

Runtime assurance of language-model-based systems: from safety monitors to certification-ready evidence

Kamal Essam

Abstract—Language models are entering systems whose failures matter: vehicles, industrial toolchains, and autonomous agents acting on real infrastructure. At the same time, emerging regulation and safety standards demand not merely safe behavior but evidence of it that a third party can re-check. This survey traces a single architectural idea across three decades of research: an untrusted, powerful component is guarded at runtime by a small, verifiable one, and every intervention leaves evidence. We follow the idea from proof-carrying code, the Simplex architecture, and shielding for learned agents, through runtime monitors for neural components, to its present forms: generate-and-verify systems in which a symbolic engine certifies every step of a model's output, and runtime guardrails that enforce policies over language-model agents. We synthesize a spectrum of verification strength, from statistical judgment to kernel-checked proof, analyze which strength each application class and regulatory obligation requires, and identify the open problem of assurance over large, versioned factual world models.

Keywords—runtime assurance; language models; safety monitors; shielding; verification; certification

I. INTRODUCTION

ASKED to account for its own behavior, a deployed language-model system can produce at best a log. Consequential engineering has long demanded more of its components: evidence that a third party can re-check without trusting the party that produced it. That gap mattered less while language models only produced text. It matters now because the systems built on them act. They execute code, call tools that carry side effects, and operate inside software toolchains and physical workflows. A model that only answers can be wrong at low cost: a degraded answer is read, doubted, and set aside. A model that acts converts the same error into an event. A file is deleted, a transaction is issued, an actuator moves; none of this is undone by regenerating the answer. Such failures must be prevented or contained at runtime, because no scrutiny of the model beforehand reaches the individual action it is about to take.

Regulation is arriving at the same demand from the opposite direction. Legal analysis of large generative models argues that duties should attach to concrete deployed applications rather than to the pre-trained model as such [1]. A three-layered blueprint for auditing language models distinguishes

governance, model, and application audits; the last of these depends on artifacts such as output logs and audit trails [2]. Work on assuring autonomous systems identifies a semantic gap, a responsibility gap, and a liability gap, and frames assurance in the safety-case tradition, where the evidence must carry moral and legal weight [3]. These positions differ in origin and instrument. They converge on the object requested: evidence.

Our thesis is that the requested artifact has a three-decade engineering history, and that the systems most in need of it have rediscovered the architecture while losing the assurance. The architecture is one idea: an untrusted, capable component is guarded at runtime by a small, verifiable one, and every intervention leaves evidence. In proof-carrying code, untrusted native code ships with a machine-checkable safety proof; the consumer validates the proof once with a small checker, in under two milliseconds on 1997 hardware, and thereafter executes the code with zero runtime overhead [4]. Simplex bounds what an untrusted high-performance controller may do with a verified simple controller and a switching rule simple enough to verify, drawing reliability from simplicity rather than diversity [5]. Shielding places a filter synthesized from a temporal-logic safety specification between a reinforcement-learning agent and its environment [6]. Runtime safety monitors carried the pattern to learned components, flagging when a neural network operates outside the experience recorded from its training set [7], [8]. The strongest modern instances pair a language model with a symbolic judge: AlphaGeometry's deduction engine derives every step itself, the model proposing only auxiliary constructions, and the system solves 25 of 30 olympiad geometry problems [9], and AlphaProof, with AlphaGeometry 2, scored in the silver-medal range at the 2024 International Mathematical Olympiad, every AlphaProof proof passing an independent final verification by the Lean checker [10]. Today's LLM guardrails are the newest instance of the same architecture, and they lose the assurance in three specific ways. The guard is often another language model, as in the NeMo Guardrails toolkit, where parts of the checking are themselves done by an LLM, so the trusted base is not small [11]. Runtime policy enforcement over agents' actions is an active line of work [12], [13], but the policies these systems enforce either begin as prose, as when platform documents are compiled into rules, or admit model-written rules and

K. Essam is with TomTom, Warsaw, Poland (e-mail: kessam1@outlook.com; ORCID: 0009-0009-8854-1613).



predicates, so the specification is not formal end to end. And an intervention leaves at best a log entry, so nothing durable remains for a third party to re-check: testimony, not evidence.

A note on scope and method. This is a narrative, argument-led review, not a systematic one. The corpus was assembled in the second half of 2026 from an industrial vantage point, restricted to English-language sources, and reduced to sixteen core papers under three criteria: original research, publication at an indexed venue with peer review, and a distinct contribution to the argument of one section. Four neighboring literatures are deliberately excluded. Training-time alignment methods act before deployment, not at runtime. The jailbreak and red-teaming literature studies attacks rather than assurance. Offline formal verification of neural networks certifies a model before deployment rather than guarding it in operation. Constrained decoding enforces syntactic validity during generation, but its guarantee holds by construction: it yields no verdict and no intervention record, so it falls outside a survey organized around runtime verdicts and their evidence. One caveat is owed: the guardrail literature is young and moving quickly, and we describe its systems as published, not as they may become.

The survey makes four contributions. First, a unified reading of runtime assurance, from proof-carrying code and Simplex (Section II) through monitors for learned components (Section III) and generate-and-verify systems (Section IV) to LLM guardrails and the evidence they should feed (Sections V and VI). Second, a spectrum of verification strength, running from statistical judgment to kernel-checked proof, on which every system surveyed in depth is placed in Section VII. Third, an analysis, built on that spectrum, of which verification strength and which durability of evidence each application class and each regulatory obligation requires. Fourth, a research agenda toward guards that produce durable, re-checkable evidence, drawn in Section VII; its final problem, assurance over large versioned factual world models, is where Section VIII closes.

