Executable Trust: The Runtime Architecture of Production-Ready Enterprise AI

A Governance-as-Code Framework for Verifiable, Enforceable, and Measurable Enterprise AI

Moataz Mahmoud

Founder & Research Director, M8 Strategies

RESEARCH PAPER
Version 1.0
July 2026

REFERENCE IMPLEMENTATION
Version 0.1.0
GitHub Release

ARCHIVED SOFTWARE
Zenodo DOI
10.5281/zenodo.21473597

Abstract

Enterprise AI cannot be trusted by policy, intent, or controls written in documents. It requires a runtime architecture where trust is specified, enforced, measured, and ratified as code. We introduce Executable Trust, a governance framework with four properties — Versioned, Enforced, Independently Measured, and Ratified — and show how it operationalizes accountability across the full AI lifecycle.

We present a five-layer reference architecture, a concrete implementation, and a reproducible evaluation baseline. Our results show that enterprise AI systems can achieve verifiable trust through deterministic enforcement and independent measurement. The companion repository provides the full implementation, contracts, datasets, evaluation harness, and reproducibility instructions.

Citation

Mahmoud, M. (2026). *Executable Trust: The Runtime Architecture of Production-Ready Enterprise AI* (Research Paper No. 02, Version 1.0). M8 Strategies. https://doi.org/10.5281/zenodo.21473596

Companion Resources

Paper
https://www.m8strategies.com/blog/executable-trust

Implementation
https://github.com/m8strategies/executable-trust-reference

Software Archive (DOI)
https://doi.org/10.5281/zenodo.21473597

PUBLICATION METADATA

AUTHOR

Moataz Mahmoud

ORGANIZATION

M8 Strategies

VERSION

1.0 · Research Paper No. 02

PUBLICATION DATE

July 2026

REPOSITORY

github.com/m8strategies/executable-trust-reference

PAPER URL

m8strategies.com/blog/executable-trust

SOFTWARE DOI

10.5281/zenodo.21473597 (v0.1.0)

LICENSE

Apache 2.0 (reference implementation)

SUGGESTED CITATION

> Mahmoud, M. (2026). *Executable Trust: The Runtime Architecture of Production-Ready Enterprise AI* (Research Paper No. 02, Version 1.0). M8 Strategies. https://doi.org/10.5281/zenodo.21473596

ABOUT THIS PAPER

Type	Reference White Paper
Audience	CIOs, CDOs, Chief Architects, AI Governance Committees, Platform Engineering Leaders
Thesis	Enterprise trust is an executable runtime capability, not a document. This paper names that capability, defines it by four properties, and shows how to build it. Governance as Code is the engineering discipline that builds it.

CONTENTS

01 Why This Matters Now 4

02 The Thesis: Trust Is a Runtime Capability 5

03 Why Documents Cannot Do This Job 6

04 The Trust Layer in a Live Request 8

05 Numbers, Not Adjectives 11

06 The Trust Stack: Seven Layers, Four Properties 12

07 What Actually Becomes Code 18

08 Trust Begins at Intent, Not at the Answer 23

09 The Stack Already Exists 25

10 Five Stages, and Where Organizations Stall 27

11 An Honest Accounting 28

12 Ten Questions a Policy Cannot Answer 29

13 What Remains Unsolved 31

14 Trust Becomes Programmable 32

15 Artifact Availability and Scope 33

16 References 35

SECTION 01

Why This Matters Now

Enterprise AI has an adoption problem, and it is not about model quality.

RAND's 2024 study of AI project failure reports that by some estimates more than 80 percent of AI projects fail, roughly twice the failure rate of comparable IT projects that do not involve AI. Its root-cause analysis points at misaligned purpose, inadequate data foundations, and organizational factors. Not model capability.

Independent sources find the same shape. Gartner predicted in July 2024 that at least 30 percent of generative AI projects would be abandoned after proof of concept by the end of 2025, citing poor data quality, inadequate risk controls, escalating costs, and unclear business value. MIT's Project NANDA found in 2025 that 95 percent of enterprise generative AI pilots were failing to deliver measurable financial return. S&P Global Market Intelligence found 42 percent of companies had abandoned most of their AI initiatives, up from 17 percent the year before, with the average organization scrapping nearly half of its proofs of concept before production. McKinsey reports that 51 percent of organizations using AI have experienced at least one negative consequence from it, with inaccuracy among the most frequently cited and now one of the two risks organizations most actively try to mitigate.

Read those findings together and a pattern emerges. The models work. What fails is everything around them: the part that decides when to trust a model's output, what to do when it is wrong, and how to prove any of that to a skeptical stakeholder afterward. In most organizations that part is a document nobody checks the running system against.

The fix is not better documents, better committees, or better principles. It is a change in what trust *is* inside an enterprise platform.

SECTION 02

The Thesis: Trust Is a Runtime Capability

Here is the mental model, stated once and defended through the rest of the paper.

Executable Trust is enterprise trust implemented as a runtime capability. It is something a platform *does* on every request, not something an organization *writes* about its platform. It has an architecture, a latency budget, failure modes, version numbers, and a test suite. It can be deployed, measured, regressed, and rolled back. When a customer, an auditor, or a regulator asks "why should we trust this system," the answer is a running subsystem you can point at, not a PDF you can attach.

The capability is defined by four properties. Everything that follows elaborates them.

EXECUTABLE TRUST IS TRUST THAT IS

1. **Versioned.** The rules carry a version identifier. Every behavior change traces to a specific rule revision, not to "the model got updated" or "someone changed a prompt."
2. **Enforced.** The system does not log a violation. It refuses to proceed. Compliance is structural, not aspirational.
3. **Independently Measured.** Something outside the enforcement path checks whether the enforcement path does what it claims. A system that only tests itself grades its own homework.
4. **Ratified.** When a rule must change, the change is a deliberate, documented, dated act. Never a silent edit.

Every architecture decision, code example, and failure story that follows is one of these four properties made concrete.

Governance is one component of the capability. So are identity, context authorization, policy evaluation, evidence verification, decision management, continuous evaluation, telemetry, resilience, and human accountability. None is sufficient alone. Together they form a layer spanning the API gateway, orchestration, retrieval, context management, decision services, observability, and the CI/CD pipeline that ships all of it. Trust, in this model, is not a feature of any component. It is a property the architecture either enforces or does not, the same way availability is.

I have built and operate a platform on this discipline. The numbers and failures cited through this paper come from that system. I will not name it, because the argument does not depend on it; it depends on whether the mechanisms hold, and every one of them is reproducible.

