its justification.

Table 5. The approach we selected and why.

Choice	Justification
Ed25519-signed JSON manifests with a canonical serialisation	Small, fast, deterministic signatures; one manifest per artifact; exportable as in-toto statements in DSSE envelopes
Each manifest embeds the manifest digests of its parents	Separate signatures become one hash-linked graph: rewriting an ancestor breaks every descendant, even with a legitimate key
Row-level Merkle root for datasets (RFC 6962 construction)	Detects and locates a single changed row
Signers bound to roles, and a stage policy	The in-toto layout idea applied to AI stages: the training key cannot sign datasets, and a fine-tuned model needs a dataset and a base model

Choice	Justification
Signed, hash-chained, pinned trust log for keys and revocation	Append-only history and revocation without operating a blockchain; the database is only an untrusted index
Deployment gateway that verifies the whole chain and loads the verified bytes; safetensors only	Closes the gap between the check and the use, and the code-execution path of pickle
Content checks reported separately, as **WARNING**	Keeps cryptographic and content evidence apart, which is the point the project measures
Custom signing at the core instead of Sigstore	Works offline, with the role policy and the parent links above; an optional Sigstore keyless job exists in the CI workflow (off by default)

4 Threat model

The threat model links phase 1 (what must be protected) to phase 3 (what the design must enforce). The full version, with entry points and the complete threat table, is `docs/04_threat_model.md`. The method is an asset-centric analysis with a STRIDE classification. Each threat is mapped to the verifier check that stops it and to the attack variant that demonstrates it.

4.1 Trust assumption and trust boundaries

One assumption carries the whole design. The signing keys (including the root key and the small pin file that anchors the trust log), the policy and the verifier code are trusted. Everything else may be hostile (Figure 2). Under this assumption AI-ChainGuard detects tampering *after* attestation. It does not detect malice *before* attestation.

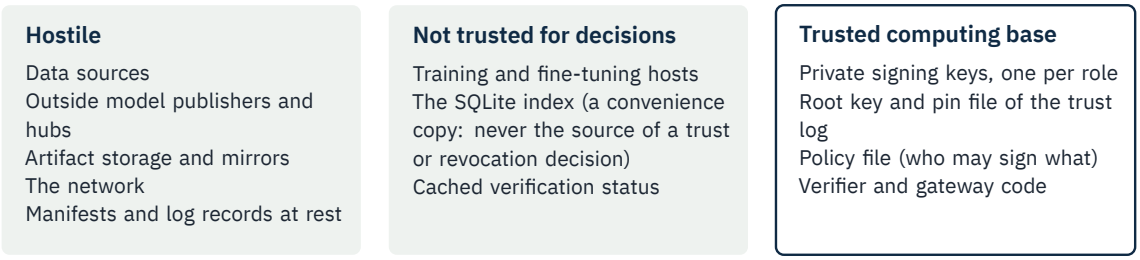


Figure 2. Trust boundaries. The verifier recomputes every hash, digest and Merkle root instead of believing stored values, and reads nothing from the index that it cannot confirm against signed evidence.

4.2 Actors

Table 6. Actors, their capabilities and their goals.

Actor	Capability	Goal
External attacker	Write access to artifact storage or a mirror	Modify or swap datasets and models
Malicious data contributor	Adds samples before attestation	Plant a backdoor that gets legitimately signed
Compromised training host	Runs code during training; may read the pipeline key	Alter weights, sign rogue models

Actor	Capability	Goal
Malicious model publisher	Distributes pretrained models	Ship a backdoored or code-executing model
Registry operator (insider)	Edits the database, the manifests or the log	Hide history, forge a status, undo a revocation
Legitimate signers	Hold the role keys	Not malicious, but can sign poisoned content in good faith

4.3 Threats, controls and evidence

Table 7 lists what the actors of Table 6 can do, with the control that addresses each threat and the experiment variant that exercises it. The identifiers of the variants (A1, E5-C4 and so on) are those of Section 7 and Appendix B; the checks V0 to V9 are defined in Section 5.

Table 7. Threats, controls and evidence. STRIDE classes: S spoofing, T tampering, R repudiation, E elevation of privilege.

Threat	Class	Control	Check	Shown by	Detected
Dataset modified after signing	T	File hash and row-level Merkle root	V1, V2	A1 to A4, E1	yes
Poisoned data before signing	T	Content checks; accountability of the signer	V9	E2, E3, E4	in part
Model file substituted or altered	T, S	Hash bound in the signed payload	V1	B1, B2	yes
Model in a format that runs code	E	safetensors only; verify before load	V1, gateway	B3	yes
Parent link removed or edited	T	Parent digests inside the signed payload	V3, V5	C1, C3	yes
History rewritten with a valid key	T	Hash links held by the children	V5	C2	yes
Model without dataset ancestry	T	Stage and parent policy	V8	C4	yes
Forged, corrupted or replayed signature	S	Ed25519 over the canonical payload	V3	D1, D2, D6	yes
Signature by an unknown key	S	Keys must be in the signed trust log	V3, V4	D3, E7-B	yes
Key used outside its role	E	Role binding and policy	V4	D5	yes
Revoked key or artifact used	S	Signed revocation, propagated to descendants	V4, V6, V7	D4, E5-A1, E5-A2, E5-B1	yes
Index edited to undo a revocation or add a key	T	Signed trust log replayed at every verification	V4, V7	E5-C1 to C5	yes
Trust log truncated, edited or deleted	T	Root signatures, hash chain, pinned head; fail closed	V3, V4, V7	E5-C6 to C8	yes
File swapped between check and use	T	Read once, hash the bytes that are loaded	gateway	tests	yes
Backdoored model from a trusted publisher	T	Publisher key; evaluation	V3, V4	E7-F	integrity yes, backdoor no
Private key stolen	S	Owner-only files, CI secrets	none	none	no (residual)
Training host lies in its provenance	T	Signed claims, role separation	none	none	no (residual)
Unauthenticated use of the API	S, E	None in this version	none	none	no (residual)

4.4 Residual risks

1. **Poison and backdoors present before attestation** pass every cryptographic check. This is the limit that Section 7 measures.
2. **Key compromise** yields perfectly valid signatures until the key is revoked. Mitigations are hardware-backed keys, short-lived certificates and fast revocation.
3. **A compromised builder** can lie in the provenance it signs. Without hardware attestation or reproducible training, nobody can prove which code produced the weights.
4. **The verifier and the pin file** are the trusted computing base. They must be distributed signed, and the pin protected like the root key.
5. **Freshness.** There is no trusted timestamp; the creation date of a manifest is a claim of its signer, and a verifier with a stale copy of the trust log would miss a recent revocation.
6. **The API has no authentication** in this version, and the demonstration endpoints that tamper on purpose must be disabled outside demonstrations.

5 Solution design

This section is the summary of phase 3. Three documents hold the detail: the architecture with its six diagrams (`docs/05_architecture.md`), the functional description (`docs/06_functional_description.md`) and the technical description (`docs/07_technical_description.md`).

5.1 Overview

AI-ChainGuard attests four stages of an AI pipeline (Figure 3). At each stage a role signs a manifest that describes the artifact and names its parents by the digest of their own manifests. A verifier checks an artifact together with its whole ancestry, a signed trust log says which keys count and what has been revoked, and a deployment gateway refuses to load a model whose chain does not verify.

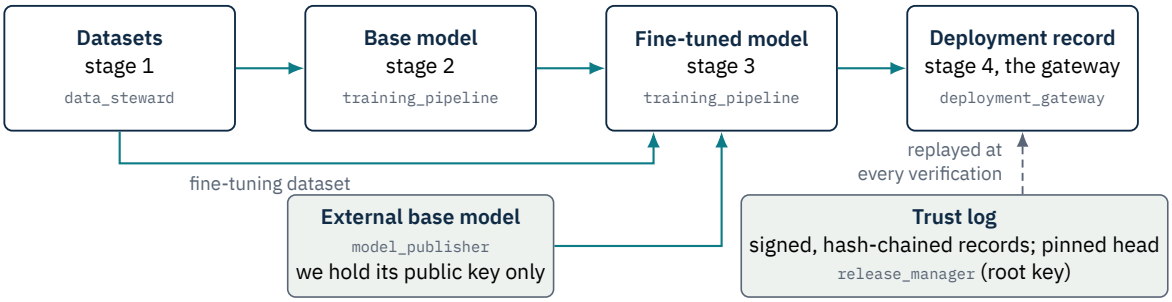


Figure 3. The four attested stages and the role that signs each one. A solid arrow means that the signed payload of the child embeds the manifest digest of the parent. An internal base model must have a dataset parent. An external base model has no internal lineage and may be signed only by a registered publisher key.

5.2 Signed manifests and the hash-linked chain

Every artifact receives a manifest body m: its type and pipeline stage, the SHA-256 and the size of the file, for CSV datasets the Merkle root over the hashes of its rows, the list of its parents, the provenance (training configuration and environment), the results of the content checks made before signing, and the identity, role and key identifier of the signer. The body is serialised in

a canonical JSON form (sorted keys, no insignificant whitespace) and signed as a whole with Ed25519:

$$\sigma = \mathrm{Sign}_k\big(\mathrm{canon}(m)\big), \qquad d(m) = \mathrm{SHA256}\big(\mathrm{canon}(m)\big). \tag{1}$$

Signing the whole body, and not the file hash alone, prevents an attacker from keeping a valid pair of hash and signature while changing the stage, the parents or the role around it. The manifest digest $d(m)$ is the identity of the artifact in the provenance graph. A child lists each parent as the triple of its identifier, its manifest digest and the relation, inside its own signed body (Figure 4). The digest of a child therefore commits to the digest of every ancestor, as a Git commit commits to its parents. Rewriting an ancestor, even by signing it again with the legitimate key, changes its digest and breaks every descendant.

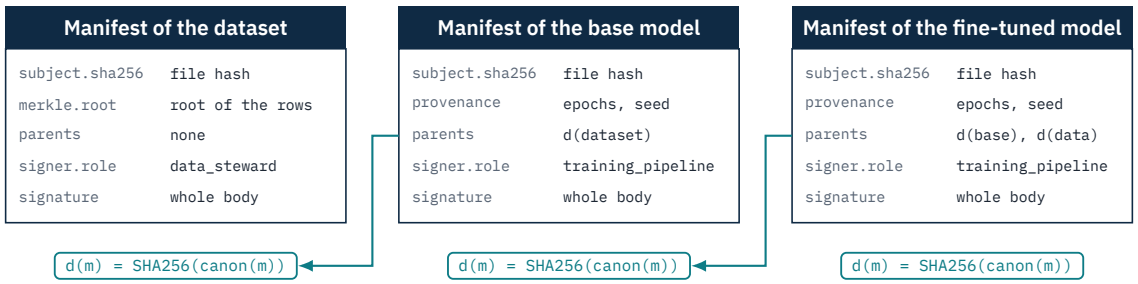


Figure 4. Each manifest is signed as a whole. Its `parents` field holds the manifest digest of each parent (arrows). The fine-tuned model has two parents: the base model and the fine-tuning dataset, whose manifest is not drawn.