II. THE RUNTIME-ASSURANCE PARADIGM

Checking a proof is cheaper than finding one, and proof-carrying code is the architecture built on that fact [4]. The consumer publishes a formal safety policy; the untrusted producer compiles the code, proves the corresponding verification condition, and ships the binary with the proof; the consumer validates the proof once, by type checking, and runs the code with no further runtime checks. In Nacula’s worked example the entire package is 730 bytes, 420 of them proof and 60 native code; validation takes 1.9 ms on a 175 MHz workstation, and the validator is a couple of pages of code. Tampering is defeated without cryptography: any modification is either caught by validation or leaves the code still safe. Certification, however expensive, stays offline, validation runs once for any number of executions, and the proof is a durable artifact that a third party can re-check without trusting its author. Nacula is candid about the limits: the proof certifies compliance with the stated policy rather than functional correctness, loop invariants must come from the

producer, and he doubts the simple theorem-proving approach scales to realistic program sizes.

In feedback control the same asymmetry becomes Simplex: a verified, simple baseline controller runs in parallel with an isolated, untrusted high-performance controller, and decision logic switches to the baseline whenever the plant state nears the boundary of its recovery region [5]. The region is computed offline, as the largest Lyapunov-stability ellipsoid inside the polytope of operational constraints, by linear matrix inequality methods; the runtime switching rule stays simple enough to verify. Sha derives the design from a reliability model resting on three postulates, that complexity breeds bugs, that bugs are not equal, and that budgets are finite: under a fixed budget, single-version programming beats three-version programming over a wide range, and a recovery block wins only when its second alternative is much simpler, a five-fold complexity reduction outweighing even an imperfect acceptance test; the active ingredient is simplicity. Boeing 777 flight control constrains its normal controller within the stability envelope of a secondary controller built on 747 control laws with over 25 years of service; the paper also reports applications in semiconductor manufacturing, in F-16 simulator flight control, and in a teaching laboratory whose inverted pendulum runs user-submitted code. Yet the intervention is an event, not a record: Simplex guards, but it does not testify.

Runtime verification supplies the vocabulary the rest of this survey uses: a monitor reads the finite trace of a single run and yields a verdict on a correctness property [14]. Monitors are synthesized from linear temporal logic into finite-state machines with a three-valued semantics (true, false, or inconclusive) under two maxims: a monitor must never deliver a final verdict prematurely, and must deliver one as soon as every continuation of the trace agrees. Because it solves the word problem over one observed execution rather than language inclusion over a model of all runs, the monitor applies to black-box systems. The authors are explicit that the field only detects violations and delegates intervention to extensions; what it establishes is the verdict stream as a durable, semantically defined runtime signal.

Shielding carries the paradigm to learned components [6]. A safety specification in the safety fragment of temporal logic becomes an automaton; its product with a coarse, conservative finite abstraction of the environment is solved as a two-player safety game, and the winning strategy runs as a finite-state reactive shield beside a reinforcement learner. The shield acts preemptively, handing the agent only safe actions, or post-posed, substituting a safe action only on an unsafe proposal, which suffices for a fixed deployed policy. Shield plus environment is again a Markov decision process, so the learner’s convergence guarantees carry over. Synthesis took 0.6 to 2 s on the case studies, a water-tank specification yielding a 602-state game solved in under a second; across grid worlds, a driving loop, an Atari game, and the water tank, no shielded run ever violated the specification, while unshielded learners did during training in the grid worlds and the driving loop, and task performance on the Atari game was unchanged. Two caveats matter. The guarantee is relative to the abstraction and stops binding the real system where reality deviates from

it, and either reward treatment of blocked actions can leave unsafe behavior in, or invisible to, the learned policy, so the shield stays on forever. Shield decisions are loggable, though the paper never develops the log. This is the direct ancestor of the language-model guardrails of Section V.

Across these four systems we read the paradigm as three invariants. The trusted base is small enough to verify exhaustively: a proof checker of a couple of pages, a switching rule simple by design, finite-state monitors and shields. The untrusted component is unrestricted at design time: any proof search, any control law, any learner. And every runtime intervention is defined by an explicit specification with a defined verdict: a safety policy, a recovery region, a temporal-logic property, a safety game. On evidence the four diverge: the proof is durable by construction and the verdict stream by definition, while the switch and the shield decision are events at best. The sections that follow ask what survives when the untrusted component is a language model and no specification of correct behavior exists, beginning with components that lack even a specification of correct perception (Section III).

III. MONITORING LEARNED COMPONENTS

Specification is the first casualty of the move to learned components. Each system of the preceding section bound its untrusted component to an explicit property with a defined verdict; a neural perception module gives that third invariant nothing to bind to, since no formal statement says what a correct reading of a road scene is. What remains checkable is competence, operation inside what the training data established, and the two monitors below deliver it as a per-decision runtime verdict rather than an offline metric.