A public reference implementation accompanies this paper. It provides an independent, publication-safe implementation of the versioned trust contract, fail-closed decision function, lifecycle enforcement, amendment process, telemetry boundaries, and evaluation harness described in the sections that follow. The repository draws on lessons from that production platform while remaining separate from it.

SECTION 03

Why Documents Cannot Do This Job

Every enterprise building AI today has a governance document somewhere. A responsible-AI policy. A model risk framework. Principles about transparency, fairness, and human oversight. Most of these documents are well written, well intentioned, and disconnected from the systems they describe.

The standard critique of document-based governance is that it is slow and expensive. That critique misses the actual problem. The actual problem is categorical: **a document cannot execute a runtime decision.** A policy PDF cannot examine the evidence behind this answer, right now, and refuse it. A quarterly review board cannot gate a system that makes a thousand governed decisions a day; by the time the board convenes, the quarter's decisions have already shipped, unexamined.

Document governance operates at human cadence. AI platforms operate at request cadence. The mismatch is not an inefficiency to optimize. It is a capability gap that no amount of committee diligence closes, and it widens as systems become agentic, because agents do not wait for the review cycle.

So the document and the system drift apart quietly. Nothing in the code checks whether a decision matches the policy. Nothing fails when the two diverge. The drift is discovered by an incident, an audit, or a customer question, and the answer at that point is usually "the document was aspirational."

Platform engineering has faced this exact structure three times and solved it the same way each time.

Infrastructure as Code took infrastructure intent out of runbooks and made it executable. Drift between the described environment and the real one became a build failure instead of a postmortem finding.

Policy as Code took authorization intent out of access-review spreadsheets and made it executable. Whether a principal may perform an action became a per-request decision by a policy engine, not a quarterly attestation.

Security as Code took security intent out of hardening guides and made it executable. Controls became pipeline gates and scanners, and a violated control became a failed build.

Each move has the same shape: a category of organizational intent that lived in documents becomes versioned, enforced, independently testable code, and an entire class of silent drift disappears. **Governance as Code** is the fourth application of the pattern, aimed at the last category still living in documents: the rules for what an AI system may assert, decide, and act on. It is the engineering discipline. Executable Trust is what the discipline produces.

SECTION 04

The Trust Layer in a Live Request

The components of Executable Trust are not bolted together after the fact. They compose into one request path plus an asynchronous measurement loop that never touches that path's latency. The diagram shows where each trust function sits in a single request, and where one versioned contract feeds multiple enforcement points.

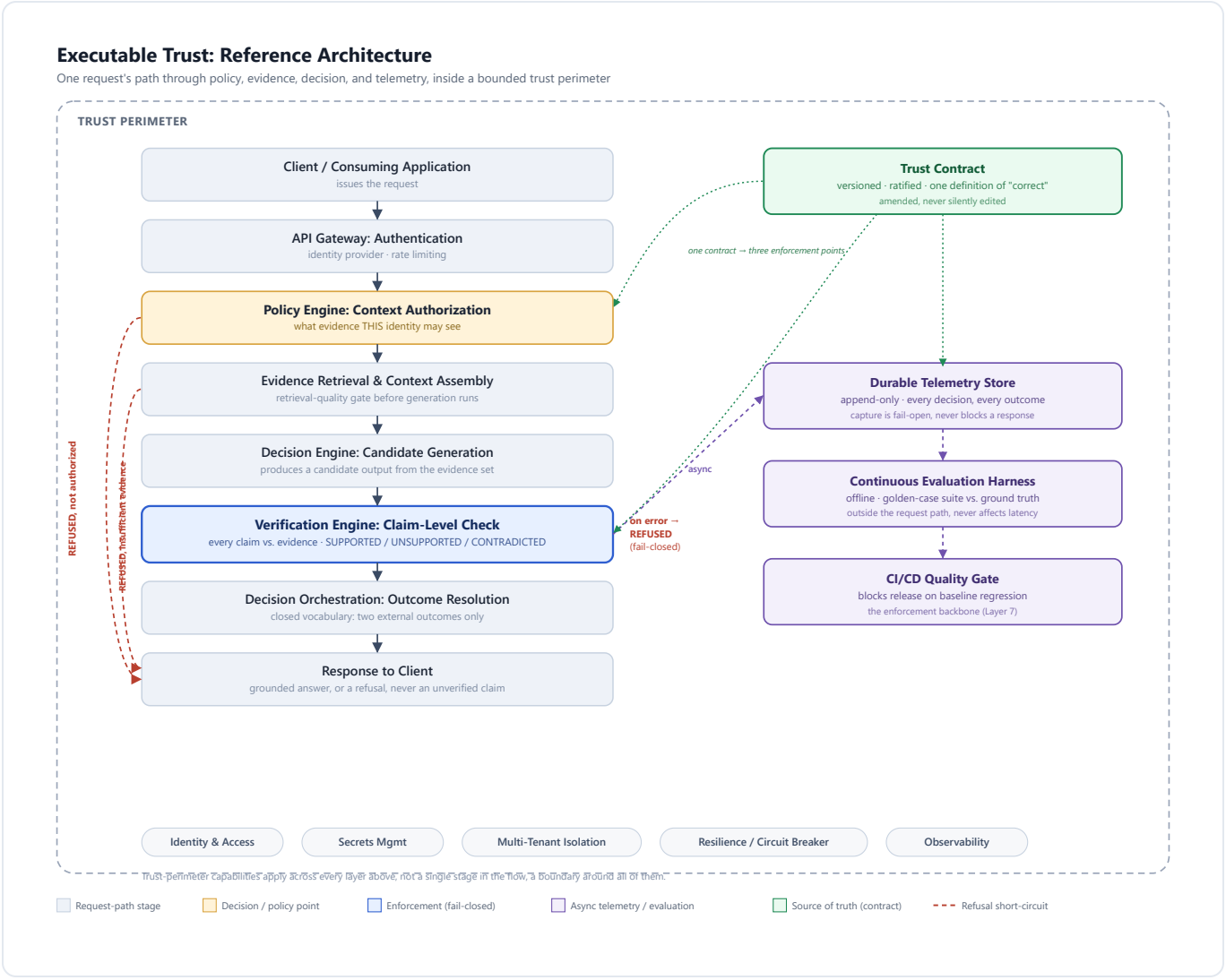


Figure 1. Executable Trust reference architecture: layered request path within a trust perimeter, with a versioned trust contract feeding policy authorization, verification, and evaluation.

