VERIFIABILITY-FIRST AI ENGINEERING IN THE ERA OF AIWARE: A CONCEPTUAL FRAMEWORK, DESIGN PRINCIPLES, AND ARCHITECTURAL PATTERNS FOR SCALABLE VERIFICATION

Liming Zhu, Qinghua Lu
Data61, CSIRO
Sydney, Australia
{liming.zhu, qinghua.lu}@data61.csiro.au

ABSTRACT

Foundation models and agentic AI systems fundamentally challenge traditional assumptions of software engineering, where system behaviour is explicitly encoded in code and verified through testing and analysis over software artefacts. As AI systems increasingly generate behaviour autonomously, the primary engineering bottleneck shifts from producing behaviour to verifying it at scale. This paper introduces verifiability-first AI engineering as a foundational paradigm for the era of AIware, in which system logic is increasingly embedded in model weights, learned representations, and agentic policies rather than explicit code. We argue that AI engineering requires treating verifiability as a first-class design objective. We present a conceptual framework that characterises verification along five dimensions: what is verified, how verification is performed, when it occurs across the lifecycle, who performs it, and why it is required. Building on this framework, we derive a set of design principles that guide the design of scalable verification, and a collection of architectural design patterns that operationalise these principles as reusable solutions for engineering verifiability-first AI systems in the era of AIware.

***K*eywords** AI · AI Engineering · AIware · Verification · Verifiability · Design Principle · Pattern

1 Introduction

Software engineering has traditionally assumed that business logic is explicitly encoded in human-written code, enabling verification through testing and analysis over code artefacts. The emergence of foundation models and agentic AI systems fundamentally challenges this assumption and reshapes software engineering for AI systems, that is, AI engineering. AI engineering is the application of software engineering principles and techniques to the design, development, and operation of AI systems [1]. As AI systems increasingly rely on learned models and autonomous behaviours rather than explicitly programmed logic, many long-standing assumptions about how system behaviour is specified, constructed, and verified no longer hold.

This shift gives rise to two emerging paradigms. The first paradigm is *AI-assisted software engineering*, in which AI systems increasingly write, improve, and test software code autonomously, turning implementation and low-level verification into cheap, automated activities. The second and more profound paradigm is *AIware engineering*, where logic is embedded in model weights, learned representations, and agentic behaviours rather than explicit code. Foundation models act as new software components, adapt to new tasks through in-context learning, and dynamically generate actions or tool invocations without direct developer involvement.

For both paradigms, the core engineering bottleneck shifts from producing behaviour to verifying behaviour. While AI has dramatically reduced the cost of generating candidate solutions, verification does not scale automatically in the same way. The key challenge is how to ensure that verification is automated, scalable, and reliable enough to match the pace of AI-driven behaviour generation. In this paper, we use the term *verification* broadly to encompass both

traditional verification and validation encompassing all activities that assess the acceptability of system behaviour with respect to specification, requirements, and intended use and goals.

AI engineering can be viewed as comprising two distinct classes of activities: *solve* tasks and *verify* tasks [2, 3, 4]. Solve tasks are concerned with producing system behaviour, including designing system architectures, training models, generating code, or producing decisions and actions. Verify tasks are concerned with assessing and interpreting that behaviour with respect to goals and specifications. Conceptually, solving and verification have long been distinguished in software engineering, with humans typically responsible for both, supported by specialised tools. What has changed is that solving is becoming increasingly automated, cheap and scalable through AI, while verification risks becoming the dominant bottleneck if it remains manual, or bespoke.

Some tasks naturally admit cheap and reliable verification. In these cases, producing a correct solution may be difficult, while verifying a candidate solution is comparatively straightforward using automated tools such as simulations, formal checkers, or mathematical procedures. A canonical example is integer factorisation: finding the factors of a large integer may be computationally expensive, while verifying a proposed factorisation is trivial via multiplication. Such tasks exhibit a strong form of *solve–verify asymmetry*, allowing verification to scale efficiently alongside automated solution generation.

However, many tasks lack such asymmetry. In mathematics, even when a problem has a determinate truth, designing a verifier is often as hard as solving the problem. Breaking a problem into verifiable steps is itself a difficult creative act, and in many cases we simply do not know how to do it. The problem itself or the problem-decomposition problem do not really exhibit solve–verify asymmetry at all. Physics makes this limitation even clearer. While fundamental laws or equations may exist, verification often requires experimental apparatus that is prohibitively expensive or not yet known how to construct. This is essentially the core challenge of modern theoretical and experimental physics. Entire classes of theories cannot be verified because the required experiments are infeasible. So epistemic clarity does not imply easy verification.

The challenge is even more pronounced for problems that do not admit a determinate truth at all. In domains shaped by human preferences, plural values, or evolving social and institutional contexts, there may be no stable ground truth to verify against. The primary scalability bottleneck here is often not the verifier itself, whether human or automated, but the generation of the reference data being verified against. In practice, this challenge is addressed through mechanisms such as synthetic data, game-based evaluation, and LLM-as-judge. These approaches do not eliminate verification difficulty, but shift it toward the controlled construction of proxy criteria that are cheap to check, stable under refinement, and reusable at scale.