Cheng et al. [7] build the verdict from Boolean activation patterns. After training, the training set is replayed; each monitored neuron in a near-output layer becomes one bit, set when its ReLU output is positive, and the patterns seen per class are stored in a binary decision diagram. A warning fires when the input’s pattern lies farther than Hamming distance γ from every recorded one, the relaxation costing nothing at query time. The authors name the dilemma: the abstraction “should be abstract enough, but not too abstract”. On MNIST at $\gamma = 0$ the monitor flags 7.66% of test images, of which 10.70% are misclassified; at $\gamma = 2$ it flags 0.6%, of which 31.66% are misclassified. On GTSRB traffic signs, $\gamma = 0$ flags 32.92% of inputs, and a monitor that distrusts a third of its inputs ships nowhere; $\gamma = 3$ flags 4.58%, of which 54.54% are misclassified, against a base error of roughly 3.3% implied by the network’s reported validation accuracy. Those are shares of flagged inputs, not detection rates, and even at the best operating point about half the warnings fall on correct classifications. Because the diagrams top out at hundreds of variables, the larger network watches only 25% of the monitored layer’s 84 neurons, selected by gradient-based sensitivity analysis. A flag says only that the decision is not supported by the training data.

Boxes replace bit-vectors in Henzinger et al. [8], importing the interval domain of classic program analysis. Activation vectors of correctly classified training inputs at layers near

the output are clustered per class by k-means, and each class-cluster pair becomes its minimal per-neuron bounding box. A prediction of class y is accepted if the vector falls inside some box for y ; otherwise the system answers *do not know*. The head-to-head with the Boolean abstraction carries the argument: constructing the Boolean formula exhausted 8 GB of memory on an 84-neuron layer, and on MNIST it took 48 s to build and 17 s to run on a single 40-neuron layer, against 8 s to build from the 60,000 training inputs and 2 s to monitor the 10,000 test inputs for boxes over four layers. The layer counts are unequal, but the extra layers sit on the box side. A conditional guarantee comes with the geometry: once the abstraction has converged, correct predictions are always accepted, so a false warning only signals missing training data; convergence is never established for real data. Novelty is simulated by holding out classes, GTSRB needed a hand-tuned threshold and a uniform 10% box enlargement, and a novelty that lands inside a box is accepted in silence.

Neither monitor judges a claim. The verdict says that this input was processed the way training inputs were, never that this output is true. The check itself is deterministic and auditable, membership in a diagram or in a box, so each flag or abstention is a loggable per-decision intervention and each abstraction a durable, inspectable record of training-time coverage; but the specification enforced is derived from data and holds only statistically. For a language model emitting factual claims and actions, competence monitoring is necessary but nowhere near sufficient, since resembling the training distribution says nothing about whether a generated answer is right. The strongest form left to the paradigm makes the output itself carry a checkable object, the move Section IV develops.

IV. GENERATE-AND-VERIFY AT SCALE

Generate-and-verify restores the paradigm of Section II at full strength. The model proposes, a formal engine certifies every step, and only certified output survives. The model’s error rate stops bearing on correctness and becomes a cost: a weaker proposer means more samples and deeper search; a wrong answer can be proposed but not certified. All three invariants return. The trusted base is the checker alone; the untrusted component may be arbitrarily wrong; the verdict binds one concrete output to a formal specification. The evidence is maximal: a machine-checkable object, re-runnable by any third party, independent of model and vendor.

Two results set the ceiling. In AlphaGeometry, a deterministic symbolic engine exhaustively computes the deduction closure of the premises; when the goal lies beyond it, a 151-million-parameter model proposes exactly one auxiliary construction and the engine re-runs, at most 16 times [9]. Every accepted step is derived by the engine, and the curriculum is the verifier’s own product: 100 million synthetic theorem-proof examples, zero human demonstrations. On IMO-AG-30, thirty olympiad geometry problems, the system solves 25; the previous best method managed 10, the engine with human-designed heuristics 18, GPT-4 writing natural-language proofs 0. The judge, though, is a bespoke engine, not an independent

kernel: correctness is relative to its rules and implementation, and its language covers roughly 75% of IMO geometry problems, excluding inequalities and combinatorial geometry. AlphaProof replaces the bespoke judge with a foundational kernel: a 3-billion-parameter network drives an AlphaZero-style tree search over Lean tactics, with the Lean proof checker as reward during training and judge at inference [10]. At IMO 2024 it solved P1, P2, and P6, the last cracked by only five human contestants; AlphaGeometry 2 added P4, for 28 of 42 points, within the silver-medal range and one point below the gold threshold. The hardest problems needed test-time reinforcement learning, days of training on synthetic variants of the target, worth about 15 absolute percentage points on their formal benchmarks. What survives is a complete Lean proof term, correct assuming the soundness of Lean’s kernel and the declared axioms.

The working-scale instances come from software verification. Baldur samples whole Isabelle/HOL proofs from fine-tuned 8- and 62-billion-parameter models, the proof assistant the sole judge: 47.9% of the 6,336 PISA test theorems proved by the larger model with file context, against 25.6% for Sledgehammer, and 65.7% combined with Thor, 8.7% of theorems beyond Thor’s 57.0% [15]. Repair is the instructive ablation: a second model, fine-tuned on triples of statement, failed attempt, and error message, adds 1.5% of theorems at matched inference cost, but only with the error message; without it, repair does no better than plain sampling. Rejections carry information. Lemur proves the composition itself sound: an untrusted oracle proposes invariants and sub-goals, off-the-shelf verifiers check every proposal, and two soundness theorems rest entirely on the backend verifier; the model never joins the trusted base [16]. With GPT-4 it solves 107 of 133 C loop-invariant benchmarks, and 25 of 47 SV-COMP instances that defeat both classical verifiers within budget, albeit on programs under 150 tokens, a proof of architecture, not scalability. DTV aims the filter at informal reasoning: problem statement and sampled solutions are autoformalized into Isabelle by few-shot prompting, and the answer is the most common among solutions that verify against a statement the model itself wrote; on GSM8K with 62-billion-parameter solution models the filter beats majority voting by more than 12 points, 67.2% to 79.4% [17]. DeepSeek-Prover-V1.5 trains an open-source 7-billion-parameter Lean prover on the proof assistant’s verdict as reward [18]; FunSearch scored model-proposed programs with a trusted evaluator and produced a new best cap-set construction [19].