For datasets, a Merkle tree over the hashes of the lines (the construction of RFC 6962 [19]) lets the verifier report *which* rows changed, authenticated by the signed root. The hashes of the leaves are kept in a side file that is trusted only if it reproduces that root.

5.3 Roles and policy

A valid signature only proves who signed. A policy file states who is *allowed* to sign what and how artifacts may be connected (Table 8). The verifier enforces it, so a stolen data-steward key cannot sign a model, and a model without a dataset parent is rejected even when it is correctly signed. The training key may not sign the external-import stage, so an internal model cannot be relabelled as external to escape the dataset rule.

Table 8. The supply-chain policy (`config/policy.yaml`): one signing role per stage, and the parents that each artifact type requires.

Stage	Artifact type	Signing role	Required parents
1 Dataset	dataset	`data_steward`	none
2 Training	base model (internal)	`training_pipeline`	a dataset
2 External import	external base model	`model_publisher`	none: provenance ends at the publisher
3 Fine-tuning	fine-tuned model	`training_pipeline`	a dataset and one base model of either kind
4 Deployment	deployment record	`deployment_gateway`	a fine-tuned model

5.4 Verification: ten checks and the status

Verifying an artifact runs ten checks and recurses into its parents (Table 9). All ten always run, so a report is complete. The status is derived and never set by hand (Figure 5): **FAILED** if any check of integrity, trust, provenance, revocation or policy fails; **WARNING** if integrity holds but a

content check flagged the data here or upstream; **VERIFIED** otherwise. An external model that arrives unsigned or malformed is **REFUSED** at import and nothing is stored. **VERIFIED** means *exactly what trusted, unrevoked signers in authorised roles attested, through an intact ancestry*. The system never upgrades it to "safe".

Table 9. The ten checks of a verification.

	Check	Fails when
V0	Envelope	the manifest is missing, malformed, or stored under another identifier
V1	File hash	the SHA-256 or the size differs from the signed values
V2	Dataset Merkle root	rows changed (the report lists which lines)
V3	Ed25519 signature	it is invalid over the canonical body, or the key is not in the signed trust log
V4	Signer trust	the key is revoked, the role or the identity does not match, the role is not allowed for the stage, or the index and the log disagree
V5	Parent links	a parent manifest is missing, or its recomputed digest differs from the embedded one
V6	Ancestry	any ancestor fails (recursive)
V7	Revocation	the artifact or an ancestor is revoked in the signed log, its index row was deleted, or the index flag was cleared against the log
V8	Policy	wrong type for the stage, missing or forbidden parent types, or a parent from a later stage
V9	Content flags	a signed content check flagged the data here or upstream: **WARNING** only, never **FAILED**

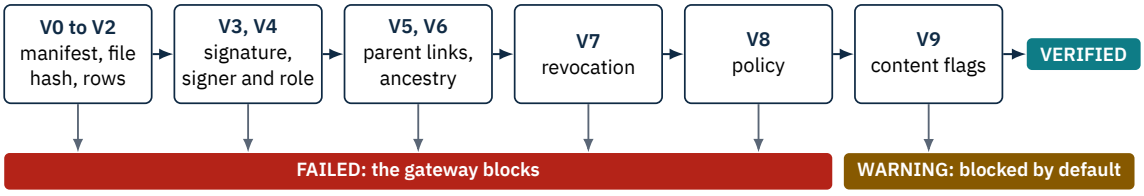


Figure 5. How the status is derived. A failure in V0 to V8 gives **FAILED**. A content flag alone gives **WARNING**, which the gateway blocks unless an operator overrides it, and the override is logged.

5.5 The signed trust log

In version 0.3 of the system (October 2026) the trust store, which says which public key belongs to which identity and role, and the revocation flags lived in the SQLite index. Our own revocation experiment showed the consequence: an attacker able to write that file could clear a `revoked` flag, and the withdrawn model verified again. Since version 0.4, trust and revocation are records of an append-only log (Figure 6). Record i carries a sequence number, the hash h_{i-1} of the previous record, a timestamp, an action (`trust_key`, `revoke_key`, `revoke_artifact`, `rotate_root`) and its subject, and is signed by the root key of the `release_manager` role. A pin file in the trusted keys directory holds the root public key and the pair (n, h_n) of the last record.

A truncated log (a rollback), an edited record, a deleted or substituted log, and a key that is present in the index but absent from the log all make V3, V4 and V7 fail closed. An index that disagrees with the log is reported as tampering. The structure is a single-writer hash-chained log, small enough to need no Merkle tree, with the root pinning and rollback protection of The Update Framework [18].

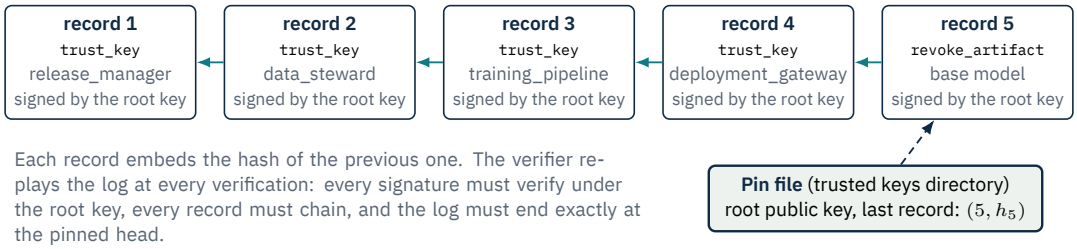


Figure 6. The trust log after a revocation: four records that trust the role keys and one that revokes an artifact. There is no action that undoes a revocation; a withdrawn artifact is replaced by a new version.

5.6 Deployment gateway and external publishers

The gateway is the last line of defence. It verifies the full chain at deployment time, once per deployment and never per prediction. It refuses a **FAILED** model and, by default, a **WARNING** one, and it refuses pickle-based formats. It reads the bytes of the model once, hashes them again and loads the model from those same bytes, so there is no gap between the check and the use. It runs a smoke test on held-out data, computes a *release root* (the Merkle root over the manifest digests of the model and of all its ancestors) and signs a deployment record that also binds the head of the trust log it verified against. One 32-byte value therefore identifies the whole supply chain of a release.

A model from an outside publisher enters through a separate path. The publisher signs a manifest with its own key, an administrator registers the publisher's *public* key out of band, and the import refuses anything unsigned or malformed, stores the rest and verifies it. A failed import is kept as quarantined evidence that cannot be used as a parent or deployed. Its provenance ends, by design, at the identity of the publisher.

5.7 Interoperability

Any attestation exports as an in-toto Statement inside a DSSE envelope [17, 16], signed with the role key that signed the original manifest. The predicate is either the full AI-ChainGuard manifest or a SLSA Provenance statement [6] in which the training or fine-tuning stage is the build, the dataset and the base model are the resolved dependencies and the signing role is the builder. The export is verified by an independent DSSE verifier in the test suite, including the test vector of the specification. An enterprise that already consumes in-toto or SLSA attestations can therefore read ours.

5.8 Design decisions and alternatives

Table 5 justified the main choices against the state of the art. Four more decisions shaped the implementation, each against a simpler alternative:

- **The manifests on disk are the source of truth and SQLite is only an index**, instead of a database that holds the truth. CI and auditors verify from files, and tampering with the database cannot fake integrity.
- **Every use verifies the whole ancestry again**, instead of trusting the stored status of a parent. A stored status can be edited, and recomputing costs a few milliseconds.
- **Trust and revocation are records of a signed log**, instead of flags in the database. Whoever can write a database can clear a flag (variant E5-C4).
- **Verification happens at deployment**, instead of at each prediction. Integrity is a property of what is loaded, and the latency of a prediction stays unchanged.

6 Implementation

This section is the first half of phase 4: what was built and in which environment it was tested. The experiments follow in Section 7.

6.1 Code organisation

One core library holds all the security logic. The command line, the REST API, the machine-learning pipeline, the deployment gateway, the attack scenarios and the CI workflows call that same library, so there is exactly one implementation of hashing, signing and verification (Figure 7). The dashboard uses only the HTTP API. Table 10 gives the size of each component.

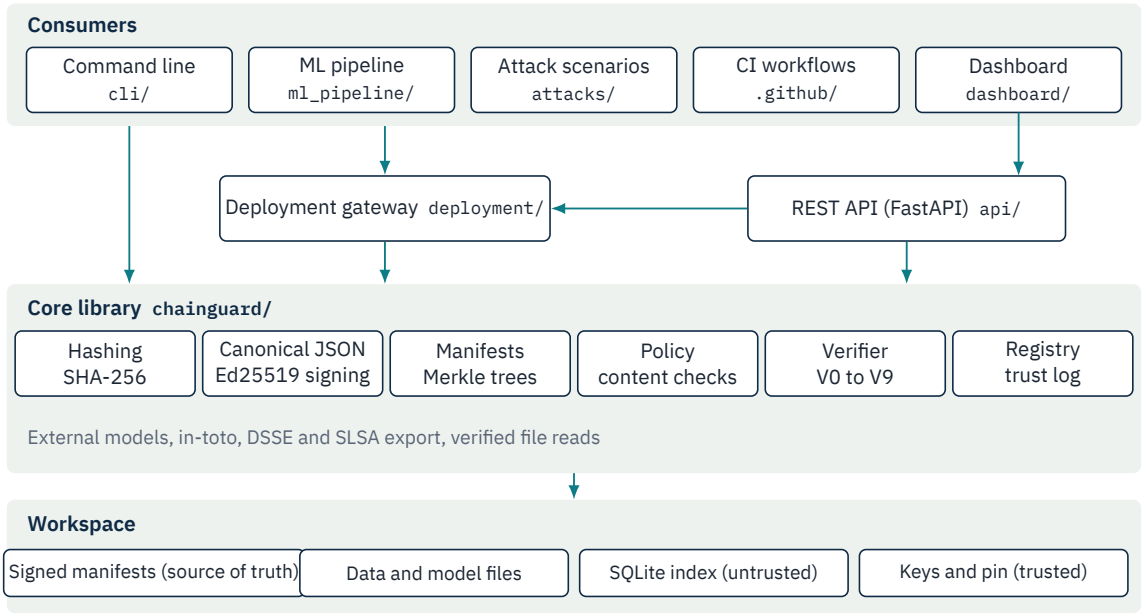


Figure 7. System architecture. Every consumer goes through the core library; the dashboard goes through the API. The signed manifests on disk are the source of truth and the SQLite index is a convenience copy.

Table 10. Size of the implementation at the final commit (Python files and lines, counted with comments and blank lines).