We propose *verifiability-first AI engineering* as a new paradigm for AI engineering, which treats verifiability as a first-class design objective rather than a downstream evaluation activity. Under this paradigm, problems are deliberately formulated and decomposed to expose verification structure, so that verificaation can be automated, scalable, and compositional even as behaviour generation becomes cheap and abundant. This involves choosing intermediate representations, behavioural claims, and workflows. Where strong solve–verify asymmetry exists, verifiability-first engineering aims to exploit it; where it does not, it seeks alternative formulations that yield partial, incremental, or proxy forms of verification that still scale.

Treating verification as first-class does not mean adding more checks to existing development lifecycles or replaying traditional artefacts with additional validation steps. Verification has always been central to software engineering through testing, property-based testing, and formal methods. What changes in AI systems is that verification must actively shape problem formulation, decomposition, and system structure from the outset, rather than being applied to fixed artefacts after the fact. The key distinction is not between artefacts and properties, but whether the chosen behavioural claims, representations, or abstractions can be verified reliably and at scale. When they cannot, verifiability-first engineering calls for redefining those targets rather than defaulting to conventional decompositions.

Problem decomposition is therefore a core mechanism of this paradigm. Decomposition does not guarantee complete verifiability, since some subproblems may resist automated checking. The strength of verifiability-first engineering lies instead in its ability to accumulate verified structure wherever possible and to compose partial verification results over time. Even when full end-to-end verification is infeasible, such stackable and incremental verification enables systems to evolve by extending verified components rather than repeatedly revisiting settled claims, substantially improving robustness, efficiency, and trustworthiness in adaptive and agentic systems.

In this paper, we make three contributions. First, we introduce a conceptual framework for verifiability-first AI engineering that characterises verification along five dimensions: what is verified, how verification is carried out, when it occurs across the lifecycle, who performs it, and why it is required. Second, we derive a set of design principles that explain how deliberate problem formulation and decomposition can enable automated, scalable verification. Third, we present a collection of architectural design patterns that operationalise these principles in practice, supporting scalable

Table 1: Mapping from framework dimensions to verifiability-first design principles and architectural patterns

Framework Dimensions	Design Principles	Architectural Patterns
What	Principle 1: Decompose Problems Around Verifiable Behavioural Claims	Atomic Verifiable Claims; Verification-Driven Workflow Decomposition; Evidence-Augmented Generation
How	Principle 2: Explicitly Separate Solve and Verify Tasks and Exploit Solve–Verify Asymmetry Where Available	Solver-Verifier Separation; Sandbox Gatekeeper; Evidence-Augmented Generation
When	Principle 3: Design Verification Across the Lifecycle and Across Levels	Stackable Verification; Context-Epistemic Verification Routing
Why	Principle 4: Design for Stackable and Incremental Verification with Scalable and Verifiable Rewards	Stackable Verification; Scalable and Verifiable Rewards; Atomic Verifiable Claims
Who	Principle 5: Allocate Verification Actors Explicitly and Preserve Human Understandability	Multi-verifiers; Context-Epistemic Verification Routing; Evidence-Augmented Generation

verification in AI engineering. Table 1 provides a roadmap of the paper by mapping the five framework dimensions to the corresponding design principles and architectural patterns. This table summarises how conceptual concerns translate into concrete design guidance and reusable architectural patterns.

The remainder of the paper is organised as follows. Section 2 introduces the methodology. Section 3 presents the conceptual framework for verifiability-first AI engineering. Section 4 summarises the design principles. Section 5 identifies the corresponding architectural design patterns. Section 6 discusses related work, and Section 7 concludes the paper and outlines future research.

2 Methodology

To develop a conceptual framework for verifiability-first AI engineering, together with corresponding design principles and architectural design patterns, we conducted a systematic literature review (SLR). The objective of the SLR is to identify how verification is conceptualised, structured, and operationalised in contemporary AI systems. We adopted Kitchenham's SLR guideline [5] to ensure rigour and reproducibility in the review process. The complete SLR protocol, including search strategy, inclusion and exclusion criteria, and extraction rules, is available as online supplementary material[1]. Figure 1 illustrates the overall methodology, including paper identification, screening, extraction, and analysis.

2.1 Research questions

The research questions are structured to progressively construct a verifiability-first view of AI engineering. We begin by examining how verification is characterised and instantiated in existing AI systems, then abstract recurring design principles that enable scalable verification, and finally identify architectural design patterns that operationalise these principles in practice.

RQ1: Verification characteristics. How is verification conceptualised and applied in AI systems?

- **RQ1-1:** What is verified?
- **RQ1-2:** How is verification performed?
  - **RQ1-2-1:** What type of problem is being verified, in terms of epistemic clarity and contextual dependence?
  - **RQ1-2-2:** How is the problem decomposed to make verification feasible or scalable?
  - **RQ1-2-3:** At which level is verification performed (case-level or system-level)?
- **RQ1-3:** When does verification occur across the system lifecycle (pre-deployment or post-deployment)?
- **RQ1-4:** Who performs verification (e.g., humans, AI tools, or other tools)?
- **RQ1-5:** Why is verification required (e.g., correctness or understandability)?