The assurance stops at two boundaries. Every system above inherited a formal semantics older than itself, a geometry language, Lean, Isabelle, the meaning of a C program; none conjures a specification from prose, and the hardest precondition was settled the moment the domain was chosen. And the informal-to-formal step, wherever it exists, is unverified: AlphaProof’s competition problems were formalized into Lean by human experts, so the silver-range score certifies the formal statements, not their English originals. DTV’s authors supply the example: $1 - 2 + 3$, formalized over the natural numbers, evaluates to 3 under truncated subtraction, so the kernel soundly certifies a valid proof of the wrong statement; the

wrong answer returns one level up, in the statement. No false-positive rate is measured, and the fallback to majority voting is silent, so assurance varies per query, unflagged. Checker soundness does not transfer across that boundary. The costs deserve equal honesty: checking is cheap, but certified output is not. AlphaProof’s main reinforcement-learning run took about 80,000 TPU-days, each hard olympiad problem 2 to 3 days more, and DTV samples 64 solutions per problem before its filter has anything to keep.

Almost nothing deployed looks like this. Most fielded language-model systems have neither a proof kernel nor a pre-existing formal semantics for what they say and do; their obligations are written in prose. What they get instead is guardrails, and Section V examines how much of the paradigm survives that translation.

V. GUARDING LANGUAGE-MODEL SYSTEMS

A guardrail is a policy layer intercepting the inputs, outputs, or actions of a language-model system. It is the shield of Section II rebuilt around an agent instead of a reinforcement learner, and two things changed in the translation while a third did not: the specification language, from temporal logic with defined semantics to prose, prompts, and purpose-built dialects; the checker, from a synthesized automaton to deterministic code interleaved with further model calls; and the evidence, which shielding had already left undeveloped and which remains, in most systems, a log line.

NeMo Guardrails [11] is the industrial baseline: a runtime proxy between user and model enforcing rails written as dialogue flows in Colang, a modeling language mixing English and Python. Each turn runs a three-stage loop: an LLM call maps the utterance to a canonical form with vector-retrieved examples; when that form matches a defined flow the Colang interpreter derives the next step deterministically, otherwise an LLM generates it; a further call writes the reply. Only retrieval, the event loop, and flow matching are deterministic; every perception step is an LLM call, and every shipped safety rail a further LLM judgment, a yes-or-no prompt or a self-consistency check. On their 200-sample balanced set drawn from Anthropic red-teaming and helpfulness data, the moderation rails lift gpt-3.5-turbo from 93% to about 99% of harmful prompts blocked, wrongly refusing 2% of helpful requests; the fact-checking rail reaches about 80% accuracy on MSMARCO-derived entailment pairs; and the three sequential calls come at roughly triple a plain call’s cost and latency. The authors warn that prompt-based rails are not a perfect solution and should not stand alone for safety-specific rails, and a measurement from the RigorLLM paper sharpens the point: under GCG-family adversarial suffix attacks NeMo detects 47% to 66% of harmful prompts [20]. The result is industrially important and formally weak.

AgentSpec [13] returns the specification to a language with defined semantics. A rule couples a trigger event in the agent loop with Boolean predicates over input and trajectory and an enforcement action: stop, route to user inspection, substitute a safe action, or ask the model to re-examine itself. Deterministic code enforces them at three hook points before

an action executes. Hand-written rules intercept over 90% of risky executions on a 750-case code-agent benchmark, cut hazardous actions to zero in all ten unsafe categories of an embodied benchmark while safe-task fulfillment slips from 58.62% to 54.26%, and yield 100% compliance across all eight law-violation scenarios on an autonomous-driving stack, at milliseconds of overhead against agent steps of roughly ten to twenty-five seconds. Checking is per-checkpoint, so a sequence that turns unsafe several steps later passes, and two escape hatches reopen the door the DSL closed: the self-examination action, and LLM-generated rules, whose 70.96% embodied-benchmark recall lets roughly 29% of unsafe cases through. ShieldAgent [12] starts from the policy, compiling platform policy documents offline into per-action probabilistic rule circuits, each rule a linear temporal logic formula over verifiable predicates. Per action, it retrieves the relevant circuits, grounds the predicates with a tool library, generates and runs verification code, and flags the action safe only when the safety margin clears a threshold; each decision emits a binary safety flag, per-rule violation flags, and a textual explanation. On ShieldAgent-Bench, 3,110 trajectories over six web environments with 1,080 annotated rules, it scores 90.4% and 91.7% accuracy on the agent-based and environment-based splits, with false-positive rates of 5.6% and 4.0%, about 10 points above the strongest baselines. The honest reading is that extraction, formulation, grounding, and code generation are all LLM work; only the final inference is formal. Beside them, RigorLLM hardens statistical moderation to 99% to 100% detection under the same GCG-family attacks [20], GuardAgent has one model write deterministically executed guardrail code [21], and AGrail’s cooperating models curate safety checks in a test-time memory [22].