Read the diagram as a layer, not a box. Identity is enforced at the gateway. Context authorization is enforced before retrieval. Evidence sufficiency is enforced before generation. Claim verification is enforced before response. Telemetry, evaluation, and the CI/CD gate operate behind the request path. No single component is "the trust feature." The capability is the composition, which is why treating trust as one more microservice to procure misses the point.

Three decisions in this composition carry the weight.

Refusals are short-circuits, not error handlers. A request that fails authorization or evidence sufficiency exits before the decision engine runs. Refusing early is cheaper, safer, and more honest than generating an answer and discarding it. Generation that never happens cannot be trusted incorrectly.

One contract feeds three points, not three copies of a rule. The policy engine, the verification engine, and the evaluation harness implement the same versioned specification. If the verification engine and the evaluation harness ever disagree on the same case, one of them has a defect. That is not a difference of professional opinion, because both claim to implement one document. This is the Versioned property doing structural work.

Telemetry and evaluation sit outside the request path. Proving trustworthiness after the fact should cost the user nothing. Every dashed line in the diagram marks work that happens asynchronously, so the synchronous path stays fast and the measurement path stays rigorous.

The runtime sequence makes the same architecture explicit at the message level.

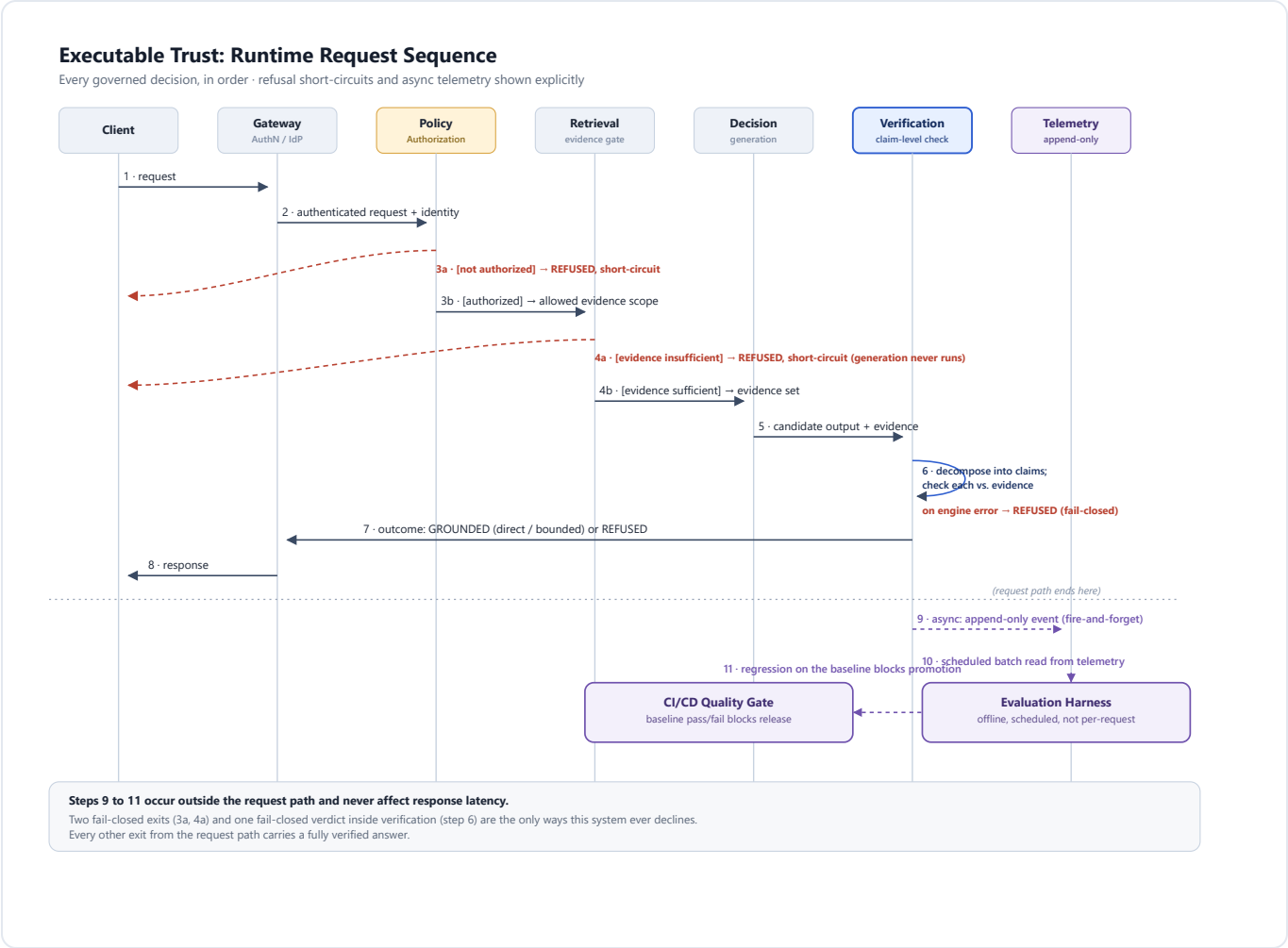


Figure 2. Executable Trust runtime sequence: an ordered request across gateway, policy, retrieval, decision, and verification, with refusal short-circuits and async telemetry.

The sequence supports one claim worth stating plainly. There are exactly three ways this system declines to answer: authorization failure, evidence insufficiency, and a failed or unfavorable claim-level check. Each is fail-closed by construction. Every other exit carries a response that passed full verification. That is a testable property of the architecture, not a promise about it.

SECTION 05

Numbers, Not Adjectives

An executable capability implies operational targets you can violate. The table below shows the kind of numbers a production platform sets for itself. They are illustrative; regulatory context and risk tolerance move them. What is not negotiable is having numbers at all, reviewed on a cadence, instead of words like "fast" and "reliable" that cannot fail an alert.