[1] `https://drive.google.com/drive/folders/1HoPJPb7CSPFKFwuLBQ-Mabdo7eMbzn7A?usp=share_link`

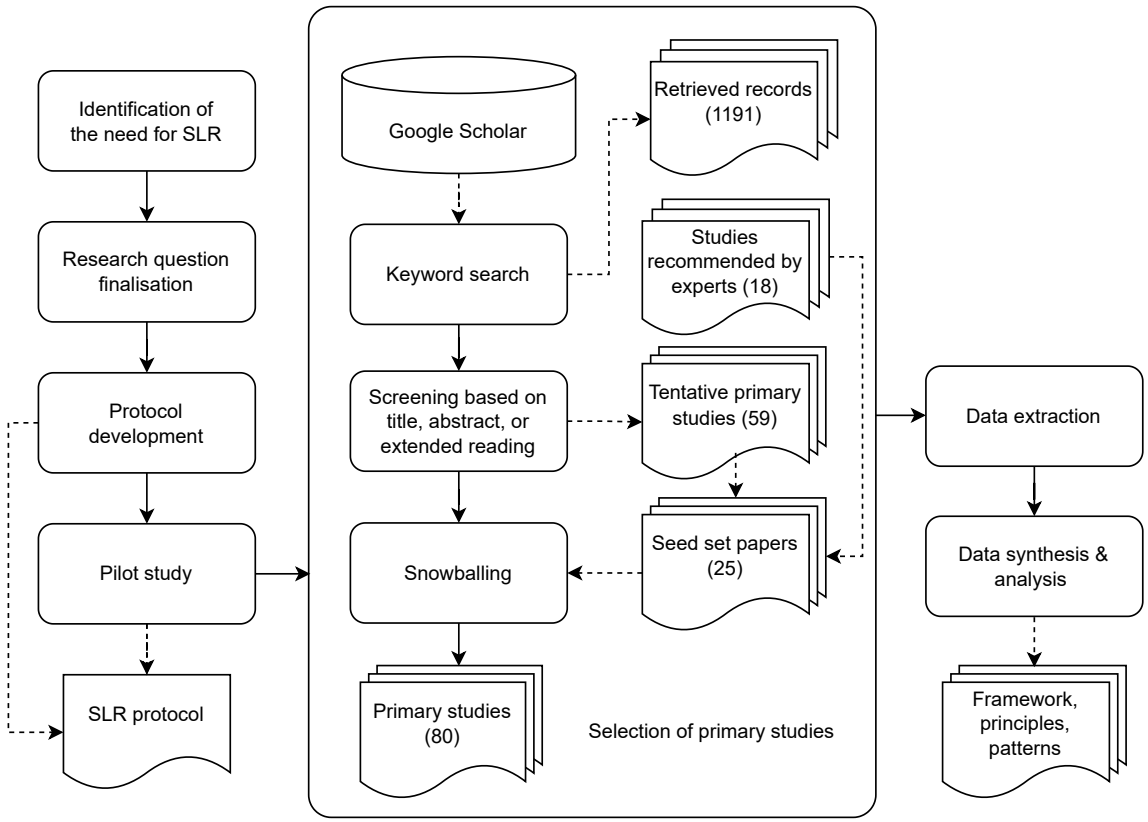

Figure 1: Methodology.

RQ2: Design principles. What design principles are proposed or implied in the study that explains how AI systems can be structured so that verification is cheap, scalable, and composable?

RQ3: Architectural design patterns. What architectural patterns are reported that operationalise these principles and enable verifiability-first AI engineering in practice?

2.2 Search strategy

We derived the search strings by systematically identifying keywords from the research questions. To refine the search strategy, we conducted a pilot study in which alternative keyword combinations were tested and their results compared to assess coverage and relevance. We used "AI" and "Verification" as the key terms and included synonyms and abbreviations as supplementary terms to increase the search results. We designed the search strings for each primary source to check the title. After completing the first draft of search strings, we examined the results of each search string against Google Scholar to check the effectiveness of the search strings. The finalised search terms are shown in Table 2. The search strings and the respective paper quantities of the initial search for each primary source are listed in our SLR protocol. We applied the final search strings using Google Scholar, as it provides broad coverage of major academic digital libraries (including ACM Digital Library, IEEE Xplore, ScienceDirect, and SpringerLink) as well as relevant industry literature. The search period spans from 2021 (when the concept of foundation models was first introduced [6] to 31 December 2025.

Table 2: Key and supplementary search terms

Key terms	Supportive terms
AI	Artificial Intelligence, Foundation Model, Large Language Model, LLM, agent, agentic
Verification	Verifiability, verified, verifiable

We screened the initial search results using predefined inclusion and exclusion criteria. The inclusion criteria were:

- papers or articles that propose a solution, method, or tool for verification of AI systems; and
- papers or articles published as peer-reviewed scientific papers or industry articles.

The exclusion criteria were:

- papers that focus exclusively on traditional machine learning systems without addressing foundation models or agentic AI;
- papers not written in English;
- conference papers that have an extended journal version; and
- dissertations, tutorials, editorials, books, technical reports, and white papers.

In addition to database searching, we employed backward and forward snowballing. We identified a seed set of highly relevant papers based on citation counts (>20 citations) and recommendations from experts. References cited by these papers and subsequent papers citing them were examined to identify additional relevant studies. Following screening and snowballing, we identified a total of 80 primary studies for inclusion in the SLR.

2.3 Data extraction, synthesis, and analysis

We derived the framework, design principles, and architectural patterns through a structured synthesis of the selected studies, guided by the predefined research questions. Based on the extracted answers, we synthesised a conceptual framework for verifiability-first AI engineering, together with a set of design principles and architectural design patterns.

All extracted data were stored in a spreadsheet to support structured analysis and cross-paper comparison. This representation allowed us to identify recurring verification concerns, common abstractions, and repeated solutions.