Set against the Introduction’s three losses, the family reads as a climb back toward the paradigm, recovering one loss at a time, and no system recovers all three. NeMo recovers the architecture and nothing else: its policies live in prompts and Colang flows, its verdicts are model prompts, and interventions leave an in-memory trace at most. AgentSpec restores determinism and an enforceable specification, genuine progress; but it reports no per-intervention record, and its self-examination action and generated rules readmit the model to an enforcement path just made deterministic. ShieldAgent restores explicit policy in temporal logic and, in its per-decision triple, the strongest evidence artifact of the family; yet the logic was model-extracted, the grounding model-generated, and the inference probabilistic. GuardAgent’s and AGrail’s guards are models outright. The constant across the climb is a language model somewhere in the trusted base. What Alshiekh’s synthesis derived from a specification in Section II, this generation writes by hand or asks a model to write.

Even where an intervention record exists, nothing consumes it. A ShieldAgent decision leaves a flag, a rule list, and an explanation; there the trail ends: no assurance case cites the record, no certification argument rests on it. A flag with an explanation is raw material for evidence, not yet evidence in an argument. What certification actually consumes, and how runtime records could feed it, occupies Section VI.

VI. FROM TELEMETRY TO CERTIFICATION EVIDENCE

Certification does not consume mechanisms. It consumes evidence structured into an argument, in the safety-case tradition. Everything surveyed so far faces one question: what would a runtime verdict have to look like to serve as evidence in such an argument? The Introduction set the bar: duties attach to deployed applications rather than to the model beneath them [1], and the evidence must carry moral and legal weight in the safety-case tradition [3]. Three papers sketch a loop that could meet the bar; each supplies one arc.

Gauerhof et al. show what the requirements end looks like [23]. Their worked assurance case for a machine-learning pedestrian detector decomposes one vehicle-level requirement, “Ego shall stop at the crossing if a pedestrian is crossing”, into hazard-traceable requirements across five lifecycle stages, the data requirements held to four desiderata: relevant, complete, accurate, balanced. Each is derived, not asserted: identify pedestrians at or near the crossing from 50 m, since comfortable braking on a damp road sheds roughly 20 km/h per second and 60 km/h needs about 50 m; miss an object in at most 1 frame in 5, a stated budget for runtime misses; localize within 50 cm. Assessed against the public JAAD dataset and a CNN trained on it, a majority of the seventeen requirements fail: the dataset has no UK locations, no fog, footage from only five locations, and no failure-derived samples, and the detector’s log-average miss rate is 29.03% for pedestrians larger than 50 pixels and 46.78% for those of 50 to 75 pixels. The explicit requirements are what made those failures visible. The contrast with Section V is exact: a guardrail policy says be harmless; this case says 50 m, 1 in 5, 50 cm, each traceable to a hazard. Verification and deployment, the stages nearest runtime, remain future work; the case builds the requirements end of the loop and stops there.

With Asaadi et al., the argument goes live [24], [25]. Their dynamic assurance case combines a structured argument, a safety architecture, and assurance measures: probabilistic quantifications, computed in operation, of confidence that a system-level property holds. An inner loop assesses risk during operation and triggers contingencies; an outer loop updates the argument between missions. In the group’s accompanying demonstration, a probabilistic monitor runs beside an autonomous-taxiing aircraft’s perception network; posterior confidence drives the contingencies [25]. Their distinction matters: a monitor reports a property of the current state; an assurance measure is predictive, fuses several sources, and quantifies confidence in knowledge of that property. This is the consumer whose absence closed the previous section, machinery turning time-stamped runtime verdicts into updated confidence in a living argument. The confidence is statistical, and the case can be no stronger than the evidence it is fed. Asaadi et al. supply the consuming end; the loop still needs a producer.

The demand side exists as a blueprint. Mökander et al. propose three audit layers for large language models: governance audits of the technology provider, model audits after pre-training but before release, and application audits of deployed applications [2]. Application audits pair functionality

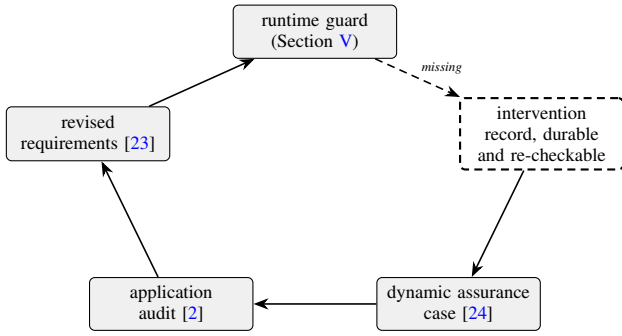


Fig. 1. The evidence loop assembled in this section. The solid elements exist in separate literatures: guards intervene, assurance cases absorb operational evidence, application audits consume logs and return requirements. The loop fails at one joint, drawn dashed: no deployed guard emits a durable intervention record that a third party can re-verify.

audits, against intended and operational goals, with impact audits of effects on users, groups, and the environment. The layers trade artifacts: model audits hand down reports on properties and limitations; ex-post application audits return output logs documenting impact in applied settings, feeding model redesign. One of its seven design claims is that LLM auditing must include elements of continuous ex-post auditing, and such auditing at the application layer needs exactly the durable runtime artifacts guardrails are positioned to emit and currently discard. The blueprint is conceptual, never executed, and presumes a third-party auditor ecosystem that does not yet exist. What it adds is the return arc, from operational logs back to requirements.