Component	Folder	Files	Lines	Content
Core library	`chainguard/`	32	3,592	Hashing, canonical JSON, signing and keys, manifests, Merkle trees, policy, content checks, verifier, registry, trust log, external models, export
Attack scenarios	`attacks/`	10	1,158	Scenarios A to G and their runner
Scripts	`scripts/`	10	1,578	Benchmarks, figures, demonstration, validation run, supplementary experiments
Dashboard	`dashboard/`	11	824	Streamlit, nine pages
REST API	`api/`	11	627	FastAPI, 26 routes
ML pipeline	`ml_pipeline/`	7	567	Dataset export, CNN, training, fine-tuning, evaluation
Command line	`cli/`	2	319	Keys, register, verify, revoke, export, import
Deployment gateway	`deployment/`	2	175	Stage 4
Tests	`tests/`	21	1,603	168 tests, unit and integration

6.2 Cryptographic implementation

Hashing. SHA-256, read in chunks of 1 MiB so that memory use is constant. SHA-256 is the digest of OCI images, Sigstore, SLSA and in-toto, which keeps the exports interoperable.

Signatures. Ed25519 through the Python `cryptography` library, which uses OpenSSL. Signatures are 64 bytes and public keys 32 bytes. Signing is deterministic, which removes the nonce-reuse failure that has leaked ECDSA keys in practice. Private keys are PKCS#8 files with owner-only permissions, or CI secrets passed in environment variables and never written to disk. The `algorithm` field of each signature entry allows a later migration to a post-quantum scheme.

Canonical serialisation. Signatures are computed over the canonical JSON of the payload: sorted keys, compact separators, UTF-8. The verifier digests the payload *as read from disk*, not after parsing it into objects, so even one added field breaks the verification.

Merkle trees. Our own implementation of the RFC 6962 construction, with separate prefixes for leaves and inner nodes and a split at the largest power of two. One leaf per line of the CSV file.

Safe reads. The function `open_verified()` verifies the chain, reads the file once, hashes those bytes and returns them. Training and the gateway load data and models from those bytes only. Model weights are stored as safetensors, and the gateway refuses pickle-based formats.

6.3 Machine-learning pipeline

The pipeline uses the digits dataset shipped with scikit-learn: 1,797 images of 8 by 8 pixels, split into 1,000 training, 437 fine-tuning and 360 held-out samples. The model is a small convolutional network with 38,282 parameters. The base model is trained for 15 epochs and fine-tuned for 8, with the seed 42. On the reference run the deployed model reaches a clean test accuracy of 95.0 %. The model is small on purpose: the cryptographic mechanisms do not depend on its size, and a full run that trains real models takes seconds, so every experiment is reproducible on a laptop.

Before a dataset is signed, three content checks run on it (Table 11). Their results are written into the signed manifest. A flag never blocks the signature: it travels with the artifact and becomes check V9 at verification.

Table 11. The three content checks that run before a dataset is signed.

Check	Detects	Known blind spot
Label distribution	Heavy label flipping or poisoning aimed at one class (chi-square test against a reference)	Low poisoning rates
Duplicate rows	Amplification by duplication (more than 2 % of duplicates)	Near-duplicates
Corner trigger	A saturated 2 by 2 patch in a corner, the BadNets trigger	Any other trigger shape (variant E4)

6.4 Interfaces

Command line. `python -m cli.main` creates the role keys, registers and signs an artifact, verifies it (exit code 0 for **VERIFIED**, 2 for **WARNING**, 1 for **FAILED**), revokes an artifact or a key, prints the trust log, exports an attestation and imports an external model.

REST API. A FastAPI application with 26 routes: read the registry and the provenance graph, register, verify, deploy through the gateway, revoke, export an attestation, read the trust log,

register a publisher and import an external model, read events and metrics. Interactive documentation is served at `/docs`.

Dashboard. A Streamlit application with nine pages: overview, provenance chain (Figure 8), verification of an artifact, attack lab with live tampering, security events, metrics, external models, trust log with bypass attempts in a sandbox, and a comparison page.

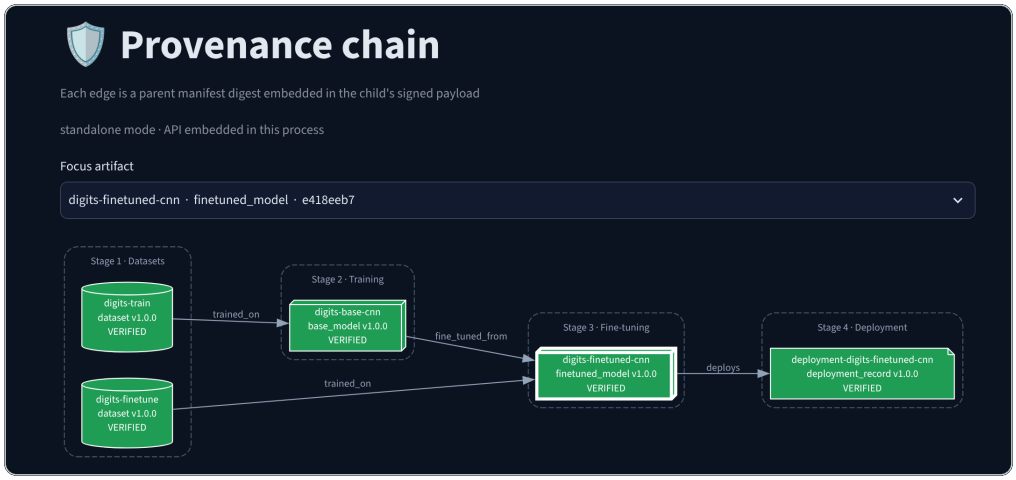


Figure 8. Dashboard, provenance chain page (run of 10 November 2026). Each edge is a parent manifest digest embedded in the signed payload of the child. The colours show the last verification result stored in the index; the verifier itself never trusts that stored status.

6.5 Docker and continuous integration

A Docker Compose file starts three services on one shared volume: the pipeline, which runs once and exits; the API on port 8000; and the dashboard on port 8501.

Two GitHub Actions workflows run on every push to the main branch:

- `ci`: the linter, a guard that fails if a private key is committed, and the 168 tests on Python 3.11.
- `attest-and-verify`: it creates keys for the run, executes the four-stage attested pipeline and fails if the gateway blocks, verifies the deployed model again independently, runs a negative control (one flipped byte of the deployed model must give **FAILED**), runs the attack scenarios A to D, F and G (43 variants) and uploads the evidence. An optional job signs with Sigstore; it is disabled by default.

This workflow is the automated attestation in CI/CD that the brief suggests: a release is refused unless the gateway returns **VERIFIED**.

6.6 Test environments

Table 12. Where the system was executed.

Environment	What ran there	Date (2026)
Reference environment: cloud Linux container, Intel Xeon at 2.10 GHz with two logical cores, 8.2 GB of memory, no GPU, Python 3.13.16, PyTorch 2.15.0 on CPU	The full validation run: linter, 168 tests, pipeline, the 48 variants, every benchmark and figure. Supplementary experiments	9 November; supplementary runs on 10 November
GitHub-hosted runner (Ubuntu, Python 3.11.18, PyTorch 2.15.0)	Both workflows on every push: 168 tests, attested pipeline, negative control, 43 attack variants	9 and 10 November

Environment	What ran there	Date (2026)
Team laptop (MacBook Air), Docker	Image build and Compose stack: the four-stage pipeline (**VERIFIED**, deployed, clean accuracy 95.0 %), the API health check and the dashboard	16 October
Any reader's browser, on the website	Checks V1 and V3 on the deployed model of experiment E2, and the model itself	since 10 November

The benchmarks of Section 7 were measured in the reference environment only. Shared CI runners are not a stable timing environment, and the full test and benchmark run has not been repeated on team hardware.

6.7 Technologies of the brief

Python and PyTorch are used throughout. For signing, the brief offers Sigstore, cosign or custom signing with OpenSSL: we use custom Ed25519 signing on the OpenSSL backend of the `cryptography` library, with the optional Sigstore job in CI. The Merkle tree is our own implementation of RFC 6962. Docker and GitHub Actions are used as described above. Hyperledger Fabric and blockchain anchoring, both optional, are not used; the signed trust log takes their place (Section 5.5).

7 Experimentation and evaluation

This section covers the experiments of phase 4 and the tests and measurements of phase 5. The scenario scripts are in `attacks/`, the raw results in `results/` and the dated record of every run in the experimentation journal (`docs/09_experimentation_journal.md`).

7.1 Method

Protocol. Each attack variant starts from a fresh copy of a clean, fully attested chain. It applies one manipulation with the access the attacker would have, verifies the targeted artifact with its whole chain, and records the status, the checks that fired and the wall time of that verification. A variant is correct when the observed status equals the status predicted before the run. Each experiment has an untouched control, which must stay **VERIFIED**.

Backdoor experiments train real models. The poison is the BadNets trigger [10]: a saturated 2 by 2 patch in a corner, stamped on 8 % of the training rows (80 rows), whose label is changed to 7. The *backdoor success rate* is the share of held-out images of the other classes (324 of the 360) that are classified as 7 once they carry the trigger. The *clean accuracy* is the accuracy on the 360 untouched held-out images.

Environment. Unless stated otherwise, the numbers come from one run of the whole suite on 9 November 2026 in the reference environment (Table 12), with the seed 42. Two supplementary runs were made on 10 November in the same environment; they are identified as such.

7.2 Every manipulation after signing was detected

Figure 9 shows the 48 variants and Table 13 summarises the seven experiments. All 48 variants ended with the predicted status. Appendix B lists each one.



Figure 9. The 48 variants, one tile each, and the verification time of each experiment. The tiles of experiments 5 and 7 omit the prefixes E5 and E7 of their identifiers. A **VERIFIED** tile is either an untouched control or a case where **VERIFIED** was the predicted status, as for E2, E4 and E7-F (Section 7.4).

Table 13. The seven experiments: 48 variants, all with the predicted status.

	Experiment (scenario)	Variants	Outcome	Checks that fired
1	Dataset tampering (A)	5	control VERIFIED; 4 FAILED	V1, V2, V6
2	Model substitution (B)	4	control VERIFIED; 3 FAILED	V1
3	Invalid signature, key misuse (D)	7	control VERIFIED; 6 FAILED	V3, V4, V6
4	Broken provenance (C)	5	control VERIFIED; 4 FAILED	V3, V5, V8
5	Revocation and its bypasses (F)	13	2 VERIFIED as predicted; 11 FAILED, the gateway blocks	V3, V4, V6, V7
6	Poison before signing (E)	5	3 VERIFIED (E0, E2, E4); 1 FAILED (E1); 1 WARNING (E3)	V6, V9
7	Untrusted external source (G)	9	3 VERIFIED; 4 FAILED; 2 REFUSED at import	V1, V3, V4, V6

Every manipulation made after signing was rejected, including the subtle ones:

- **On datasets:** flipping the labels of five rows, appending one row, or adding a single trailing space to one line of a signed file. The report names the changed lines (V1, V2).
- **On models:** replacing the file, nudging one weight by 10^{-6}, which leaves the test accuracy unchanged, or saving identical weights again in the pickle format (V1). Integrity is stricter than behaviour: a functionally identical model is still rejected.
- **On provenance:** deleting a parent record, editing a parent link without signing again, or presenting a validly signed model with no dataset ancestry (V5, V3, V8). In variant C2 an insider modifies the dataset and signs it again *with the real data-steward key*: the dataset alone verifies, and the model that descends from it fails at V5. A legitimate signature cannot rewrite history beneath existing descendants.
- **On keys:** corrupting or replaying a signature, signing with an unknown or a revoked key, or using the data-steward key to sign a model (V3, V4).