From this analysis, we synthesised a conceptual framework for verifiability-first AI engineering. The framework captures five orthogonal dimensions that consistently shaped verification design decisions across the literature: what is verified, how verification is performed, when it occurs across the system lifecycle, who performs verification, and why verification is required. These dimensions are descriptive rather than prescriptive and provide a unifying lens for analysing verification practices in AI systems.

Design principles were then derived by abstracting normative guidance from this framework. Each principle articulates how systems should be designed to support scalable and reliable verification along one or more framework dimensions. The principles are not independent heuristics; rather, they summarise recurring structural commitments observed across the literature and explain how verification considerations should shape problem formulation, decomposition, and system architecture.

Architectural design patterns were identified as concrete, reusable structures that operationalise one or more design principles. Candidate patterns were identified by clustering recurring architectural solutions observed in the extracted data. Each pattern was documented following a standard pattern description structure, including context, problem, solution, benefits, drawbacks, and known uses. Known uses were drawn primarily from the analysed studies and supplemented, where necessary, by targeted manual searches to identify real-world deployments that instantiate the same architectural structure.

This layered synthesis, from empirical extraction, to conceptual framework, to normative principles, and finally to operational architectural patterns, ensures traceability between observed practice and the prescriptive guidance presented in this paper. Design principles provide the conceptual bridge between framework dimensions and architectural patterns, while patterns demonstrate how the principles can be realised in practice.

3 A Conceptual Framework for Verifiability-First AI Engineering

In this section, we introduce a conceptual framework for verifiability-first AI engineering structured along five complementary dimensions: *what* is verified, *how* verification is performed, *when* verification occurs, *who* performs verification, and *why* verification is required.

3.1 What is Verified

In verifiability-first AI engineering, verification targets are not implicitly inherited from conventional software artefacts or lifecycle stages. Instead, they are explicitly constructed as behavioural claims that a system makes about its behaviour, outputs, or operation. These claims may be realised through artefacts such as execution traces, models, logs,

or certificates, but the artefacts themselves are not the objects of verification. Rather, they function as representations through which specific claims can be evaluated. Verification is therefore claim-centric rather than artefact-centric, with artefacts introduced only insofar as they render claims amenable to reliable and scalable checking.

At one end of this spectrum, verification targets may be concrete artefacts produced by AI systems, such as code, structured text, or formal specifications. In these cases, verification assesses explicit properties including functional correctness, syntactic validity, consistency, or satisfaction of formal constraints. This mode of verification is most effective when artefacts admit precise semantics and machine-checkable specifications, as in software and hardware generation [7, 8, 9] or natural-language-to-structure transformations [10, 11]. Importantly, the artefact itself is only a suitable verification target to the extent that the underlying properties are clearly defined and mechanisable.

Beyond final artefacts, verification targets may instead be defined over *reasoning processes and intermediate states*. In this setting, the object of verification is not the final answer but the structure of the reasoning that leads to it. Properties such as logical soundness, completeness, faithfulness to evidence, or absence of critical gaps become the primary verification targets [12, 13]. This shift typically occurs when end-to-end correctness is difficult or ambiguous to establish, and intermediate claims provide a more tractable and informative basis for verification.

For agentic AI systems, verification targets frequently extend to *behaviour over time*. Rather than static artefacts, the object of verification becomes an execution trajectory: a sequence of actions, tool invocations, or interactions with an environment. Verification in this regime checks whether individual actions are admissible, whether trajectories satisfy safety or policy constraints, and whether control remains within acceptable bounds [14, 15, 16, 17]. Here, verification focuses on properties such as constraint satisfaction, controllability, and prevention of harmful escalation, reflecting the temporal and interactive nature of the system.

A closely related class of targets arises in tool use, workflows, and multi-step task execution. In these cases, correctness is not attributable to any single output but to the coordination, ordering, and completion of a sequence of steps. Verification therefore targets tool-call sequences, intermediate results, and aggregate task outcomes, often using execution traces or transactional logs as the verifiable representation. Such targets enable replay, auditability, and partial or step-wise verification even when holistic correctness cannot be conclusively determined [18, 19, 20].

In multi-agent and distributed systems, verification targets shift further toward *interaction-level and trust-related properties*. Here, the focus is on agent identity, authenticity, message integrity, provenance, and compliance with shared security or privacy policies [21, 22, 23, 24]. These targets reflect the fact that correctness and safety are emergent properties of interaction patterns across agents and infrastructure, rather than attributes of any single component.

In safety-critical and scientific domains, verification targets are often defined at the *system level*. Rather than validating individual outputs, verification assesses global properties such as safety envelopes, robustness to perturbations, adherence to modelling assumptions, or consistency with physical or biological laws [25, 26, 27, 28, 29, 30]. In these settings, verification rarely reduces to simple correctness checks and instead evaluates whether system behaviour remains acceptable under uncertainty and variation.

Finally, verification targets may be defined at a *meta or infrastructural level*, covering claims about how AI systems are developed, trained, and governed. These include training correctness [31], data quality [32], provenance [33], and frontier-scale governance assurances [34]. Here, verification supports accountability, auditability, and trust in processes rather than validation of individual inference outcomes.