On paper, then, the loop closes (Fig. 1). Guards intervene (Section V); interventions could be logged as structured verdicts with the defined semantics of runtime verification (Section II); the verdict trace could feed a dynamic assurance case whose confidence updates in operation [24]; application audits could consume the logs and return revised requirements [2]; and the requirements end already has a worked template [23]. No surveyed system closes this loop. Guardrails, ShieldAgent’s per-decision triple excepted, emit no durable records, and no assurance case cites one. Every conditional above fails at the same joint: the missing artifact is an intervention record a third party can re-verify. Section VII asks what that record must contain, and which verification strength can back each claim inside it.

VII. SYNTHESIS: A SPECTRUM OF VERIFICATION STRENGTH

Four classes order every checker this survey has examined. S1, statistical judgment: an LLM or a classifier decides. S2, rule-checked: deterministic code with defined semantics decides. At S3, solver-checked, a symbolic engine or program verifier discharges the property, and at S4, kernel-checked, a foundational proof kernel accepts the argument or does not. Table I records, for each system surveyed in depth, the checker, what it guarantees, and what a run leaves behind. Read as a whole, the table shows two patterns. Strength correlates with a formal semantics that predated the system: S4 appears only where Lean, Isabelle, or a logical framework had already

defined validity, the precondition Section IV showed was met by choice of domain. Durable evidence, meanwhile, clusters at the top of the table; the systems least in need of a certificate are the ones that produce it.

Read down the class column, the table trades power against generality. The S3 and S4 rows guard mathematics and programs, domains whose objects were born formal; the deployed guardrail family sits at S1 to S2, only ShieldAgent reaching toward S3, and guards open-ended language and action. The DTV row shows why the two ends resist merging. A chain’s effective class is that of its weakest link, the specification step included, so kernel-checking a model-written statement yields S4 per proof and S1 end to end; the same arithmetic applies to every guardrail whose policy a model wrote. The evidence column repeats the ordering: proof-carrying code ships an object a third party can re-check, while the deployed family, ShieldAgent’s triple excepted, keeps no per-intervention record. The realistic frontier is accordingly not certifying everything but moving specific claim and action classes up one level, which AgentSpec’s deterministic checkpoint rules and ShieldAgent’s grounded rule circuits already demonstrate.

The matching below is engineering judgment rather than a theorem. Conversational moderation can live at S1, where a wrong block costs a degraded answer. Agentic tool execution wants S2 or better, deterministic pre-execution checks over auditable rules, because an executed tool call is an event, not an answer (Section I). Physical or safety-relevant actuation wants S3 or better, joined to an engineered fallback in the Simplex tradition. The regulatory obligations surveyed earlier divide the same way. Duties that attach to the deployed application [1] call for intervention records kept per application, at whatever strength the guard already has; what they demand is durability rather than any particular class. Continuous ex-post application audits [2] likewise consume durable operational logs, and a persisted S1 verdict stream already serves an impact audit. Certification in the safety-case tradition [3], [23] asks the most: requirements traceable to hazards, records a third party can re-check, and a stated strength for every claim, because an assurance case inherits the strength class of the evidence feeding it. Here the two axes of the table cross. Durability is orthogonal to strength, and a persisted S2 verdict stream a third party can re-check serves certification better than an S4 proof discarded at runtime; the loop of Section VI needs durability first, maximal strength second.

The agenda we draw from this reading has four problems. Small trusted bases for language systems come first. Guard DSLs whose interpreters and predicate libraries are themselves verified would close the escape hatches that readmit the model, among them AgentSpec’s self-examination action, LLM-authored rules, and unverified predicate code. Evidence-producing guardrails advance the durability axis. Every intervention should emit a durable record with defined verdict semantics, naming the claim or action checked, the policy version, the verdict, and enough context to re-run the check; the producer and checker split of proof-carrying code and the three-valued verdicts of runtime verification supply the templates, and dynamic assurance cases are the consumer. What can be checked at all is the third problem. A monitorability

TABLE I

VERIFICATION STRENGTH AND RUNTIME EVIDENCE ACROSS THE SURVEYED SYSTEMS. CLASSES: S1 STATISTICAL JUDGMENT, S2 RULE-CHECKED, S3 SOLVER-CHECKED, S4 KERNEL-CHECKED. SPLIT ENTRIES SEPARATE THE STRENGTH OF THE RUNTIME CHECK FROM THAT OF THE SPECIFICATION IT ENFORCES; RANGES SPAN A SYSTEM'S COMPONENTS.

System	Checker	What is guaranteed	Runtime evidence	Class
Proof-carrying code [4]	small proof checker (type checking)	compliance with the stated policy	the shipped proof, third-party re-checkable	S4
Simplex [5]	verified baseline controller, simple switching rule	recovery within an offline-proven envelope	switch events only	S2
Runtime verification [14]	synthesized finite-state monitor	the observed run satisfies the property	three-valued verdict stream	S2
Shielding [6]	game-synthesized reactive shield	specification compliance relative to the abstraction	loggable decisions, log undeveloped	S2
NN runtime monitors [7], [8]	BDD membership; box containment	input processed like training data	per-decision flag or abstention	S2 check, S1 spec
AlphaGeometry [9]	bespoke symbolic engine	every step derived by the engine	full deduction trace	S3
AlphaProof [10]; Baldur [15]	Lean or Isabelle kernel	proof valid modulo kernel and axioms	kernel-checkable proof object	S4
Lemur [16]	program verifiers, proved-sound composition	property holds when the calculus succeeds	verifier-discharged proposals	S3
DTV [17]	Isabelle, over a model-written statement	the formalization verifies	proof sketch; the statement itself unverified	S4 per proof, S1 end to end
NeMo Guardrails [11]	LLM prompts around a flow interpreter	none formal	in-memory trace	S1–S2
AgentSpec [13]	deterministic trigger-predicate rules	rule compliance at each checkpoint	fired rule; no record reported	S2
ShieldAgent [12]	probabilistic rule circuits, LLM-grounded predicates	safety margin clears a threshold	flag, violated rules, explanation	S2–S3