Revoking an artifact or a key fails every descendant and blocks the deployment, while a dataset signed by another role still verifies after the training key is revoked: the damage is limited to

the revoked key. Detection, measured as the verification time of the attacked artifact with its chain, took 1.7 to 15.1 ms, with a median of 7.5 ms.

7.3 Attacks on the trust itself

Experiment 5 targets the mechanism that decides who is trusted. After a base model is revoked, eight bypasses (C1 to C8) try to make its fine-tuned descendant verify again:

- **Editing the index** (deleting the row of the revoked artifact, C1; deleting the revoked key, C2; overwriting the cached status with **VERIFIED**, C3). These fail because the verifier never reads a cached status and treats a missing row as a revocation it cannot rule out.
- **Clearing the `revoked` flag in SQLite** (C4). This was the documented limitation of version 0.3. It now fails: the signed log still says revoked, and the disagreement raises an `INDEX_TAMPERING` event (Figure 10).
- **Inserting a rogue key** into the index with the training role and using it to sign a substituted model (C5). It fails because the key never entered the signed log.
- **Attacking the log**: truncating it to drop the revocation (C6), editing the record in place (C7) and deleting the file (C8). All three fail closed: the replay no longer ends at the pinned head, or no longer verifies under the root key.

What remains trusted is what the threat model declared trusted: whoever holds the root key and can rewrite the pin controls the trust decisions.

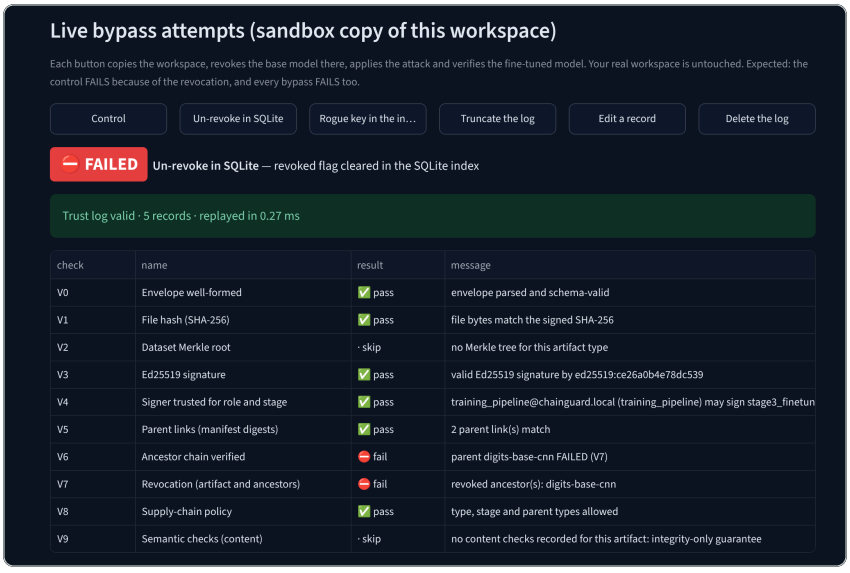


Figure 10. Dashboard, trust log page: the bypass C4 replayed in a sandbox copy of the workspace (run of 10 November 2026). The revoked flag was cleared in the SQLite index; the fine-tuned model still fails at V6 and V7 because the signed log is the authority.

7.4 Signed is not safe

Experiments 6 and 7 measure what the signature does not cover (Figure 11).

The clean pipeline (E0) deploys a model with a clean accuracy of 95.0 % and a backdoor success rate of 0 %. Poison injected into the signed dataset *after* signing (E1) fails the chain at V6, as expected. When the data steward signs a dataset that was poisoned *before* attestation (E2), every check passes. The fine-tuned model deploys as **VERIFIED** with a clean accuracy of 94.2 % (339 of the 360 held-out images), within one point of the clean model, and a backdoor success rate of 77.2 % (250 of 324). Ordinary testing would not reveal it.

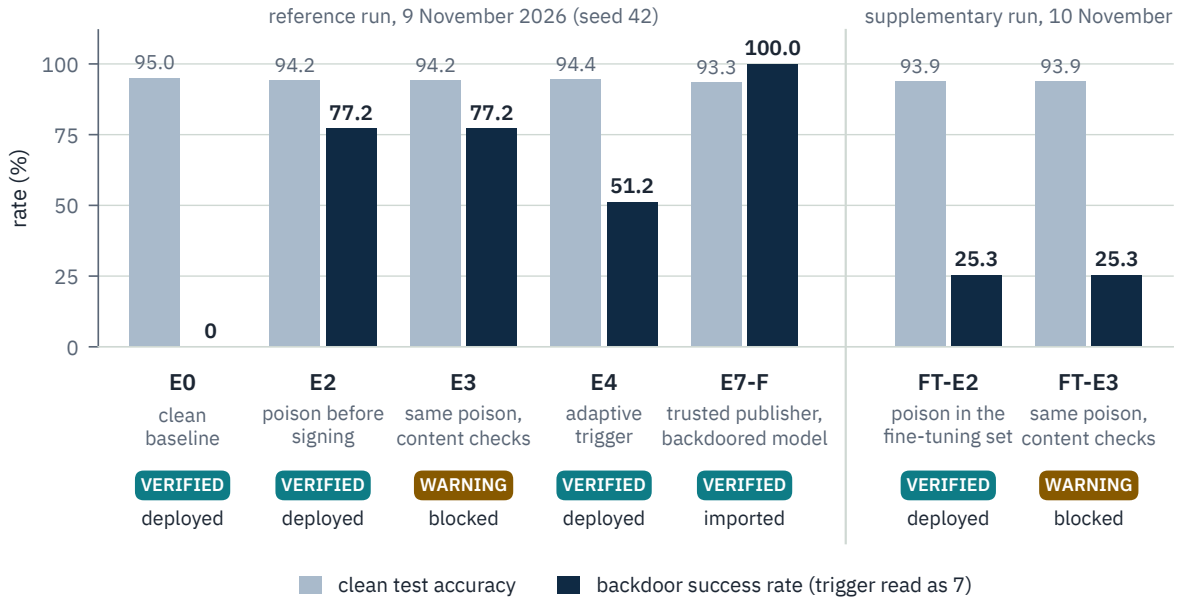


Figure 11. Clean accuracy and backdoor success rate of the deployed or imported model, with its verification status. Signed poison (E2), an adaptive trigger (E4) and the backdoored model of a trusted publisher (E7-F) pass every cryptographic check. Content checks catch the trigger they know (E3, FT-E3).

With the content checks enabled before signing (E3), the known corner trigger and the shifted label distribution are flagged. The dataset carries a signed flag, the model verifies with **WARNING** and the gateway blocks it by default: heuristics work as evidence. An adaptive attacker who moves the trigger to a pattern the checks do not know (E4, a bar on the left edge, 5 % of the rows) deploys as **VERIFIED** with a backdoor success rate of 51.2 %.

Experiment 7 repeats the lesson at the boundary of the organisation. A backdoored model trained by a *trusted*, registered publisher (E7-F) imports as **VERIFIED** with a clean accuracy of 93.3 % and a backdoor success rate of 100 %. The other variants are all rejected: the unknown publisher and the model modified in transit are **FAILED**, the unsigned model and the manifest without its signature are **REFUSED** at import, and the model of a publisher whose key was revoked is **FAILED**.

Fine-tuning on clean data weakens the backdoor without removing it: in E2 the base model has a backdoor success rate of 99.1 %, which drops to 77.2 % after fine-tuning (99.4 % to 51.2 % in E4). Fine-tuning is not a defence to rely on.

> **The result in one sentence.** A valid signature proves *who* attested an artifact and *that nothing changed since*. It transfers trust to the signer and creates none.

7.5 The four example tests of the brief

The brief gives four example tests. Table 14 states where each one was run and what was observed. The first one names the fine-tuning dataset, while experiment 6 poisons the training set of the base model, so we ran it literally as a supplementary experiment on 10 November (FT-E2 and FT-E3, on the right of Figure 11): 8 % of the 437 fine-tuning rows (35 rows) receive the corner trigger and the label 7 before the data steward signs them, and the base model is trained on clean data.

Table 14. The four example tests of the brief and what was observed.

Example test of the brief	Where	Observed
Inject a backdoor into a fine-tuning dataset and verify detection failure or success	FT-E2, FT-E3 (and E2 to E4 for the training set)	Detection failure by cryptography: VERIFIED, deployed, clean accuracy 93.9 %, backdoor success 25.3 % (base model: 0 %). Detection success by the content checks: WARNING, blocked
Substitute a signed model with a tampered version	Experiment 2 (B1 to B3)	Control VERIFIED. A different model, one weight changed by 10^{-6} and the same weights saved as pickle are all FAILED at V1; the gateway blocks
Verify the provenance of a model from an untrusted repository	Experiment 7 (nine variants)	Trusted publisher VERIFIED. Unknown publisher, model modified after signing and revoked publisher key FAILED. Unsigned model and stripped signature REFUSED at import
Measure the attestation overhead on training and inference	Pipeline and inference benchmarks	Attestation 10.7 ms against 1,288 ms of training (0.83 %). One prediction: 0.084 ms before and 0.085 ms after the verified deployment

The backdoor of FT-E2 is weaker than that of E2 (25.3 % against 77.2 %) because eight epochs of fine-tuning on 35 poisoned rows teach the model less than fifteen epochs of training on 80. The boundary is the same wherever the poison enters before signing.

7.6 Robustness to the random seed

The reference run uses one seed. The seed decides the split of the data, the rows that are poisoned, the initial weights and the order of the mini-batches, so a single run could be a lucky or an unlucky draw. As a second supplementary experiment on 10 November, we repeated the four pipeline variants of experiment 6 for ten other seeds, 0 to 9 (Figure 12).