This dimension highlights that verification in AI engineering is neither purely artefact-centric nor exclusively behavioural. Verification targets range from static artefacts to dynamic behaviours, interactions, system-level properties, and procedural claims. The central design question is therefore not whether to verify artefacts or behaviour, but which *claims* must remain verifiable, through which representations, and at what level of granularity. When direct verification is infeasible, systems can instead redefine the verification target itself by introducing alternative claims, intermediate representations, or structured traces that reduce verification burden, even if this shifts complexity into problem formulation or decomposition.

3.2 How Verification is Performed

In verifiability-first AI engineering, verification is performed by analysing and exploiting the *verification structure* of a problem, rather than by decomposing systems into conventional workflows and attaching verification tasks to final artefacts after generation. The emphasis shifts from verifying products of computation to shaping how computation itself unfolds so that its critical properties become observable, checkable, and reusable.

Verification is therefore not treated as a single post hoc check, but as a set of structured mechanisms embedded at different stages of system execution and at different levels of abstraction. These mechanisms differ in how they compose,

what reference standards they rely on, and how verification effort scales relative to generation. Effective verification therefore depends less on the power of individual checkers than on aligning verification mechanisms with the structure of the task, the nature of the claims being verified, and the desired mode of compositional improvement.

One prominent class of approaches performs verification through *runtime monitoring* of agent actions or model outputs. Here, verification operates as an observer that evaluates actions, tool invocations, or outputs against explicit policies, constraints, or admissibility rules prior to execution or commitment [14, 35, 16]. Rather than attempting to establish end-to-end correctness, runtime verification focuses on bounding behaviour within acceptable regions, preventing unsafe escalation, and enforcing local constraints. Verification scales in this setting because checks are local, cheap, and applied selectively to high-risk decisions.

A second major class of approaches performs verification by integrating *formal methods and programmatic checking* directly into the generation loop [7, 36, 9, 37]. In these systems, candidate outputs are accepted only if they satisfy formally specified properties such as logical constraints, invariants, or preconditions. Verification is exact and automated, and strong solve–verify asymmetry can be exploited when checking is substantially cheaper than generation. This approach scales when reference standards such as specifications or contracts are stable and machine-checkable, but becomes brittle when specifications are incomplete or costly to construct.

When correctness cannot be fully formalised, verification is often enabled by *structuring outputs into intermediate artefacts* such as reasoning steps, claims, predicates, or partial plans [38, 12, 39]. Verification then targets these intermediate claims rather than final outcomes. This enables temporal or stepwise composition, where earlier verification results constrain later behaviour and reduce downstream verification burden. Verification scales here by accumulation: verified structure persists across steps unless assumptions change.

Verification is also performed through *cross-checking and multi-agent verification*, where correctness emerges from redundancy rather than reliance on a single authoritative checker [40, 41, 42]. Multiple generators or verifiers independently assess the same claim, and inconsistencies become actionable signals. This mode is particularly effective when verifiers are themselves learned or approximate, and scales by reducing correlated failure rather than by proving absolute correctness.

In distributed, adversarial, or privacy-sensitive settings, verification is frequently performed using *cryptographic and ledger-based mechanisms* that provide strong guarantees without requiring re-execution [22, 43, 44, 45]. Here, verification targets integrity, provenance, and compliance rather than behavioural correctness. Evidence such as signatures, certificates, or logs can be checked cheaply and reused by downstream verifiers, enabling evidential composition. Verification scales because evidence persists and does not require re-execution of the original computation.

Across scientific and safety-critical domains, verification is often grounded in *external trusted processes* rather than internal model confidence [28, 25, 46]. In these contexts, verification assesses consistency with physical laws, robustness to perturbations, or adherence to safety envelopes defined by simulators, experiments, or control-theoretic models, rather than correctness of individual outputs alone.

Finally, a growing body of work performs verification at the *process and governance level*, through mechanisms such as human-led audits, cryptographic audit trails, hardware attestation, and institutional oversight [34], or through modular audit pipelines and continuous assessment agents [47]. Here, verification supports accountability, traceability, and trust in system development and deployment processes rather than individual inference outcomes.

In summary, verification in AI systems is realised through a diverse but converging set of mechanisms, including runtime monitoring, formal checking, structured decomposition, cross-agent redundancy, cryptographic attestation, grounding in external processes, and procedural oversight. Verification is most effective when treated as a first-class design constraint that shapes how problems are structured, how systems interact with their environment, and which claims are exposed for checking, rather than as an afterthought applied to final outputs.

3.2.1 What Type of Problem is Verified

How verification is performed depends heavily on the type of the problem: whether correctness is well-defined, how it varies with context, and how uncertainty arises.

Across AI systems, verification mechanisms consistently align with the *epistemic clarity* and *context dependence* of the underlying task, rather than with any particular application domain or system architecture. Tasks with determinate, context-independent truth naturally admit formal, cryptographic, or exact verification mechanisms, including proof assistants, SMT-based checking, zero-knowledge proofs, and deterministic test suites. Such mechanisms enable conclusive verification that is independent of deployment conditions and is therefore highly automatable. This is evident in software and hardware verification, mathematical reasoning, privacy enforcement, and distributed training equivalence [36, 45, 43, 44, 31].

By contrast, tasks whose correctness is determinate but context-specific typically rely on runtime monitors, policy checking, execution-trace validation, and specification-driven guards, where correctness is defined relative to a stable environment, protocol, or domain model. In these settings, verification enforces compliance with contextual constraints rather than universal correctness. This mode of verification recurs in agent safety, tool-use workflows, credential verification, legal reasoning, and scientific pipelines [14, 21, 10, 48].