Control	Illustrative target	Why it sits there
Verification call budget (added to request latency)	150 to 400 ms at p50, capped under the response SLO	The claim-level check runs in-path; without a ceiling it becomes the slowest step in every request
Overall response latency (p95)	Under 2 to 3 seconds end to end	Past this, users treat the system as unreliable regardless of correctness
Verification engine fault rate	Alert in the low single digits (%)	Separates "the checker is unwell" from "the evidence did not support the claim"
Minimum sample before displaying a rate	10 to 30 decisions in the window	Below this a percentage is noise wearing a decimal point
Golden-case regression threshold	A drop of 1 to 3 points on the baseline blocks promotion	Silent regressions compound across releases if nothing catches the first one
Telemetry capture success rate	99.9% or higher of decisions logged	Capture is fail-open by design, so this tracks system uptime, not model accuracy
Audit retention	12 months minimum; 7 years in regulated financial services	Set by the longest regulatory look-back, not by storage cost
Secrets rotation	30 to 90 days, shorter for high privilege	The first thing an external security review checks, and the first thing AI-first teams skip
DR objectives (RTO / RPO)	Defined and drilled, not documented	An undrilled DR plan is a hope

These numbers belong on a dashboard the platform team watches, not only in a table. A target never displayed against real data is a policy document in engineering units.

SECTION 06

The Trust Stack: Seven Layers, Four Properties

The seven layers below are how the four properties become architecture. Layer 1 is Versioned. Layer 2 is Enforced. Layer 3 is Independently Measured. Layer 4 is Ratified. Layers 5 through 7 apply the same four properties to the evidence *about* the system: its telemetry, its dashboards, and its review practice.

Layer 1: The Contract as Source of Truth (Versioned)

Everything starts with a single document, the trust contract, that defines operationally what "trustworthy" means for this system. For a system answering questions from a governed evidence base, the contract must answer at least these:

- What counts as evidence, and what does not?
- What is the smallest unit of a claim that can be checked against evidence: a sentence, a fact, a causal statement?
- What happens when part of a question is answerable and part is not?
- What are the only outcomes the system may produce? Not "grounded, mostly grounded, probably fine." A small closed set a downstream system and a human can both reason about.
- What happens when the verification mechanism itself fails?

Write this before implementation. A contract retrofitted onto existing behavior describes what the code already does, which defeats its purpose. The contract exists to be the thing the code is measured against, and that only works if it existed first.

Layer 2: Runtime Enforcement (Enforced)

The contract is inert until something enforces it on every live decision, in the request path. This is the layer most governance efforts skip, because it is the hardest: it treats the model's own output as a claim to be checked, not a conclusion to be trusted.

The mechanism that works is decomposition. Break the output into individually checkable claims and check each against the evidence the system actually retrieved for this request. Not general world knowledge. Not what sounds right. Each claim is supported by that evidence, unsupported, or contradicted. There is no fourth category where a confident phrasing earns a pass.

Two rules make the layer trustworthy rather than merely present.

Fail-closed is an invariant, not a preference. When the verifier itself errors, through timeout or malformed output or anything else, the default is silence. The most dangerous design decision available in this space is a verification step that lets unverified output through when it breaks. A verifier that fails open is a false sense of safety.

The enforcement mechanism must be swappable without touching its callers. Verification sits behind a narrow interface. The rest of the system depends on the interface, never on the model, prompt, or library beneath it. In my implementation this rule paid for itself within two days: an independent review triggered four fixes to the verification engine, and every one of them landed without a single caller changing. A verifier that requires rewriting its callers to replace will never be replaced.

Layer 3: Independent Measurement (Independently Measured)

Runtime enforcement answers "did this decision follow the rules." It cannot answer "are the rules working," because it is the same mechanism grading its own output. That takes a second path: an offline harness that replays a curated set of known-answer cases through the real production decision path and grades results against ground truth a human wrote in advance.

The harness has no latency budget, so it affords a more careful verdict than the runtime check. Its job is to be the reference the fast check is compared against. One discipline is critical: the offline evaluator and the runtime enforcer share one definition of correct, because both implement the Layer 1 contract. A disagreement between them is definitionally a bug.

The strongest argument for this layer is what it catches, and I have two cases from my own system.

The first: a replacement verification engine passed every unit test its author wrote, then scored 78 percent against a 30-case golden baseline whose incumbent mechanism scored 90. The regression traced to two prompt defects: paraphrased declines were being scored as grounded answers, and legitimate multi-source synthesis was being refused as unsupported. Both were fixed before merge. The shipped engine measured 93 percent overall with 12 of 12 refusal cases correct, and later hardening changes re-ran the identical suite and held the number. Without a baseline that predated the change, the 78-percent engine would have shipped, and nobody would have learned of it until a customer did.

The second cuts the other way, and matters more. A faster, cheaper model tier was evaluated as a replacement for a classification path. It measured 91.1 percent agreement against a required threshold of 98. It did not ship. The measurement also surfaced something nobody was looking for: two services silently routing to a deprecated model and falling through to keyword heuristics, a live defect found as a side effect of establishing the baseline. That is what an independent measurement path buys. It stops bad changes, and it finds problems you did not know to ask about.

Layer 4: Change Control for the Contract Itself (Ratified)

Eventually the contract and the implementation disagree, not through carelessness but because building the thing surfaces a tension the specification did not anticipate. That is the process working: the contract said one thing, an independent check found the running system doing another, and now there is a decision to make.

The wrong response is letting the code win quietly, editing the contract to match whatever shipped. That is governance as documentation with better formatting. The right response is a ratified amendment: a numbered, dated addition stating what changed, why, what trade-off is accepted, and, where relevant, a constraint that closes the loophole for whoever touches the code next.

Here is the real case from my system. The contract specified that a partially supportable answer should be repaired in place: state what is supported, disclaim what is not. The implementation instead refused the entire answer whenever any part was unsupported. The engineer's reasoning: a repair produced by the same mechanism that produced the flawed answer is itself an unverified generation. You cannot trust a patch you have not checked, and checking it puts you back at the start. An independent review caught the divergence. The resolution was neither "fix the code" nor "quietly fix the document." It was an amendment that changed the contract's rule, recorded why refuse-and-ask-again beats repair-and-hope, accepted explicitly that a user sometimes loses an answer they could have partially had, and pre-committed that any future repair mechanism must pass its own output back through verification. The disagreement improved the system and left an attributable record of why.

The record is the point. Six months on, nobody remembers the reasoning behind a one-line code change. Everybody can read a dated amendment. This is also the layer that handles **trust drift**, the slow divergence between stated rules and running behavior that document-based programs cannot even detect. Drift still happens in an executable program. The difference is that it surfaces as a measurable disagreement between contract and code, and the amendment process metabolizes it into an improvement instead of a liability.