theory for language-model outputs would state which claim and action classes admit deterministic oracles and at what cost; today that boundary is drawn by engineering accident rather than analysis. The fourth problem spans both axes: assurance over versioned world models, where the oracle is not a fixed formal theory but a large, evolving factual dataset such as a map, a knowledge graph, or a product catalog. A verdict there is meaningful only relative to the data version it was checked against, so records must state that version. The spatial instance of this problem, with a certificate format binding assertions to map releases, has been mapped [26]; the general conditions under which a verdict survives a version change are, to our knowledge, undefined. Our ongoing work continues there.

This synthesis has blind spots of its own. One author screened the corpus, from an industrial vantage point, and both the selection and the agenda carry that slant. The reference budget forced exclusions: the training-time alignment literature, offline formal verification of neural networks, the runtime-verification literature beyond its canonical survey, and the fine-tuned moderation models of the LlamaGuard family all fall outside scope. And the guardrail literature is young; the caveat of Section I applies doubly to the placements in Table I.

VIII. CONCLUSION

One architecture spans the three decades traced here, from proof kernels to prompt filters: a capable component that is not trusted, a small component that is, and a boundary where every intervention is meant to leave evidence. It guarded machine code, then control loops, then learned policies; it now guards prose. The gain was generality: the guarded component converses and acts in an open world no formal

theory covers. The loss was the classical guarantee: the trusted base is no longer small, the specification no longer formal, and interventions leave nothing a third party can check. The architecture arrived intact; the assurance did not.

The repair requires no new theory. Enforcement already exists, in shields synthesized from specifications and in deterministic guardrail languages. Evidence semantics already exist, in runtime verification's three-valued verdicts and in the split of certificate producer from checker. Assurance cases were built to absorb operational evidence, and audits now attach duties to the deployed application itself. Each literature solved its part in its own vocabulary, and the parts have not met. What they lack is the artifact that would connect them: a durable record of every runtime intervention whose validity does not rest on the good behavior of the system it describes. Building it means connecting parts that already exist; the agenda of Section VII names the steps.

For systems grounded in an evolving model of the world, a map or a knowledge graph, the record needs one more field. A verdict there holds relative to a stated version of the ground truth, and whether it survives the next version is a question no one has yet been obliged to define. The version-aware form of that record, proposed for road networks in [26], is the subject of our ongoing research. Called once more to account, a deployed language-model system offers not a log but a record: specified before it was needed, written at the moment of intervention, and checkable by parties who never had to trust it.

REFERENCES

- [1] P. Hacker, A. Engel, and M. Mauer, "Regulating ChatGPT and other large generative AI models," in *Proceedings of the 2023 ACM*