As epistemic uncertainty increases, verification mechanisms shift away from outcome-level correctness toward system-level properties such as robustness, calibration, consistency, assumption transparency, and controllability. This shift reflects the fact that ground truth may be delayed, partial, or fundamentally unobservable at the instance level. Verification therefore operates over aggregate behaviour and structural guarantees rather than individual outputs. This transition is apparent in autonomous driving, robotics, scientific discovery, and probabilistic verification [25, 46, 47, 29].

In domains where truth is indeterminate or inherently contested, verification becomes primarily procedural. Rather than adjudicating correctness, verification ensures that decision-making processes satisfy requirements of legitimacy, fairness, transparency, auditability, and appropriate human oversight. In such settings, trust is established not through convergence on a single "correct" answer, but through guarantees about how disagreements are handled and decisions are justified [34].

Problem Classification Dimensions. Thus, we classify problem types along two orthogonal dimensions: (1) *epistemic clarity*: whether a determinate true answer exists and is directly verifiable, exists but is difficult to establish, or does not exist at all; and (2) *context dependence*: whether correctness is invariant across contexts, context-specific but stable, or contested across perspectives. Together, these dimensions yield six recurring problem types. Each type exhibits distinct implications for uncertainty, oversight, solve-verify asymmetry, appropriate problem decomposition strategies, and verification mechanisms.

Problem-Type Template. To make these distinctions operational, we characterise each problem type using a descriptive template:

- **Summary**: a concise characterisation of the problem type.
- **Examples**: representative real-world use cases.
- **Uncertainty**: who is uncertain (the AI system, users, or verifiers) and about what.
- **Oversight**: the role and intensity of human involvement.
- **Solve-Verify Asymmetry**: the relative difficulty of producing solutions versus verifying them.
- **Decomposition Strategy**: how the problem should be structured so that verification remains effective and scalable.
- **Verification**: what properties can be checked automatically at the system and case level.

Problem Type 1: Determinate truth, context-independent.

- **Summary**: A single correct answer exists and does not vary with context.
- **Examples**: Physics equations, orbital mechanics, prime factorisation.
- **Uncertainty**: Once correctly designed, the AI system, users, and verifiers all face low uncertainty.
- **Oversight**: Minimal; automated verification is sufficient.
- **Solve-Verify Asymmetry**: Very strong. Finding a solution may be costly, but checking it is cheap (e.g., verifying a hash is easy even if finding a preimage is hard).
- **Decomposition Strategy**: Isolate certifiable subproblems and exploit automated verification (e.g., proofs, certificates, error bounds), with minimal human oversight.
- **Verification**: Focus on correctness. System-level evaluation compares outputs to ground truth; case-level verification uses proofs, certificates, or error bounds.

Problem Type 2: Determinate truth, context-specific but stable.

- **Summary**: The correct answer depends on context but remains uniquely defined.
- **Examples**: Airline rostering, shortest-path planning, medical dosing with known physiology.
- **Uncertainty**: The AI system may be uncertain due to noisy or incomplete inputs; users are clear once objectives are specified; verification is low-uncertainty because feasibility can be checked deterministically.

Table 3: Problem Decomposition for Verifiability-First AI Engineering

Problem Type	Solve–Verify Asymmetry	Decomposition Strategy	Verification Focus
Type 1: Determinate truth, context-independent	**Strong**	Decompose into certifiable subproblems with explicit, invariant correctness conditions	Direct correctness via proofs, certificates, or exact checks
Type 2: Determinate truth, context-specific but stable	**Strong**	Separate context-specific inputs from invariant rules and constraints	Feasibility, constraint satisfaction, and regret across contexts
Type 3: Epistemic uncertainty, context-independent	**Weak**	Prioritise system-level properties over individual outputs	Consistency, calibration, robustness, and ensemble agreement
Type 4: Epistemic uncertainty, context-specific	**Mixed**	Structure guardrails, monitoring, and adaptive feedback mechanisms	Assumption transparency, stability under perturbation, and rollback conditions
Type 5: Indeterminate truth, context-specific	**Moderate**	Separate preference elicitation from optimisation or recommendation generation	Preference alignment, autonomy preservation, and reversibility
Type 6: Indeterminate truth, context-contested	**Strong**	Structure decision processes rather than outcomes	Procedural legitimacy, including fairness, transparency, and due process

- **Oversight**: Human oversight is applied by exception, for out-of-distribution inputs or conflicting constraints.
- **Solve-Verify Asymmetry**: Strong. Optimisation is hard; feasibility checking is easy.
- **Decomposition Strategy**: Separate context-specific inputs from the rules and constraints that must always be satisfied.
- **Verification**: Verification checks whether each solution is feasible and meets all constraints, and compares its quality to alternative solutions when needed. System-level evaluation compares performance across contexts (e.g., regret); case-level verification checks constraint satisfaction.

Problem Type 3: Epistemic uncertainty, context-independent.

- **Summary**: A true answer may exist, but complex dynamics make it hard to confirm.
- **Examples**: Climate modelling, long-term macroeconomic forecasting.
- **Uncertainty**: The AI system is uncertain about model adequacy; users are uncertain about prediction reliability; verification is uncertain because ground truth emerges slowly.
- **Oversight**: Human oversight is required for modelling choices and interpretation.
- **Solve-Verify Asymmetry**: Weak. Both solving and verifying are difficult.
- **Decomposition Strategy**: Prioritise system-level properties (such as consistency, calibration, robustness) over individual outputs.
- **Verification**: Verification focuses on consistency rather than correctness. System-level checks focus on calibration, robustness, and ensembles; case-level verification is limited to consistency checks.