Layer 5: Durable Telemetry, Honest About What It Is

Every governed decision leaves a permanent, append-only record: what was decided, by what mechanism, under which contract version, and how long it took. This is the raw material for every trust claim the organization will make to a customer, an auditor, or its own board.

Four disciplines keep the telemetry trustworthy, and I learned three of them the uncomfortable way.

Store facts, not derived scores. Under a refuse-anything-imperfect enforcement rule, the verification score of every emitted answer is mathematically constant: 1.0, always. My system was displaying that constant as "confidence: 100%" on every answer. Technically true and completely uninformative, which in a trust product means dishonest. The fix was structural: the telemetry stores what varies (verification path taken, evidence volume, check duration) and any display summary is derived at read time from real variance.

One field, one meaning, forever. The same review found a boolean that meant "the outcome was compliant" in one code path and "the behavior taken was correct" in another. Harmless in a log file. In a persisted table it is a contradiction waiting for whoever queries it next. Both this and the constant-confidence defect were deliberately fixed at the persistence boundary, before the first row was written, because after data accumulates the fix becomes a migration. Semantics defects are cheapest at the boundary where the data is born.

Know whose behavior your telemetry describes. Within a day of the telemetry table going live, its averages were wrong. The automated test suite exercises the real production paths, so every test run wrote fixture decisions into the table: stubbed verifications with near-zero latency, deliberately injected faults, synthetic refusals. The dashboard was faithfully aggregating a mixture of real behavior and test traffic and reporting numbers true of neither. Telemetry integrity is not only about capture reliability. It is about provenance: knowing which recorded decisions belong to the population you claim to describe.

Capture must never break the thing it watches. A telemetry write is less important than the decision it describes. In my system the first live proof of this rule was accidental: the capture path hit a missing database table mid-request, logged one warning, and the user received the identical verified answer. That is the mirror image of Layer 2. Enforcement fails closed; telemetry fails open. Reversing either direction is a design defect.

Layer 6: Consumer-Side Honesty

The dashboard on top of this telemetry is where trust discipline survives contact with a product owner, or does not. Two rules keep it honest.

Withhold rates below a minimum sample; never withhold counts. A grounded-rate of 100 percent computed from three data points is worse than no number. The interface says "not enough data yet, here is what we have observed" rather than displaying a rate that is mathematically valid and practically meaningless.

A behavioral dashboard never claims a quality number it has not earned. Telemetry describes what the system did. Only the Layer 3 harness, with human-authored ground truth, can say how often it was right. Blending "how often we answered" with "how often we were correct," using whichever number is more impressive where it is needed, is where Executable Trust reverts to governance as marketing. My implementation states the boundary in the interface itself: this panel describes behavior; quality is measured by the offline baseline, and the panel names it.

Layer 7: Independent Review as an Institution, Not an Event

The most useful discovery in building this capability is that the people who wrote the contract and the code will not find most of the gaps between them. Not through carelessness. The gaps live in the seams between documents, code, and downstream consumers, and each author is, by construction, the person least positioned to see where their piece stops lining up with the others.

The concrete case: a subsystem in my platform shipped green across a suite of roughly 850 tests, written by a careful author against a ratified contract. An independent review that read the running code first and the documents second surfaced six real defects. One was the constant-confidence problem above. One was a fail-open edge where two individually reasonable defaults compounded: an empty claim list scored as fully verified, and a missing verdict field defaulted to true, so a degenerate verifier response could promote an unchecked answer. One was the contract divergence that became the Layer 4 amendment. Unit tests missed all six because unit tests encode the author's assumptions, and the defects lived in the assumptions.

This is not an argument for a one-time review. It is an argument for a standing practice, applied to every subsystem that makes governed decisions, on a cadence. The gap class is structural to how software gets built.

SECTION 07

What Actually Becomes Code

"The policy becomes executable" is easy to say and easy to leave abstract. Four translations follow: a policy statement in plain English, then its executable form in illustrative pseudocode.

A Verification Policy Becomes a Decision Function

> **POLICY** *The system may only present a claim as fact if it is entailed by evidence actually retrieved for this request. If any claim is unsupported or contradicted, the system must not present it as fact.*

```
def decide(claims: list[ClaimVerdict], answers_question: bool) -> Outcome:
    if not claims:
        return Outcome.REFUSED(reason="empty_claim_set")
    if not any(c.verdict == SUPPORTED for c in claims):
        return Outcome.REFUSED(reason="no_supported_claims")
    if any(c.verdict in (UNSUPPORTED, CONTRADICTED) for c in claims):
        return Outcome.REFUSED(reason="unsupported_claim")
    if not answers_question:
        return Outcome.REFUSED(reason="does_not_address_question")
    strategy = BOUNDED if has_scope_limitation(claims) else DIRECT
    return Outcome.GROUNDED(strategy=strategy)
```

The policy sentence and the function are the same rule in two languages. That is the entire discipline. Not clever code; zero daylight between what the document says and what the function does.

The guard order is part of the policy, not an implementation detail. Every clause above refuses, so reordering changes no outcome. What it changes is which reason code gets recorded, and reason codes are the audit surface. Three of the four guards exist to keep failures distinguishable in telemetry.

- `empty_claim_set` is the fail-open edge from Layer 7 made explicit. A verifier that decomposes an answer into zero claims has verified nothing, yet an empty list trivially satisfies "no claim was unsupported." Written as `all(c.verdict == SUPPORTED for c in claims)`, that check returns true for an empty list and promotes an answer nobody examined.
- `no_supported_claims` is evaluated before the unsupported check so that "nothing here was supportable" and "part of this was supportable" carry different codes. A run of the first points at retrieval; a run of the second points at generation. Collapsing them hides which system is misbehaving.
- `does_not_address_question` separates relevance from grounding. Every claim can be supported and the answer still not be an answer.

Two further guards precede this function in a working implementation and are omitted above for clarity. A truncated verifier response is refused before its claims are parsed: a response cut off at a size cap can parse into a prefix of the intended claim set, and the unexamined tail is where the unsupported claim most likely sits. A claim carrying a missing or unrecognized verdict is refused as malformed output rather than defaulted to anything.

There is no default value for any required field anywhere in this path. A default that lets an unchecked claim through is the fail-open shape this architecture exists to prevent.