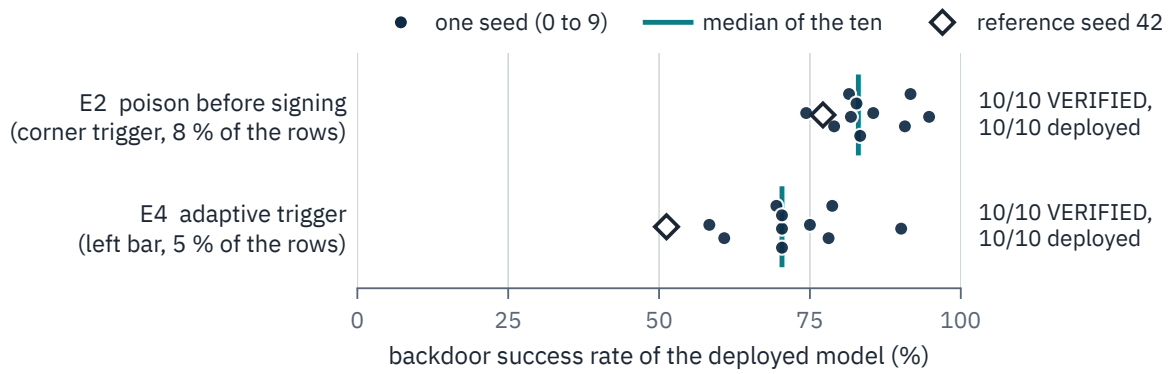


Figure 12. Backdoor success rate of the deployed model for ten other seeds. The status never changes. The reference seed gives the second weakest backdoor of the eleven runs for E2 and the weakest for E4.

The verification status did not change once in the 40 runs:

- **E0, clean:** ten times **VERIFIED** and deployed; no content check raised a false alarm; clean accuracy 95.8 to 98.6 % (median 97.2 %); backdoor success rate 0 %.
- **E2, poison before signing:** ten times **VERIFIED** and deployed; backdoor success rate 74.4 to 94.8 % (median 83.0 %); clean accuracy 95.8 to 98.3 %.
- **E3, same poison with content checks:** ten times **WARNING** and blocked.
- **E4, adaptive trigger:** ten times **VERIFIED** and deployed; backdoor success rate 58.3 to 90.1 % (median 70.4 %).

The run with the seed 42 reproduced the reference results exactly. The figures quoted in this report (77.2 % and 51.2 %) are therefore at the low end of what the attack achieves, not a favourable draw.

7.7 Cost

Figure 13 gives the cost of each operation and Figure 14 shows how it grows with the size of what is attested.

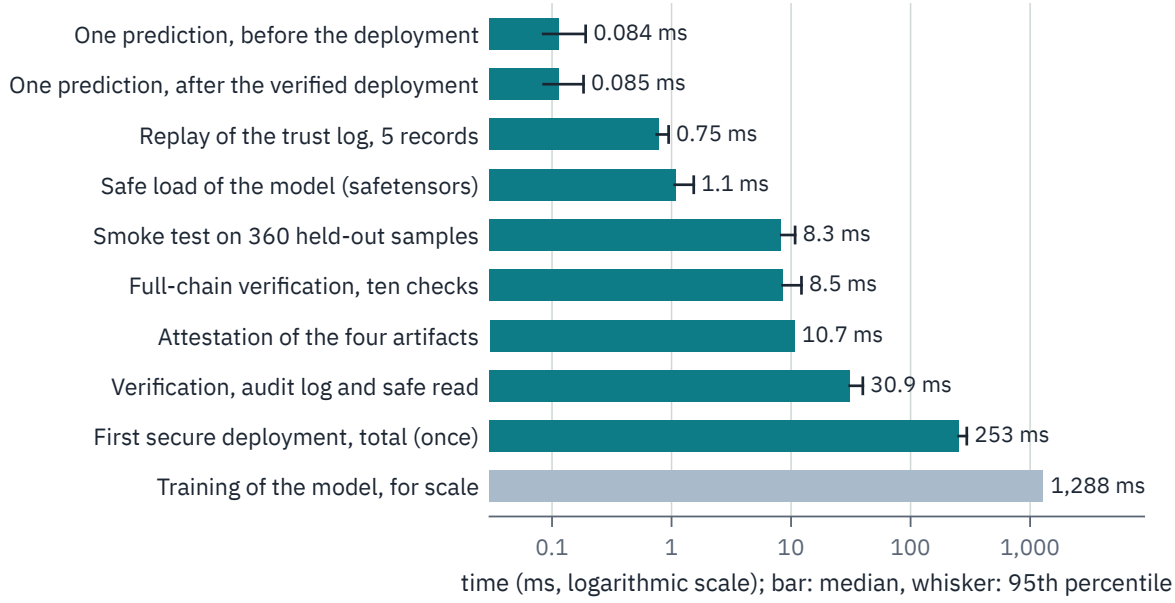


Figure 13. What attestation costs in the reference environment. Verification happens at deployment, so a prediction costs the same before and after.

Training. Attesting the four artifacts of the pipeline took 10.7 ms against 1,288 ms of training, or 0.83 %. The share depends on the workload and should shrink for larger models, because attestation grows with the bytes of the artifacts while training grows with the computation.

Verification. A full-chain verification of the fine-tuned model with its three ancestors takes 8.5 ms (median of 200 runs; 95th percentile 12.2 ms). Replaying the five-record trust log accounts for 0.16 ms of it; a first replay, in which every signature is verified, takes 0.75 ms.

Deployment. The first secure deployment costs 253 ms, once. The chain verification with its audit writes and the safe read take 31 ms, the safetensors load 1.1 ms and the smoke test 8.3 ms. The remaining 210 ms or so go to signing and registering the deployment record, which verifies the chain a second time, and to event and metric writes.

Inference. A prediction costs 0.084 ms before and 0.085 ms after the verified deployment (medians of 1,000 runs each). Verification is not on the inference path.

Storage. A signed manifest is 1.4 to 3.1 KB per artifact. The side file that holds the row hashes of a dataset grows with the number of rows (30 and 68 KB here).

Scaling. SHA-256 runs at about 1.2 GB/s, so a model of 256 MiB hashes in 220 ms. The Merkle root of a dataset of one million rows takes 1.7 s. Replaying the trust log costs about 0.14 ms per record when every signature is verified (0.75, 6.9, 69 and 288 ms for 5, 50, 500 and 2,000 records) and about 15 microseconds per record with cached signatures. With thousands of records, one would checkpoint a signed snapshot of the state instead of replaying from the first record.

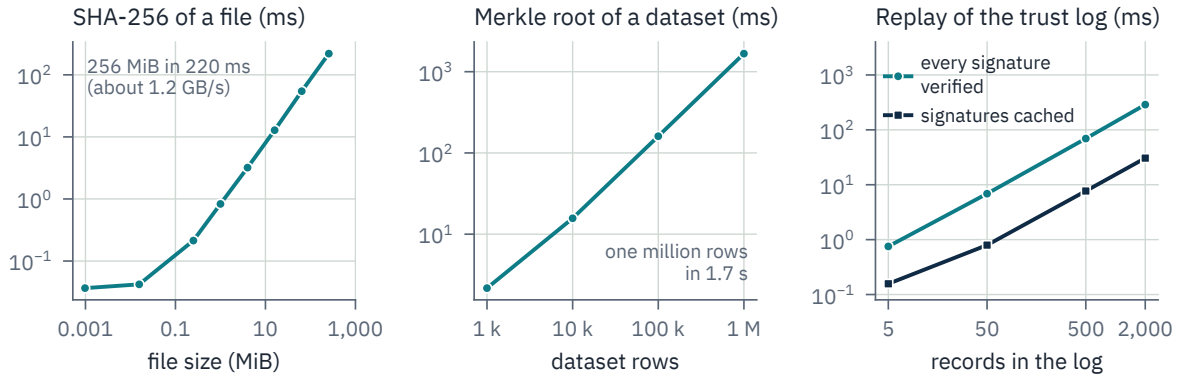


Figure 14. How the cost grows. Hashing is linear in the size of the file and dominates for large artifacts; signing and verifying a signature do not depend on the size (0.04 ms and 0.12 to 0.14 ms). Both axes are logarithmic.

7.8 Tests and reproducibility

Automated tests. 168 tests pass in 36 s in the reference environment, with none failed and none skipped: unit tests of hashing, canonical serialisation, signing, manifests, Merkle trees, the policy, the content checks and the trust log; integration tests of registration and verification, the pipeline, the revocation experiment, external models, the interoperability export, the API and the scenarios. The suite grew with the system: 113 tests on 5 October, 143 on 16 October, 168 on 9 November.

Continuous integration. Both workflows passed on the first execution on GitHub (9 November) and on every push since, on Python 3.11, a different version from the reference environment.

Regeneration. `python -m attacks.run_all` regenerates every table of this section, `python -m scripts.benchmark` and its companions every cost figure, and `report/figures/make_figures.py` every chart of this report from the result files (Appendix D).

Verification by the reader. The website serves the signed model of experiment E2, its signed manifest and the public key. The page checks the file hash (V1) and the Ed25519 signature (V3) in the reader's browser and runs the network in JavaScript: 339 of 360 clean images correct and 250 of 324 triggered images read as 7, the figures of E2. Against PyTorch on all 684 inputs, the JavaScript network gave no different prediction.

7.9 Threats to validity

- **The attack suite is written by the defenders.** It shows that the checks behave as designed against the manipulations we thought of, not that no other manipulation exists.
- **The model is small.** An 8 by 8 digit classifier shows how large the gap is on a small model, not that it has that size for a language model, where poisoning is cheaper [4].
- **The content checks are simple heuristics**, tuned by a defender who knows the trigger. E4 shows the cost of that assumption. Stronger detectors would raise the bar and, as adaptive attacks show, not close the gap.
- **Timings come from one environment.** They were not measured again on team hardware, and shared CI runners are not a stable timing environment.
- **One trigger family and one dataset.** The seed experiment varies the split, the poisoned rows and the initialisation, not the trigger family or the task.

None of this touches the central result, which holds by construction: a signature covers bytes, not behaviour.

8 Critical analysis

This section is the critical analysis that phase 5 asks for: what the system guarantees, what it does not, and how it could be improved.

8.1 What the system guarantees

Table 15 states each guarantee with the mechanism that provides it and the experiment that tests it.

Table 15. Security guarantees, the mechanism behind each one and the evidence.

Guarantee	Mechanism	Evidence
The bytes of an artifact are those that were signed, down to the row for a dataset	SHA-256 of the file and Merkle root of the rows (V1, V2)	Experiments 1 and 2
A known signer, in an authorised role, attested it	Ed25519, trust log, policy (V3, V4, V8)	Experiment 3
The lineage from the dataset to the deployment is intact	Parent digests in the signed payload, recursive verification (V5, V6)	Experiment 4
A withdrawn key or artifact stays withdrawn, even against an insider who can write the database	Signed, hash-chained log with a pinned head (V7)	Experiment 5
What is loaded is what was verified	One read, hashed again, loaded from the same bytes; safetensors only	Variant B3 and the gateway tests
An outside model comes from a registered publisher	Publisher key, separate import stage	Experiment 7

These guarantees hold under the assumption of Section 4: the keys, the pin, the policy and the verifier are trusted. They hold against every manipulation of our suite, at a cost that an enterprise would not notice: under one per cent of training time, a few milliseconds per verification and nothing per prediction.