Problem Type 4: Epistemic uncertainty, context-specific.

- **Summary**: Correctness is unstable due to complexity, feedback loops and heterogeneity.
- **Examples**: Public policies with complex societal effects, such as carbon pricing or migration policy.
- **Uncertainty**: AI uncertainty is high because causal pathways are unclear; users are uncertain because assumptions shape conclusions; verifiers are uncertain because impacts unfold slowly.
- **Oversight**: Continuous human oversight and adaptive governance are essential.
- **Solve-Verify Asymmetry**: Mixed; outcomes are hard to verify, guardrails are easier.

- **Decomposition Strategy**: Emphasise guardrails, monitoring, and adaptive feedback.
- **Verification**: Verification focuses on transparency, assumptions, and rollback capacity rather than final outcomes. System-level testing uses pilot programs, causal inference, and natural experiments. Case-level verification is limited to ensuring assumptions and constraints are transparent.

Problem Type 5: Indeterminate truth, context-specific.

- **Summary**: Each individual has a "right" answer, but across people there is irreducible diversity.
- **Examples**: Career advice, diet or entertainment recommendations.
- **Uncertainty**: AI is uncertain about latent preferences; users may be uncertain about future preferences; verification is subjective.
- **Oversight**: Focus on autonomy and reversibility.
- **Solve-Verify Asymmetry**: Moderate. It is hard to find the best option, easier to verify that recommendations match constraints and preferences.
- **Decomposition Strategy**: Separate preference elicitation from recommendation generation.
- **Verification**: Verification ensures autonomy, constraint respect, and reversibility rather than outcome correctness. System-level evaluation focuses on preference elicitation quality, stability, and fairness. Case-level verification checks that options respect stated constraints and allow opt-out.

Problem Type 6: Indeterminate truth, context-contested.

- **Summary**: No unique answer exists due to plural and conflicting values.
- **Examples**: Distributive justice in climate policy, free speech vs harm trade-offs in content moderation, resource allocation in public health.
- **Uncertainty**: AI has little uncertainty in calculation but cannot collapse plural values; users are uncertain because preferences conflict; evaluation is uncertain because legitimacy replaces accuracy.
- **Oversight**: Structured, multi-stakeholder human oversight.
- **Solve-Verify Asymmetry**: Strong but procedural. Substantive "correctness" is not verifiable, but procedural guarantees are.
- **Decomposition Strategy**: Design for legitimacy, due process, and accountability.
- **Verification**: Verification is primarily procedural (such as fairness and transparency) and requires structured human oversight.

3.2.2 How is the Problem Decomposed

Across problem types, verification rarely operates on end-to-end outputs directly. Instead, problems are deliberately decomposed into intermediate units that expose verifiable structure. This is a design choice rather than an implementation detail: verification becomes feasible only when a problem is reshaped so that its critical properties can be checked locally, repeatedly, and at acceptable cost. As a result, problem decomposition and verification design are inseparable.

One decomposition strategy is atomic decomposition, in which complex outputs are broken into minimal, independently verifiable units such as claims, steps, predicates, or assertions. This approach is effective when holistic correctness is difficult or ambiguous to assess. By isolating atomic units, verification focuses on local soundness, consistency, or evidence support prior to aggregation. Verification is thus shifted from opaque final answers to explicit intermediate structure, trading global simplicity for local checkability.

Another strategy is specification-driven modularisation. Here, problems are structured around explicit specifications such as preconditions, postconditions, invariants, or logical constraints. Verification targets are defined by these specifications rather than by surface-level outputs, allowing modules or stages to be verified independently. This strategy is most effective when correctness conditions can be stated precisely, even if solution construction remains complex or uncertain.

In agentic and multi-agent systems, decomposition often takes the form of pipeline and role separation. Rather than treating an agent as a monolithic policy, systems are structured into distinct components such as planning, execution, monitoring, verification, and auditing. Verification responsibilities are assigned explicitly to particular roles or stages, enabling independent checking of behaviour without requiring full introspection of the agent's internal decision process. This structure supports scalability and accountability for systems operating over extended time horizons.

A further strategy is representation lifting, where problems are translated into alternative representations that admit stronger verification guarantees. This includes formalising natural language into logic, abstracting high-dimensional trajectories into symbolic summaries, or encoding computation into representations suitable for mathematical or cryptographic checking. The objective is not semantic completeness, but the exposure of properties that can be verified with high confidence.

Some systems instead employ interaction-based or adversarial decomposition, treating verification as an interaction between distinct roles rather than a static check. Verification emerges through structured challenge–response dynamics or competitive evaluation, rather than reliance on a single authoritative verifier. This approach is particularly useful when individual verifiers are fallible or biased.

Verification-oriented decomposition therefore prioritises the exposure of intermediate artefacts (such as claims, logs, constraints, abstractions, or proofs) that make checking tractable. Decomposition reshapes the problem so that its critical properties become locally verifiable, allowing systems to trade formulation complexity for scalable and reliable verification when end-to-end correctness cannot be established directly.