An Approval Workflow Becomes a State Machine

> **POLICY** *A generated decision record must be reviewed by an accountable human before it can inform future recommendations. Rejected alternatives are retained, not deleted, so the organization does not repeat a path it already evaluated and declined.*

From state	Transition	To state	Trigger
PROPOSED	human review, approve	ACCEPTED	accountable reviewer
PROPOSED	human review, reject	REJECTED (terminal, retained)	accountable reviewer
PROPOSED	direct promotion attempt	*rejected at the data layer*	invalid transition, not merely hidden in the UI
ACCEPTED	superseded by newer decision	SUPERSEDED (append-only link, original retained)	system, on new acceptance

The third row is the design decision that matters. The invalid transition is rejected by the persistence layer, not absent from a dropdown. A hidden UI control is bypassed by anyone calling the API directly. A state machine enforced in the data layer is not. This is also where **human accountability** stops being a principle and becomes a mechanism: the reviewer is a named actor in a state transition, recorded immutably, not a sentence in a policy about oversight.

An Evidence Threshold Becomes Enforcement Logic

> **POLICY** *Answer only when retrieved evidence quality clears a minimum bar. Decline before generation runs, rather than generating an answer and discarding it.*

```
RETRIEVAL_GATE = {
    "min_similarity_score": 0.35,
    "min_chunks_required": 1,
    "top_k": 4,
}

if (retrieval.top_score < RETRIEVAL_GATE["min_similarity_score"]
        or retrieval.chunk_count < RETRIEVAL_GATE["min_chunks_required"]):
    return Outcome.REFUSED(reason="insufficient_evidence")
    # generation is never invoked; the refusal is pre-generation
```

This is the insufficient-evidence short-circuit from the diagrams, made literal. The thresholds are configuration, reviewable and changeable without a deploy. That is itself a trust property: the people accountable for the evidence bar can see and adjust the number, not just the intent.

A Resilience Policy Becomes a Circuit Breaker

> **POLICY** *If a downstream verification dependency degrades, the system degrades gracefully. It never skips verification to keep answering.*

```
if verification_engine.recent_error_rate() > CIRCUIT_BREAKER_THRESHOLD:
    escalate_to_conservative_tier()   # a more expensive, more reliable tier
    alert_on_call()
    # never: fall back to "skip verification and answer anyway"
```

Teams get this one backwards under production pressure. When a dependency is unhealthy, the tempting fix is bypassing it so the user experience holds. The circuit breaker is allowed to make the system slower or more conservative. It is never allowed to make the system less trusted.

Companion Reference Implementation

The policies in this section are implemented as versioned contracts, typed domain models, deterministic decision functions, schemas, lifecycle rules, tests, and an offline evaluation harness in the companion repository. The artifact makes the paper's mechanisms inspectable and reproducible without publishing the proprietary production platform.

Repository	github.com/m8strategies/executable-trust-reference
Release	`v0.1.0`
Tests	`pytest`
Baseline	`python evaluation/run_baseline.py --check`
Traceability matrix	`docs/paper-to-code-traceability.md`

The four translations above map to `src/executable_trust/verification/decision_function.py`, `src/executable_trust/lifecycle/state_machine.py`, `src/executable_trust/evidence/gate.py`, and `src/executable_trust/verification/resilience.py` respectively. Each rule in the contract carries a stable identifier, and a build-time check fails when any identifier has no test naming it.

SECTION 08

Trust Begins at Intent, Not at the Answer

Everything so far verifies a response: a model produced output, and the trust layer checked it. Stopping there treats Executable Trust as an AI safety control. The larger claim is that it is an **enterprise execution framework**, because the same four properties apply to everything between a business request entering the platform and the organization learning from what it shipped.



Walk the chain and apply the properties at each link.

- **Business intent.** A request enters through a closed intake contract: classified against a fixed vocabulary by inspectable logic, not free-text routed by a model's mood. The intent is versioned from its first moment, so everything downstream traces to it.
- **Requirements.** Completeness is scored against explicit weighted criteria, and lifecycle advancement is a state machine with enforced transitions. "Ready for review" is a gate the system evaluates, not an opinion someone types.

- **Architecture decisions.** Each decision is an immutable record with context, rationale, alternatives considered, and consequences. Records are never edited in place; they are superseded with a link to the original. Rejected alternatives are retained, so the organization can detect when a new initiative is about to repeat a path it already evaluated and declined.
- **Enterprise context.** The evidence base itself is a governed asset: what is in it, who authorized it, which version of it informed which decision. An answer verified against an ungoverned corpus is a precise measurement of an unknown quantity.
- **Execution.** Actions against external systems (creating work items, submitting artifacts, triggering workflows) pass through the same gate discipline as answers: authorized, approved by an accountable human where the contract requires it, and logged append-only.
- **Organizational learning.** What the system learns from accepted decisions feeds back into future recommendations only after human ratification. The learning loop is governed like everything else, because an ungoverned learning loop is drift with a flywheel.

This chain is what makes the framework durable through the agentic transition. An agent is a system that acts, not just answers. Every hard question about agent safety (what may it do, on whose authority, against what evidence, with what audit trail, reversed how) is already answered by the governed chain above. Organizations that build Executable Trust for their AI responses today are building what makes delegating execution to agents survivable tomorrow. Organizations that govern agents with documents will rediscover every failure mode in this paper, at machine speed.

SECTION 09

The Stack Already Exists

Executable Trust maps to production capabilities most enterprise platform teams already own pieces of. The table is vendor neutral; the right choice in each category depends on what the platform already runs.

Trust layer	Capability	Technology categories	Illustrative examples
Contract and policy store	Versioned, ratified rules	Policy-as-code engines; version-controlled config	Open Policy Agent, AWS Verified Permissions, a Git-tracked policy repository
Identity and access	AuthN/AuthZ, RBAC, audit	Identity providers; authorization services	Okta, Microsoft Entra ID, Auth0, AWS IAM/Cognito
Context authorization	Scope and visibility enforcement	Attribute- or row-based access control	PostgreSQL row-level security, OPA, custom middleware
Evidence verification	Claim-level checking, fail-closed	LLM-as-judge and NLI models; custom verification services	In-house verification services, NLI cross-encoders, evaluation SDKs
Decision orchestration	Deterministic outcome routing	Workflow and state-machine engines	AWS Step Functions, Temporal, a custom state machine
Durable telemetry	Append-only event capture	Event streaming; append-only stores	Kafka, AWS EventBridge, append-only Postgres tables, WORM object storage