8.2 What it does not guarantee

- **That signed content is benign.** Poison that is present when the data steward signs is faithfully preserved and verified (E2, E4, FT-E2). A backdoored model from a trusted publisher verifies (E7-F). The signature moves the question from “are these the right bytes?” to “do I trust this signer?”.
- **That a signer is honest or uncompromised.** A stolen key that has not been revoked signs valid manifests. A compromised training host can sign a false description of how the weights were produced.
- **That a verifier sees the latest revocations.** The trust log is local and has a single writer. A verifier that holds an old copy would miss a recent revocation, and the pin file is as critical as the root key.
- **That content checks find a backdoor.** They flag the trigger family they model and miss the others. They are evidence, never proof.

8.3 How the two kinds of evidence fit together

The design keeps cryptographic evidence and content evidence in separate checks, V0 to V8 on one side and V9 on the other. A heuristic can never produce **FAILED**, and a signature can never

produce “safe”. The status **WARNING** exists for exactly this reason: it says that the integrity is intact and that something about the content deserves a human decision. Reports also name the external artifacts whose provenance ends at a publisher. We consider this separation the main lesson of the project for an enterprise. A dashboard that shows a green **VERIFIED** without saying what the status covers invites the wrong conclusion.

8.4 From this implementation to production

Several choices fit an eight-week project and would change in production. Table 16 sets each one against what a production deployment would use.

Table 16. Choices that fit an eight-week implementation and what a production deployment would use.

In this implementation	In production
Private keys as owner-only files or CI secrets	An HSM or a cloud KMS, or keyless signing with short-lived certificates (Sigstore)
Trust log local, with a single writer	Records published to a shared transparency log, or signed snapshots, so that every verifier sees one history
Publisher trust is a manual registration of a public key	Identity-based policies (Sigstore, OpenSSF Model Signing)
Canonical JSON written in Python (sorted keys, compact)	RFC 8785 [29], for signers in other languages
API without authentication or TLS	Mutual TLS or OpenID Connect, authorisation per role, keys never on the API host
The build environment is a signed claim	Hardware attestation or reproducible training
Heuristic content checks (corner trigger, label drift)	Trigger-agnostic detectors and evaluation suites
The gateway verifies the chain twice on a first deployment	One verification, reused for the deployment record

8.5 Improvement paths

1. **Shared history.** Publish the trust-log records and the release root of each deployment to a transparency log. This is where a blockchain anchor, optional in the brief, would also fit.
2. **Keys.** Keyless signing for the roles, hardware-backed root keys, then post-quantum signatures through the algorithm field of the manifest.
3. **Content evidence.** Detectors that do not assume a trigger shape (activation clustering, spectral signatures, Neural Cleanse) as further V9 checks, measured against adaptive triggers.
4. **Scale.** The evaluation repeated on team hardware and with larger models, where hashing dominates and the relative overhead should fall.
5. **Inventory.** The provenance graph exported as a signed bill of materials (CycloneDX or SPDX).

9 Demonstration, use case and valorisation

Phase 6 asks for a demonstration or use case and for a conference proposal. This section presents both, with the website that makes the result checkable by anyone.

9.1 Demonstration

The system can be demonstrated in three ways, from the most complete to the most accessible (Table 17).

Table 17. Three ways to see the system work.

Demonstration	How to start it	What it shows
Command-line demonstration, about ten minutes	`python -m scripts.demo`	Fifteen steps in a throw-away workspace: a clean chain, the ten checks, **VERIFIED**, the provenance tree, a tampered dataset, **FAILED**, the gateway blocking, an invalid signature, a revocation, attacks on the index and on the trust log, the poisoned-before-signing limit, a trusted and an untrusted external model, the in-toto export, live metrics
Dashboard	`make standalone`, then a browser on port 8501	The same operations by clicking: build the chain, verify, tamper with a real file and restore it, try the bypasses of the trust log in a sandbox, import a model from a trusted, an unknown or an unsigned publisher
Website	`ai-chainguard.vercel.app`	The deployed model of experiment E2 verified and run in the browser, a demonstration of the chain with ten attacks, the results, and a 40-second recorded tour of the dashboard

The first two run the complete system on the presenter's machine. The website is a static page: it runs no server of ours. The two checks it performs and the model itself execute in the visitor's browser, on the real files of the run (Figure 15). The page is available in English, French, Spanish, German and Hindi.



Figure 15. The website (10 November 2026). Left: the deployed model of experiment E2 passes the hash and signature checks in the browser. Right: the same model reads a 3 as a 7 once the trigger is stamped on it.

9.2 Use case: a model taken from a hub and fine-tuned in the enterprise

A team wants to fine-tune a pretrained model from an outside publisher on internal data and deploy it.

1. The security team registers the public key of the publisher. The model is imported: an unsigned file is refused, an altered one fails verification and is quarantined, and the accepted model is labelled as external.
2. The data steward registers the internal dataset. Content checks run, and their result is signed with the data.
3. The training pipeline fine-tunes and signs the result, whose manifest embeds the digests of the dataset and of the external model.
4. At deployment the gateway verifies the whole chain, loads the verified bytes and signs a deployment record with a release root.
5. Months later the publisher announces that its key was compromised. The release manager revokes the key in the trust log. From that moment every model that descends from that publisher fails verification, and the gateway refuses to deploy them again, even if an insider edits the database.
6. During the investigation, the release root of each deployment tells the auditors exactly which dataset, which base model and which keys were behind the model in production, and the attestations can be handed to tools that read in-toto or SLSA.

Steps 1 to 5 are exercised by experiments 5 and 7. Step 6 uses the deployment record and the export of Section 5. The use case also shows the limit: if the publisher's model was already backdoored when it was signed, steps 1 to 4 succeed (variant E7-F).

9.3 Conference proposal

The common framework of the projects asks for a conference proposal with a title, an abstract and a targeted venue (`docs/13_conference_proposal.md`). We went one step further and wrote the full paper, six pages in IEEE conference format (`paper/`).

Title. *Signed Is Not Safe: Measuring the Boundary Between Cryptographic Integrity and Semantic Trust in AI Supply Chains.*

Abstract, in short. The paper presents the attestation chain, then measures the boundary of its guarantees with the 48 variants: every manipulation after signing is detected, while a dataset poisoned before signing verifies and deploys. It argues that AI supply-chain controls must report cryptographic evidence and content evidence separately.

Venues. Table 18 lists the venues we target.

Table 18. Targeted venues. Deadlines and page limits must be checked on each call for papers before submitting.

Venue	Why it fits	Format
ACM SCORED, workshop on software supply-chain security, co-located with ACM CCS (first choice)	The contribution is a supply-chain mechanism and a measurement of its boundary: attestation, in-toto, SLSA, Sigstore	Workshop paper
ACM AISec, workshop on artificial intelligence and security, co-located with ACM CCS	Security of machine-learning systems, poisoning and backdoors	Workshop paper
IEEE SaTML, conference on secure and trustworthy machine learning	Trustworthy machine learning, measurement of defences	Conference paper
C&ESAR (Rennes) or RESSI	The French security research and teaching community; a French version of the paper exists	Conference or short paper

9.4 Value for an enterprise security team

Three properties make the work usable beyond the course. It needs no outside service: it runs offline, with files and one SQLite database. It speaks the formats that software supply-chain tools already consume (in-toto statements, DSSE envelopes, SLSA provenance). And it states its limit in the product itself: a status that means “exactly what trusted signers attested”, a separate status for content doubts, and a label on models whose provenance ends outside the organisation.

10 Project management

10.1 Plan and milestones

The scoping note planned four milestones over the eight weeks, with an end-to-end chain first so that integration would never be left to the end. Table 19 compares the plan with the dated evidence.

Table 19. Milestones planned in the scoping note and when each one was reached.

Milestone	Planned	Reached
Minimum chain: register, hash, sign, verify, detect tampering	Week 2	5 October: version 0.2 already attested the four stages, with 113 tests and 26 attack and control variants in five scenarios. First progress report
Four-stage chain with roles, policy and revocation	Week 4	16 October: version 0.3 added the revocation experiment and external models, with 143 tests and 44 variants. The Compose stack ran on a team laptop. Second progress report
Attack scenarios and measurements	Weeks 5 to 7	26 variants on 5 October, 44 on 16 October, 48 in the final run of 9 November, each time with the cost measurements
Report, paper and demonstration	Week 8	9 and 10 November: version 0.4 with the signed trust log and the export, full validation run, public repository with both CI workflows passing, the paper, the website and this report

Each full run is kept: the results of 5 October and 16 October are archived unchanged in the repository (`results/`), and the experimentation journal has one dated entry per run, which is never deleted.

10.2 Team and ownership

Table 20 gives the area each member is responsible for. It describes ownership, not a record of individual contributions.

Table 20. Team roles and areas of ownership.

Member	Role	Owns
Karan DESHMUKH	Technical lead, security architecture	Architecture and integration, verifier and trust-log design, external-model design, deployment gateway, API, interoperability export, coordination, paper
JOGI Sudheer reddy	AI and ML engineer	Dataset and training pipeline, backdoor experiments (experiments 6 and 7), inference benchmark
DAMA Dola Krishna	Testing, documentation and quality	Test suite, experimentation journal, collection of results, validation script, dashboard support, reports
MEKALA Rambabu	Security and DevOps engineer	Hashing, signatures, key and publisher-key handling, revocation and trust-log operations, Docker, CI/CD

10.3 Reporting to the referent

The brief asks for a detailed weekly progress report. Karan DESHMUKH sent it to the referent by email. Two formal progress reports mark the main steps: the report on the initial period (5 October 2026) and Progress Report 2 (16 October 2026). The progress update for version 0.4 (9 November 2026) is in the repository (`docs/16_progress_update_v0.4.md`). The two formal reports are archived with this report so that the progression can be checked.

10.4 Method

Evidence rule. From the first report, a feature is reported as completed only when code and repeatable execution evidence exist, and every number is tied to the run that produced it. This report follows the same rule.

Use of AI assistance. As stated in each progress report, the code was produced with AI-assisted code generation and validated by execution: tests, complete validation runs in the reference environment, the CI workflows and the Docker run on a team laptop. The design decisions, the experiments, their analysis and the conclusions are the responsibility of the team.

10.5 Difficulties and lessons

- **Our own experiment broke our design.** In version 0.3 the revocation experiment showed that clearing one flag in the database undid a revocation. We reported it as a limitation on 16 October and removed it in version 0.4 with the signed trust log. Lesson: attack your own trust anchors, and write the limitation down before fixing it.
- **A detector must be measured on clean data first.** The first version of the corner-trigger check flagged 11 clean digits. Requiring the minimum of the four pixels instead of their mean brought the false alarms to zero, on the reference split and on the ten other seeds of Section 7.6.
- **Run the automation before trusting it.** The CI guard against committed private keys matched its own search pattern in the workflow file and would have failed on every run. Executing the workflow steps locally found the bug before the first run on GitHub.
- **One environment is not enough.** Docker could not run in the reference environment, so the Compose stack was executed on a team laptop. Timings still come from one environment.
- **Saying what a result does not mean is part of the result.** Keeping **VERIFIED** from being read as “safe” shaped the statuses, the dashboard and the wording of every report.