- Conference on Fairness, Accountability, and Transparency (FAcT '23)*. Chicago, IL, USA: ACM, 2023, pp. 1112–1123. [Online]. Available: <https://doi.org/10.1145/3593013.3594067>
- [2] J. Mökander, J. Schuett, H. R. Kirk, and L. Floridi, “Auditing large language models: a three-layered approach,” *AI and Ethics*, vol. 4, no. 4, pp. 1085–1115, 2024. [Online]. Available: <https://doi.org/10.1007/s43681-023-00289-2>
 - [3] S. Burton, I. Habli, T. Lawton, J. McDermid, P. Morgan, and Z. Porter, “Mind the gaps: Assuring the safety of autonomous systems from an engineering, ethical, and legal perspective,” *Artificial Intelligence*, vol. 279, p. 103201, 2020. [Online]. Available: <https://doi.org/10.1016/j.artint.2019.103201>
 - [4] G. C. Necula, “Proof-carrying code,” in *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 1997, pp. 106–119. [Online]. Available: <https://doi.org/10.1145/263699.263712>
 - [5] L. Sha, “Using simplicity to control complexity,” *IEEE Software*, vol. 18, no. 4, pp. 20–28, Jul. 2001. [Online]. Available: <https://doi.org/10.1109/MS.2001.936213>
 - [6] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, “Safe reinforcement learning via shielding,” in *Proc. 32nd AAAI Conf. on Artificial Intelligence*, vol. 32, 2018. [Online]. Available: <https://doi.org/10.1609/aaai.v32i1.11797>
 - [7] C.-H. Cheng, G. Nührenberg, and H. Yasuoka, “Runtime monitoring neuron activation patterns,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Florence, Italy: IEEE, 2019, pp. 300–303. [Online]. Available: <https://doi.org/10.23919/DATE.2019.8714971>
 - [8] T. A. Henzinger, A. Lukina, and C. Schilling, “Outside the box: Abstraction-based monitoring of neural networks,” in *ECAI 2020 — 24th European Conference on Artificial Intelligence*, ser. Frontiers in Artificial Intelligence and Applications, vol. 325. IOS Press, 2020, pp. 2433–2440. [Online]. Available: <https://doi.org/10.3233/FAIA200375>
 - [9] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, “Solving olympiad geometry without human demonstrations,” *Nature*, vol. 625, no. 7995, pp. 476–482, 2024. [Online]. Available: <https://doi.org/10.1038/s41586-023-06747-5>
 - [10] T. Hubert, R. Mehta, L. Sartran *et al.*, “Olympiad-level formal mathematical reasoning with reinforcement learning,” *Nature*, vol. 651, no. 8106, pp. 607–613, 2026, published online 12 November 2025. [Online]. Available: <https://doi.org/10.1038/s41586-025-09833-y>
 - [11] T. Rebedea, R. Dinu, M. N. Sreedhar, C. Parisien, and J. Cohen, “NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Y. Feng and E. Lefever, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 431–445. [Online]. Available: <https://doi.org/10.18653/v1/2023.emnlp-demo.40>
 - [12] Z. Chen, M. Kang, and B. Li, “ShieldAgent: Shielding agents via verifiable safety policy reasoning,” in *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, vol. 267. PMLR, 2025, pp. 8313–8344, arXiv:2503.22738. [Online]. Available: <https://proceedings.mlr.press/v267/chen25ae.html>
 - [13] H. Wang, C. M. Poskitt, and J. Sun, “AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents,” in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. Rio de Janeiro, Brazil: ACM, 2026, arXiv:2503.18666. [Online]. Available: <https://doi.org/10.48550/arXiv.2503.18666>
 - [14] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009. [Online]. Available: <https://doi.org/10.1016/j.jlap.2008.08.004>
 - [15] E. First, M. N. Rabe, T. Ringer, and Y. Brun, “Baldur: Whole-proof generation and repair with large language models,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. San Francisco, CA, USA: ACM, 2023, pp. 1229–1241. [Online]. Available: <https://doi.org/10.1145/3611643.3616243>
 - [16] H. Wu, C. Barrett, and N. Narodytska, “Lemur: Integrating large language models in automated program verification,” in *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=Q3YaCghZNt>
 - [17] J. P. Zhou, C. Staats, W. Li, C. Szegedy, K. Q. Weinberger, and Y. Wu, “Don’t trust: Verify – grounding LLM quantitative reasoning with autoformalization,” in *The Twelfth International Conference on Learning Representations (ICLR 2024)*. Vienna, Austria: OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=V5tdi14ple>
 - [18] H. Xin, Z. Z. Ren, J. Song *et al.*, “DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search,” in *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025, arXiv:2408.08152. [Online]. Available: <https://openreview.net/forum?id=14YAIwrsXa>
 - [19] B. Romera-Paredes, M. Barekatin, A. Novikov *et al.*, “Mathematical discoveries from program search with large language models,” *Nature*, vol. 625, no. 7995, pp. 468–475, 2024. [Online]. Available: <https://doi.org/10.1038/s41586-023-06924-6>
 - [20] Z. Yuan, Z. Xiong, Y. Zeng, N. Yu, R. Jia, D. Song, and B. Li, “RigorLLM: Resilient guardrails for large language models against undesired content,” in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, vol. 235. PMLR, 2024, pp. 57953–57965. [Online]. Available: <https://proceedings.mlr.press/v235/yuan24f.html>
 - [21] Z. Xiang, L. Zheng, Y. Li, J. Hong, Q. Li, H. Xie, J. Zhang, Z. Xiong, C. Xie, C. Yang, D. Song, and B. Li, “GuardAgent: Safeguard LLM agents via knowledge-enabled reasoning,” in *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, vol. 267. PMLR, 2025, pp. 68316–68342. [Online]. Available: <https://proceedings.mlr.press/v267/xiang25a.html>
 - [22] W. Luo, S. Dai, X. Liu, S. Banerjee, H. Sun, M. Chen, and C. Xiao, “AGrail: A lifelong agent guardrail with effective and adaptive safety detection,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vienna, Austria: Association for Computational Linguistics, 2025, pp. 8104–8139. [Online]. Available: <https://doi.org/10.18653/v1/2025.acl-long.399>
 - [23] L. Gauerhof, R. Hawkins, C. Picardi, C. Paterson, Y. Hagiwara, and I. Habli, “Assuring the safety of machine learning for pedestrian detection at crossings,” in *Computer Safety, Reliability, and Security (SAFECOMP 2020)*, ser. Lecture Notes in Computer Science, A. Casimiro, F. Ortmeier, F. Bitsch, and P. Ferreira, Eds., vol. 12234. Cham: Springer, 2020, pp. 197–212. [Online]. Available: https://doi.org/10.1007/978-3-030-54549-9_13
 - [24] E. Asaadi, E. Denney, J. Menzies, G. J. Pai, and D. Petroff, “Dynamic assurance cases: A pathway to trusted autonomy,” *Computer*, vol. 53, no. 12, pp. 35–46, Dec. 2020. [Online]. Available: <https://doi.org/10.1109/MC.2020.3022030>
 - [25] E. Asaadi, E. Denney, and G. J. Pai, “Quantifying assurance in learning-enabled systems,” in *Computer Safety, Reliability, and Security (SAFECOMP 2020)*, ser. Lecture Notes in Computer Science, vol. 12234. Springer, 2020, pp. 270–286. [Online]. Available: https://doi.org/10.1007/978-3-030-54549-9_18
 - [26] K. Essam and R. S. Romaniuk, “Verifiable reasoning of large language models over structured spatial data: A survey and research agenda,” *International Journal of Electronics and Telecommunications*, 2026, in press. [Online]. Available: <https://doi.org/10.24425/ijet.2026.6314>