Trust layer	Capability	Technology categories	Illustrative examples
Continuous evaluation	Offline measurement, regression gating	Evaluation harnesses; experiment tracking	Custom harnesses, MLflow-style tracking
CI/CD enforcement backbone	Gate releases on evaluation	CI/CD pipelines	GitHub Actions, GitLab CI, Jenkins, ArgoCD
Secrets management	Rotation, least privilege	Secrets managers and vaults	HashiCorp Vault, AWS Secrets Manager, Azure Key Vault
Multi-tenant isolation	Data and schema isolation	Row-level security; schema-per-tenant	PostgreSQL RLS, dedicated-schema isolation
Observability	Tracing, SLO monitoring	APM and observability platforms	OpenTelemetry, Datadog, Grafana with Prometheus
Resilience	Circuit breaking, graceful degradation	Resilience libraries; service mesh	Istio, standard circuit-breaker libraries
Immutable audit	Non-repudiation, long retention	WORM storage; append-only logs	S3 Object Lock, database-enforced append-only constraints

The table exists to make one claim credible: every layer in this paper is buildable with tools the team likely already licenses. Executable Trust does not require a new vendor category. It requires pointing capabilities you already own at a class of risk they were not bought for.

SECTION 10

Five Stages, and Where Organizations Stall

Organizations do not jump from documents to Executable Trust in one step. The stages below describe the path, and what typically stalls each one.

Stage	Name	Defining characteristic	Typical gap
0	**Document-based**	Policies exist as prose; nothing in the running system enforces them	Behavior and policy diverge silently, discovered by audit or incident
1	**Manual enforcement**	Checklists, human review gates, spreadsheet exception tracking	Does not scale; inconsistent across teams and time
2	**Partial automation**	Some runtime checks; telemetry collected but not systematically reviewed	No independent measurement; dashboards can silently overclaim
3	**Enforced, unverified**	Versioned contract; fail-closed runtime enforcement	Nothing checks whether the enforced rules remain correct as the system evolves
4	**Executable Trust**	All seven layers operate; CI/CD gates releases on the baseline; contract changes are ratified amendments	The remaining perimeter components (identity depth, isolation, resilience, DR) are usually still in flight

This model covers decision trust specifically. The wider architecture matures on its own curve, and an organization can be genuinely at Stage 4 on decisions while still building tenant isolation or rehearsing disaster recovery. That is a sequencing choice, not a contradiction. The discipline is naming it honestly.

SECTION 11

An Honest Accounting

The Capability Is Never Finished All at Once

The trust architecture spans more than the decision path, and no platform builds all of it simultaneously. Mine has not. I include this section because a maturity claim that omits its own gaps fails the standard this paper sets.

- **Identity and access as a first-class system, not an environment variable.** Authentication, role-based authorization, and an audit trail of who did what. These land late in most AI-first builds because model work is more urgent and more visible. Name the gap explicitly rather than letting "we have an API key" stand in for "we have access control."
- **Secrets management with rotation.** Not credentials in a `.env` file that nobody has scheduled to rotate. Unglamorous, and the first control an auditor checks.
- **Tenant isolation**, if the platform will ever serve more than one customer from shared infrastructure. Enforced at the schema or data-access layer, with isolation tests that actively try to breach the boundary. A `tenant_id` column with an honor system is not isolation.
- **Model-upgrade gating as standing practice.** A new model version changes the system's judgment and deserves the same treatment as a code change: through the Layer 3 harness, against the existing baseline, before promotion. The 91.1-versus-98 case in Layer 3 is what this looks like when it works: the upgrade that measured well but not well enough did not ship.
- **Resilience as an explicit signal.** Per-dependency health and a circuit breaker that degrades to a more conservative tier, rather than a system that either works perfectly or fails opaquely.
- **Operational readiness.** Defined SLOs, an incident process, and a disaster-recovery drill that has been run and timed. A trust program that has never rehearsed its own failure is a hope.
- **CI/CD as the enforcement backbone.** The evaluation baseline becomes a control only when a pipeline gates every release on it. A human remembering to run the suite is a process. A pipeline that refuses to ship without it is a control. The same holds for security scanning and the isolation tests above.

Decision-level trust is visible to users and regulators first, so building it first is defensible sequencing. What is not defensible is claiming platform maturity while auth is an environment variable. That team has broken the Ratified property in the roadmap instead of the dashboard.

SECTION 12

Ten Questions a Policy Cannot Answer

A team can self-assess with honest yes/no answers and no tooling. The first four questions are the four properties; the rest probe the layers.

Question	If the answer is no
1. Is there a single, versioned document that defines "trustworthy output" operationally, not aspirationally?	You have a values statement, not a contract.
2. Does the runtime refuse to emit output it cannot verify, by default, on every error path?	You have monitoring, not enforcement.
3. Is there an independent measurement path that shares no code with the thing it measures?	You are grading your own homework.
4. Are contract-versus-code disagreements resolved by dated, ratified amendment, rather than by quietly editing whichever is convenient?	Your documentation will drift from your system within a quarter.
5. Does your telemetry store facts, or a pre-computed score that is secretly constant?	You have a vanity metric, not a trust signal.

Question	If the answer is no
6. Does any dashboard on that telemetry ever say "not enough data yet"?	Your dashboard is optimized for reassurance.
7. Does anything in your product claim accuracy from data with no ground truth behind it?	You have a marketing number in an engineering costume.
8. Has a reviewer other than the author read the running code against the contract in the last quarter?	Your gaps are still there. Nobody has looked.
9. Does your release pipeline enforce the quality gate, or does a person remember to run a script?	You have a testing policy, not a testing control.
10. Can you name, specifically, which parts of your trust architecture (identity, secrets, isolation, resilience, DR) are not yet built?	If you cannot name the gap, you cannot be trusted to be closing it.

In my experience, a team answering these honestly says no to at least half. That is not a condemnation; it is where the industry is, and the research in the opening section is what that looks like at survey scale. The organizations that convert those answers are not the ones with the most sophisticated models. They are the ones whose trust is versioned, enforced, independently measured, and ratified, and who are honest in public about what remains unbuilt.

SECTION 13

What Remains Unsolved

Three problems in this discipline are genuinely open, and anyone adopting it should walk in knowing them.

Verification cost scales with trust. The claim-level check adds a model call to every governed request. At current inference prices that is affordable for high-value decisions and expensive for high-volume ones. Cheaper verifiers (small NLI models, distilled judges) reduce cost but must themselves be baselined against the reference judge, which is more machinery. Nobody has a free verification path yet.