11 Conclusion and future work

We built an attestation chain for AI pipelines. It links datasets, base models, external models, fine-tuned models and deployments through signed manifests that embed the digests of their parents. It enforces who may sign which stage, keeps trust and revocation in a signed, pinned, append-only log, and gates deployment on a verification of the whole chain. The system is complete from the command line to the dashboard, tested by 168 automated tests and by two CI workflows, and documented phase by phase as the brief requires.

On 48 attack and control variants, every manipulation made after signing was detected, including the bypasses aimed at the index and at the log itself. The cost is under one per cent of training time, 8.5 ms per verification and nothing per prediction. The answer to the first sub-question of the project is therefore yes: an enterprise can verify cryptographically that an artifact, and everything it descends from, is exactly what authorised parties attested.

The same suite answers the second sub-question. Poison that is present before signing verifies and deploys, with a backdoor success rate of 77.2 % in the reference run and of 74 to 95 % for ten other seeds. Content checks catch the trigger they know and miss an adaptive one. A backdoored model from a trusted publisher verifies. Cryptographic attestation is necessary for an AI supply chain and it is not sufficient: it must be combined with evidence about content, and the two must be reported separately.

Future work. In order of priority: publish the trust log and the release roots to a shared transparency log; replace long-lived keys by keyless signing and hardware-backed roots; repeat the evaluation on team hardware and with larger models; add trigger-agnostic detectors as further content checks and measure them against adaptive triggers; export the provenance graph as a signed bill of materials. The paper is ready to be adapted to the page limit of the first targeted venue.

References

[1] PyTorch team, “Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022,” Dec. 31, 2022. [Online]. Available: https://pytorch.org/blog/compromised-nightly-dependency/. Accessed: Nov. 9, 2026.

[2] D. Cohen, JFrog Security Research, “Data scientists targeted by malicious Hugging Face ML models with silent backdoor,” Feb. 2024. [Online]. Available: https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. Accessed: Nov. 9, 2026.

[3] N. Carlini et al., “Poisoning web-scale training datasets is practical,” in *Proc. IEEE Symp. Security and Privacy (SP)*, 2024, pp. 407-425.

[4] A. Souly et al., “Poisoning attacks on LLMs require a near-constant number of poison samples,” arXiv:2510.07192 [cs.LG], Oct. 2025.

[5] European Parliament and Council, “Regulation (EU) 2024/1689 of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act),” *Off. J. Eur. Union*, L, 2024/1689, Jul. 12, 2024, Art. 15(5).

[6] OpenSSF, “SLSA specification v1.1: Supply-chain Levels for Software Artifacts,” Apr. 2025. [Online]. Available: https://slsa.dev/spec/v1.1/. Accessed: Nov. 9, 2026.

[7] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, “in-toto: Providing farm-to-table guarantees for bits and bytes,” in *Proc. 28th USENIX Security Symp.*, 2019, pp. 1393-1410.

[8] Z. Newman, J. S. Meyers, and S. Torres-Arias, “Sigstore: Software signing for everybody,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2022, pp. 2353-2367.

[9] OpenSSF, “Model Signing v1.0,” Apr. 2025. [Online]. Available: https://github.com/sigstore/model-transparency. Accessed: Nov. 9, 2026.

[10] T. Gu, B. Dolan-Gavitt, and S. Garg, “BadNets: Identifying vulnerabilities in the machine learning model supply chain,” arXiv:1708.06733 [cs.CR], 2017.

[11] K. Zanki, ReversingLabs, “Malicious ML models discovered on Hugging Face platform,” Feb. 2025. [Online]. Available: https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face. Accessed: Nov. 9, 2026.

[12] NIST, “Artificial Intelligence Risk Management Framework (AI RMF 1.0),” NIST AI 100-1, Jan. 2023.

[13] MITRE, “ATLAS technique AML.T0010: AI supply chain compromise.” [Online]. Available: https://atlas.mitre.org/techniques/AML.T0010. Accessed: Nov. 9, 2026.

[14] OWASP Foundation, “OWASP Machine Learning Security Top Ten, ML06:2023 ML supply chain attacks.” [Online]. Available: https://owasp.org/www-project-machine-learning-security-top-10/docs/ML06_2023-AI_Supply_Chain_Attacks.html. Accessed: Nov. 10, 2026.

[15] S. Josefsson and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA),” RFC 8032, IETF, Jan. 2017.

[16] Secure Systems Lab, “DSSE: Dead Simple Signing Envelope, protocol v1.” [Online]. Available: https://github.com/secure-systems-lab/dsse. Accessed: Nov. 9, 2026.

[17] in-toto project, “in-toto Attestation Framework: Statement v1.” [Online]. Available: https://github.com/in-toto/attestation. Accessed: Nov. 9, 2026.

[18] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, “Survivable key compromise in software update systems,” in *Proc. ACM Conf. Computer and Communications Security (CCS)*, 2010, pp. 61-72.

[19] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” RFC 6962, IETF, Jun. 2013.

[20] M. Mitchell et al., “Model cards for model reporting,” in *Proc. Conf. Fairness, Accountability, and Transparency (FAT*)*, 2019, pp. 220-229.

[21] T. Gebru et al., “Datasheets for datasets,” *Commun. ACM*, vol. 64, no. 12, pp. 86-92, 2021.

[22] OWASP CycloneDX, “Introducing OWASP CycloneDX v1.5: Advanced bill of materials standard empowering transparency, security, and compliance,” Jun. 2023. [Online]. Available: https://cyclonedx.org/news/cyclonedx-v1.5-released. Accessed: Nov. 10, 2026.

[23] The Linux Foundation, “SPDX 3.0 revolutionizes software management in systems with enhanced functionality and streamlined use cases,” Apr. 2024. [Online]. Available: https://www.linuxfoundation.org/press/spdx-3-revolutionizes-software-management-in-systems-with-enhanced-functionality-and-streamlined-use-cases. Accessed: Nov. 10, 2026.

[24] B. Chen et al., “Detecting backdoor attacks on deep neural networks by activation clustering,” in *Proc. AAAI Workshop on Artificial Intelligence Safety (SafeAI)*, 2019.

[25] B. Tran, J. Li, and A. Madry, “Spectral signatures in backdoor attacks,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

[26] B. Wang et al., “Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks,” in *Proc. IEEE Symp. Security and Privacy (SP)*, 2019, pp. 707-723.

[27] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal, “STRIP: A defence against trojan attacks on deep neural networks,” in *Proc. Annu. Computer Security Applications Conf. (ACSAC)*, 2019, pp. 113-125.

[28] Hugging Face, “Safetensors: a simple, safe way to store and distribute tensors,” documentation; security audit by Trail of Bits, 2023. [Online]. Available: https://huggingface.co/docs/safetensors. Accessed: Nov. 9, 2026.

[29] A. Rundgren, B. Jordan, and S. Erdtman, “JSON Canonicalization Scheme (JCS),” RFC 8785, IETF, Jun. 2020.

Appendix A Requirements of the brief and where each one is met

The table follows the brief of Projet Applicatif 2 line by line. The same checklist, with direct links to the files, is `docs/18_requirements_checklist.md`.

Requirement of the brief	Section	Evidence in the repository	Status
Common framework			
Project over two months, eight weeks	10	Week 1 from 21 September 2026; final day 10 November 2026	Done
Detailed weekly progress report to the referent	10	Sent by email; progress reports of 5 and 16 October; `docs/14`, `docs/16`	Done
Demonstration of the realisation	9	`scripts/demo.py` (15 steps), dashboard, website; `docs/15`, `docs/17`	Ready
Conference proposal: title, abstract, targeted venue	9	`docs/13`; full paper in `paper/`	Done
Pedagogical objectives			
Understand the threats specific to AI supply chains	2, 3	`docs/02` section 2, `docs/04`	Done
Analyse the risks for model provenance, dataset lineage and pipeline integrity	4	`docs/04`: assets, actors, trust boundaries, STRIDE, residual risks	Done
Study cryptographic signing, hash chains and Merkle trees	3, 5	`docs/02` sections 3 and 4, `docs/07`	Done
Design a coherent attestation architecture	5	`docs/05`, `docs/06`, `docs/07`	Done
Implement a functional system	6	Source code, 168 tests, two CI workflows passing	Done
Evaluate the results objectively	7, 8	`results/`, `docs/08`, including the measured limit	Done
Phase 1: scoping and problem definition			
Scope for the attestation of AI artifacts in an enterprise; a clear cybersecurity problem	1, 2	`docs/01` sections 1, 2 and 6	Done
Assets: datasets, pretrained models, fine-tuned weights, training scripts, inference APIs	2	`docs/01` section 3, `docs/04` section 1	Done
Threats: data poisoning, backdoor injection, model substitution, pipeline compromise	2, 4	`docs/01` section 4, `docs/04`	Done
Deliverable: scoping note of 2 to 3 pages	2	`docs/01_scoping_note.md`	Done
Phase 2: state of the art and critical analysis			
Attacks: direct poisoning, backdoored base models, environment poisoning	3	`docs/02` sections 2.1 to 2.3	Done
Existing solutions: cryptographic signing, SLSA for ML, model cards, Sigstore	3	`docs/02` sections 3 to 5	Done
Compare open-source, industrial and academic approaches; identify the limitations	3	`docs/02` sections 8 and 9, `docs/03`	Done