The golden set decays. Ground-truth cases encode the domain as it was when a human wrote them. Corpora change, policies change, terminology drifts, and a baseline that is never refreshed slowly measures the wrong world. Refreshing it requires the same scarce human judgment that built it. Treat golden-set maintenance as a standing cost of the discipline, not a one-time investment.

Who verifies the verifier's upgrades? The evaluation harness gates model changes, but the harness itself contains a judge model. Upgrading the judge changes the measuring instrument. My current answer is to re-run the full baseline under both judges and require agreement before switching, and I do not claim that answer is complete.

SECTION 14

Trust Becomes Programmable

The industry's instinct, faced with the trust problem in enterprise AI, has been to write more policy. More principles, more committees, more PDFs describing what responsible AI should look like. The research at the start of this paper suggests the instinct is not working, and the reason is structural: a document cannot execute a runtime decision. A description of correct behavior, sitting next to a system with no obligation to match it, is a promise, not a control.

The deeper claim of this paper is about what comes next. Autonomous enterprise systems will not scale because models become more intelligent. Intelligence was never the binding constraint; trust is. They will scale when trust becomes programmable: versioned, enforced, independently measured, and ratified, from the moment business intent enters the platform to the moment the organization learns from what it shipped. Infrastructure as Code made environments programmable. Policy as Code made authorization programmable. Governance as Code, applied across the full execution chain, makes trust programmable. Executable Trust is the capability that results, and it is the substrate delegated execution will require.

You do not need to build all of it this quarter. You need to make one move that changes what kind of program you are running: write the contract. One document, operational definitions, a version number, ratified before the next AI feature ships. Every other layer in this paper has a natural place to attach once the contract exists. Without it, everything else is better-instrumented drift, and the gap between what your organization says about its AI and what its AI does remains what it is today: unmeasured.

> **Intelligence was never the binding constraint; trust is. Autonomous enterprise systems will scale when trust becomes programmable.**

SECTION 15

Artifact Availability and Scope

A companion reference implementation accompanies this paper.

Repository	github.com/m8strategies/executable-trust-reference
Release	`v0.1.0`
Archive DOI	10.5281/zenodo.21473596 (all versions) · 10.5281/zenodo.21473597 (v0.1.0)
Canonical paper	m8strategies.com/blog/executable-trust
Supported Python	3.11 and 3.12

```
pip install -e ".[dev]"                    # installation
pytest                                     # tests
python evaluation/run_baseline.py --check  # verify the committed baseline
python scripts/validate_contracts.py       # contract validation
```

The traceability matrix is at `docs/paper-to-code-traceability.md`. It maps every contract rule identifier to its policy statement, source module, tests, and known limitation, and a build-time check fails when any rule has no test naming it.

Scope. The artifact runs with no model provider, no network access, no database, and no credentials. Every example, fixture, evidence passage, actor, document, and evaluation case in it is synthetic and newly authored: the evaluation domain is a fictional company's internal engineering handbook. The repository excludes the production platform, its source code, operational data, prompts, integrations, deployment architecture, and any customer information.

What the artifact demonstrates. It demonstrates that the mechanisms described in this paper are implementable as inspectable, deterministic code, and that a contract, a runtime enforcement path, and an offline harness can share one definition of correct. It does not reproduce the complete production platform.

Distinguishing the evidence. Four kinds of evidence appear in this work and must not be conflated: production observations reported in this paper; normative mechanisms this paper specifies; the behavior of the public reference implementation; and synthetic evaluation results produced by that implementation against its own human-authored golden set. Results published in the repository are labelled *Reference implementation synthetic baseline*. They describe that implementation's conformance to its own contract, are not a measurement of any production system, and are not numerically comparable to the field observations reported here.

The reference implementation includes several mechanisms specified by this paper that extend beyond the production implementation from which some field observations were drawn. In particular, the reference artifact implements explicit verification dependency degradation and telemetry population provenance. These components demonstrate the paper's normative architecture and must not be interpreted as evidence that every mechanism was operating in the production system at the time of the reported observations. A mechanism-by-mechanism comparison is provided in the repository at `docs/production-reference-boundary.md`, and the artifact's known gaps are enumerated at `docs/limitations.md`.

The three artifacts. This paper is the specification and the research argument, and its canonical location is the address above. The repository is an independent reference implementation of the mechanisms the paper describes. The Zenodo record is an archived, citable snapshot of a software release. They are cited differently and should not be substituted for one another: the repository is not a publication venue for the paper, and the paper is not a substitute for reading the code.

Citation. To cite the artifact, use the metadata in `CITATION.cff` at https://github.com/m8strategies/executable-trust-reference; to cite this paper, use https://www.m8strategies.com/blog/executable-trust. Where both are relevant, cite both: the paper is the specification, and the repository is one implementation of it.

License. The reference implementation is licensed under the Apache License 2.0. The paper and its visual assets remain subject to the publication terms established by M8 Strategies.

SECTION 16

References

1 Mahmoud, Moataz, *Executable Trust Reference Implementation*, version v0.1.0, 2026. Zenodo. https://doi.org/10.5281/zenodo.21473596. Companion software artifact to this paper. https://github.com/m8strategies/executable-trust-reference

2 RAND Corporation, *The Root Causes of Failure for Artificial Intelligence Projects and How They Can Succeed: Avoiding the Anti-Patterns of AI* (RRA2680-1), 2024. Based on structured interviews with 65 experienced data scientists and engineers. https://www.rand.org/pubs/research_reports/RRA2680-1.html

3 Gartner, *Gartner Predicts 30% of Generative AI Projects Will Be Abandoned After Proof of Concept By End of 2025*, press release, July 29, 2024. https://www.gartner.com/en/newsroom/press-releases/2024-07-29-gartner-predicts-30-percent-of-generative-ai-projects-will-be-abandoned-after-proof-of-concept-by-end-of-2025

4 MIT Project NANDA, *The GenAI Divide: State of AI in Business 2025*, 2025. Based on 150 executive interviews, a 350-employee survey, and analysis of 300 public AI deployments.

5 S&P Global Market Intelligence, *Voice of the Enterprise: AI & Machine Learning*, survey findings on generative AI adoption and project abandonment, 2025. https://www.spglobal.com/market-intelligence/en/news-insights/research/ai-experiences-rapid-adoption-but-with-mixed-outcomes

6 McKinsey & Company, *The State of AI: How Organizations Are Rewiring to Capture Value*, March 2025. https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai

---

Survey figures reflect the cited editions. Recurring surveys (Gartner, McKinsey, S&P Global) publish updates; check for newer editions before republication.