Requirement of the brief	Section	Evidence in the repository	Status
Deliverables: synthesis of 3 to 5 pages, comparative table, argued justification of the approach	3	`docs/02` (justification in section 10), `docs/03`	Done
Phase 3: solution design			
Provenance and integrity architecture for AI artifacts	5	`docs/05`	Done
Signing, verification and revocation mechanisms	5	`docs/07`; `chainguard/trustlog.py`	Done
Justified technical choices	3, 5	`docs/05` section 7, `docs/02` section 10	Done
Deliverables: architecture diagrams, functional and technical description, attack scenarios	5	`docs/05` (six diagrams), `docs/06`, `docs/07`, `docs/08`	Done
Indicative architecture: hash chain or Merkle tree	5	Hash-linked manifests and row-level Merkle roots for datasets	Done
Indicative architecture: signing of models and datasets at each pipeline stage	5	Ed25519 manifests for datasets, base model, fine-tuned model and deployment record	Done
Indicative architecture: blockchain anchoring (optional)	3	Not built; a signed, hash-chained, pinned trust log instead, justified in `docs/02` sections 4 and 10	Optional, not built
Indicative architecture: CI/CD integration for automated attestation	6	`.github/workflows/attest-and-verify.yml`	Done
Indicative architecture: verification API for deployment gateways	6	`api/` and `deployment/gateway.py`	Done
Phase 4: implementation and experimentation			
Simulated ML pipeline	6	`ml_pipeline/`	Done
Attestation across at least three pipeline stages	5	Four stages	Done
Documented code and scripts	6	`README.md`, docstrings, `scripts/`, `docs/07`	Done
Description of the test environment	6	`docs/09`, `results/validation_logs/environment.txt`	Done
Experimentation journal	7	`docs/09_experimentation_journal.md`	Done
Technologies: Python, PyTorch; Sigstore, cosign or custom signing with OpenSSL; Merkle tree library; Docker, GitHub Actions; Hyperledger Fabric (optional)	6	Python and PyTorch; custom Ed25519 signing on OpenSSL and an optional Sigstore job; own RFC 6962 implementation; `docker/`; `.github/workflows/`; Hyperledger not used	Done
Phase 5: testing, evaluation and critical analysis			
Test against realistic attack scenarios	7	48 variants in seven experiments, `attacks/`, `docs/08`	Done
Measure detection rates and overhead	7	`results/attacks/`, `results/metrics/`	Done
Deliverables: test results as logs, tables and graphs	7	`results/validation_logs/`, `results/attacks/summary.csv`, `results/figures/`, `report/figures/`	Done
Critical analysis of the guarantees and the limitations; improvement paths	8	Paper sections V and VI, `docs/04` (residual risks)	Done
The four example tests	7	Table 14 of this report	Done
Phase 6: reporting and valorisation			
Final report of about 30 pages	all	This document; source in `report/`	Done
Oral defence	9	Presentation guide (`docs/17`) and slides prepared	Ready

Requirement of the brief	Section	Evidence in the repository	Status
Demonstration or use case	9	Demonstration script, dashboard, website	Done
Conference proposal	9	`docs/13`, `paper/`	Done
Expected results			
A functional attestation system; a complete engineering approach with threat modelling; a critical analysis of the results; work that can be valorised in an enterprise security context	all	The repository, documents `01` to `09`, this report, the paper and the website	Done

Appendix B The 48 variants

Reference run of 9 November 2026. For every variant the observed status equals the status predicted before the run. The time is that of the verification of the attacked artifact with its chain.

Variant	Manipulation	Status	Checks that fired	Time (ms)
Experiment 1: dataset tampering (scenario A)				
A0	Control: untouched signed dataset	VERIFIED	none	5.4
A1	Flip labels of 5 signed rows	FAILED	V1, V2	7.4
A2	Downstream impact on the final model	FAILED	V6	14.5
A3	Append one injected row	FAILED	V1, V2	7.5
A4	One trailing space on one line	FAILED	V1, V2	7.5
Experiment 2: model substitution (scenario B)				
B0	Control: legitimate fine-tuned model	VERIFIED	none	9.9
B1	Replace with a different model	FAILED	V1	15.1
B2	Nudge one weight by 10^{-6}	FAILED	V1	10.4
B3	Same weights re-saved as pickle	FAILED	V1	10.8
Experiment 3: invalid signature and key misuse (scenario D)				
D0	Control: valid signature	VERIFIED	none	9.4
D1	Corrupt one bit of the signature	FAILED	V3	9.9
D2	Replay a signature from another manifest	FAILED	V3	10.5
D3	Sign with an unregistered key	FAILED	V3, V4	9.4
D4	Signature by a revoked key	FAILED	V4, V6	10.1
D5	Trusted key used outside its role	FAILED	V4	10.3
D6	Malformed signature encoding	FAILED	V3	13.8
Experiment 4: broken provenance (scenario C)				
C0	Control: intact chain	VERIFIED	none	6.0
C1	Delete the parent dataset record	FAILED	V5, V8	2.7
C2	Re-sign a modified parent (history rewrite)	FAILED	V5	5.5
C3	Edit parent link without re-signing	FAILED	V3	7.5
C4	Model with no dataset ancestry	FAILED	V8	2.1

Variant	Manipulation	Status	Checks that fired	Time (ms)
Experiment 5: revocation and its bypasses (scenario F)				
E5-0	Control: nothing revoked, gateway deploys	VERIFIED	none	13.9
E5-A1	Revoke the base model: descendant fails	FAILED	V6, V7	10.6
E5-A2	Revoked ancestor: deployment blocked	FAILED	V6, V7	10.4
E5-B1	Revoke the training_pipeline key: its models fail	FAILED	V4, V6	10.8
E5-B2	Dataset signed by another role still verifies	VERIFIED	none	3.9
E5-C1	Bypass: delete the revoked artifact's index row	FAILED	V7	7.2
E5-C2	Bypass: delete the revoked key from the index	FAILED	V4, V6	7.0
E5-C3	Bypass: overwrite cached status with VERIFIED	FAILED	V6, V7	6.9
E5-C4	Bypass: un-revoke by editing the index (the limitation of version 0.3)	FAILED	V6, V7	6.7
E5-C5	Bypass: rogue key inserted into the index signs a model	FAILED	V3, V4	6.8
E5-C6	Bypass: truncate the trust log (rollback)	FAILED	V3, V4, V6, V7	5.3
E5-C7	Bypass: edit a trust-log record in place	FAILED	V3, V4, V6, V7	6.2
E5-C8	Bypass: delete the trust log	FAILED	V3, V4, V6, V7	5.1
Experiment 6: poison before signing (scenario E)				
E0	Clean baseline	VERIFIED	none	11.1
E1	Poison injected after signing	FAILED	V6	10.6
E2	Poison before signing, no content checks	VERIFIED	none	13.5
E3	Poison before signing, content checks on	WARNING	V9	12.9
E4	Adaptive attacker evades content checks	VERIFIED	none	10.7
Experiment 7: untrusted external source (scenario G)				
E7-A	Trusted registered publisher signs the model	VERIFIED	none	2.3
E7-A2	Fine-tuned on the trusted external model, deployed	VERIFIED	none	6.2
E7-B	Unknown / unregistered publisher	FAILED	V3, V4	1.7
E7-C	Trusted publisher's model modified after signing	FAILED	V1	1.8
E7-D1	Unsigned external model (no manifest)	REFUSED	none	1.8
E7-D2	Manifest with its signature stripped	REFUSED	none	1.9
E7-E	Trusted publisher key later revoked	FAILED	V4	2.5
E7-E2	Model fine-tuned from it, after revocation	FAILED	V6	6.0
E7-F	Limit: trusted publisher signs a backdoor	VERIFIED	none	2.2

Appendix C A signed manifest

The manifest of the fine-tuned model of the reference run, as stored on disk. Three provenance fields are shortened here for space (marked by three dots), so this printed version is not the signed payload itself. The two parents are the base model and the fine-tuning dataset, each named by its manifest digest.


```
{
  "payload_type": "application/vnd.chainguard.manifest+json; version=1",
  "payload": {
    "schema_version": "1.0",
    "artifact_id": "9f25b977-0aa0-4aa8-a149-4f19d79f3ee2",
    "artifact_name": "digits-finetuned-cnn",
    "artifact_type": "finetuned_model",
    "version": "1.0.0",
    "pipeline_stage": "stage3_finetune",
    "subject": {
      "uri": "artifacts/models/digits-finetuned-cnn-v1.0.0.safetensors",
      "sha256": "1f1448ea86c028d9d36e80b764eca9a52368fdf9847cb901db394f338b16a0ae",
      "size_bytes": 153944,
      "media_type": "application/safetensors",
      "merkle": null
    },
    "parents": [
      { "artifact_id": "a85de012-e158-4221-85e7-826ff7749d5f",
        "manifest_digest": "sha256:1f930c8538ccf7ed0ef95813eed5d8fad5741bad7956e8fab1faf6d8ea82fd21",
        "relation": "fine_tuned_from" },
      { "artifact_id": "3c4e300b-12fe-4883-89c6-cdb3ce96758a",
        "manifest_digest": "sha256:6959cdaa0188c84cf87e0b9b28dfa6d84c0e1f38b68193f5fb49133f212048ea",
        "relation": "trained_on" }
    ],
    "provenance": {
      "source": "internal fine-tuning pipeline",
      "builder": "ml_pipeline/finetune.py",
      "parameters": { "epochs": 8, "lr": 0.0005, "batch_size": 64, "seed": 42 },
      "metrics": { ... },
      "input_verification": { ... },
      "environment": { "python": "3.13.16", "chainguard": "0.4.0", "torch": "2.15.0+cu132", ... }
    },
    "semantic_checks": [],
    "created_at": "2026-11-09T09:26:46Z",
    "signer": {
      "identity": "training_pipeline@chainguard.local",
      "role": "training_pipeline",
      "key_id": "ed25519:cec6a3e317c4a128"
    }
  },
  "signatures": [
    { "key_id": "ed25519:cec6a3e317c4a128", "algorithm": "Ed25519",
      "sig": "n5sYF5FF/diG4Vp88bYcJcsciUHMH9cHh+N8uvt+vXnMwpxp/DCfUaGCEpX3G0yhkzudUNgZMkF3Fuban7syBA=="
    }
  ],
  "manifest_digest": "sha256:133c0274c7b49c52fb8a58504bb8f090a7a28853d8f1e0be98ffd73004f1e0e9"
}
```


The last field is a convenience copy. The verifier never trusts it: it recomputes the digest from the payload.

Appendix D Reproducing the results

From a clone of the repository, with Python 3.11 or later:

```
make install          # dependencies, PyTorch for CPU
make keys             # one Ed25519 key pair per role
make pipeline         # the four-stage attested pipeline and the deployment gateway
make attacks          # the 48 variants of the seven experiments
make benchmark        # cost of hashing, signing and verification
make test             # the 168 tests
python -m scripts.e2_finetune_poison      # supplementary run: poison in the fine-tuning set
python -m scripts.seed_robustness         # supplementary run: ten other seeds
python report/figures/make_figures.py     # the charts of this report
make demo             # the ten-minute demonstration
make standalone       # the dashboard, on port 8501
docker compose -f docker/docker-compose.yml up --build    # pipeline, API and dashboard in containers
make validate LABEL="<machine>"           # the whole validation sequence, with its logs
```

Appendix E Deliverables and where to find them

Deliverable	Location
This report, PDF and LaTeX source	`report/`
Website	`https://ai-chainguard.vercel.app`
Source code, tests, Docker and CI files	`https://github.com/domail2karandeshmukh-cyber/ai-chainguard`
Conference paper, PDF and LaTeX source	`paper/`
Conference proposal	`docs/13_conference_proposal.md`
Scoping note, state of the art, comparative table, threat model, architecture, functional and technical descriptions, attack scenarios, experimentation journal	`docs/`, documents `01` to `09`
Requirements checklist	`docs/18_requirements_checklist.md`
Results: logs, tables, figures, archived earlier runs	`results/`
Demonstration runbook	`docs/15_professor_demo_10min.md`
Earlier progress reports of 5 and 16 October 2026	Folder *Previous reports* of the hand-in package