3.2.3 At Which Level is Verification Performed

An important design dimension in verifiability-first AI engineering is the *level at which verification operates*. We distinguish between *case-level* and *system-level* verification, not as competing techniques, but as complementary forms of assurance that address fundamentally different verification targets.

Case-level verification concerns discrete, bounded instances of system behaviour. The object of verification may be a single reasoning trace, a generated code artifact, an individual agent action, or a specific model output. What characterises case-level verification is not when it occurs, but what it evaluates: a concrete instance with a well-defined scope. As a result, case-level verification can occur at design time (e.g., validating a generated proof or program), at runtime (e.g., checking an action before execution), or post hoc (e.g., auditing logged behaviour). Its defining feature is locality: each verification decision applies to one instance at a time.

This level of verification is particularly effective when correctness or acceptability can be decided cheaply and reliably for individual cases. It supports iterative refinement mechanisms such as rejection sampling, retries, counterexample-driven repair, or verifier-guided feedback loops, allowing systems to improve behaviour instance by instance [12, 49, 11, 42]. Case-level verification is therefore well suited to tasks where local correctness is meaningful even if global guarantees are unavailable.

System-level verification, by contrast, targets properties that only become meaningful when behaviour is considered in aggregate. These include safety envelopes, policy compliance across time, robustness under distribution shift, interoperability constraints, and governance-level assurances [14, 15, 47, 34]. The verification target is no longer a single output or action, but the behaviour of the system as a whole across executions, components, environments, or operational histories.

System-level verification may be realised through design-time mechanisms such as architectural constraints, certified components, or formally specified invariants, as well as through runtime mechanisms such as continuous monitoring, enforcement layers, or invariant checking across executions. Although runtime observation is often involved, the objective is not to judge individual decisions in isolation, but to ensure that the system's overall behaviour remains within acceptable bounds over time.

Importantly, case-level and system-level verification are not mutually exclusive. Many verifiability-first designs deliberately integrate both levels. In such systems, individual actions or artifacts are verified locally, while the resulting verification evidence is accumulated, summarised, or constrained in ways that support higher-level guarantees about reliability, autonomy, or alignment [17, 50, 51]. Local verification thus serves as an input to system-level assurance rather than as an endpoint.

From a design perspective, the distinction between case-level and system-level verification does not map cleanly onto a design-time versus runtime dichotomy. Both levels can appear at multiple points in the system lifecycle. Instead, the distinction captures *what* is being verified: isolated instances versus emergent global properties. Scalable and trustworthy AI systems increasingly rely on explicit coordination between these levels, using per-instance verification to maintain control locally while enforcing system-wide guarantees through global verification structures.

3.3 When does Verification Occur

Verifiability-first AI engineering spans the full lifecycle of an AI system, but verification plays fundamentally different roles depending on whether it occurs *pre-deployment* or *after deployment*. These phases are not merely temporal stages;

they correspond to distinct verification objectives, actors, feedback loops, and failure modes. Across both phases, the core separation between *solve* tasks and *verify* tasks remains intact, but what is being solved, what is being verified, and how verification influences system behaviour changes substantially.

Pre-Deployment Phase. In the pre-deployment phase, verification is primarily concerned with shaping future system behaviour before it is exposed to real-world consequences. This phase includes system design, model training, fine-tuning, integration, and validation. A significant class of verification mechanisms operate at this stage, particularly those grounded in formal methods, testing, simulation, and training-time supervision (e.g., verifier-in-the-loop generation, formal code verification, training data verification, and process-level supervision) [36, 7, 8, 49].

In this phase, the *solve* task encompasses engineering activities that determine future system behaviour, including defining objectives, constructing architectures, designing verification criteria, and training models or agents. Importantly, verification mechanisms themselves are first-class design artefacts: specifications, test suites, simulators, certificates, learned verifiers, and acceptance thresholds are intentionally constructed to make checking scalable and repeatable. When verification conditions are precise and machine-checkable, they can be embedded directly into development and training loops, allowing verification failures to serve as structured feedback rather than as terminal blockers.

The corresponding *verify* task consists of evaluating artefacts prior to deployment, including code, models, configurations, and emergent system behaviour. This includes static analysis, proof checking, simulation-based stress testing, and benchmark evaluation [25, 46]. Human involvement at this stage therefore concentrates less on inspecting individual outputs and more on designing the verification infrastructure itself.

Post-Deployment Phase. Once deployed, AI systems operate in open, dynamic, and often adversarial environments. Empirically, the dominant pattern across agentic, multi-agent, and tool-using systems is that verification occurs during operation rather than being fully resolved in advance. Post-deployment verification therefore serves as a mechanism for ongoing oversight, behavioural control, and risk mitigation [14, 35, 16, 43, 22, 45].

In this phase, the *solve* task corresponds to the system producing a concrete output, decision, or action in response to live context. The *verify* task is performed by a distinct actor—such as a runtime monitor, verifier model, formal checker, cryptographic protocol, or human—who evaluates the system's behaviour during or after execution. Verification may occur synchronously (e.g., validating an action before execution), asynchronously (e.g., post-hoc audit of execution traces), or continuously (e.g., enforcing invariants across time) [15, 45].

Runtime monitoring deserves particular emphasis. Although it operates at deployment time, runtime monitoring often performs *case-level* verification on individual actions or steps rather than holistic system judgment. Its role is to ensure that local behaviour remains within specified bounds, thereby preventing unsafe escalations before harm propagates. This illustrates that the timing of verification (pre- vs. post-deployment) is orthogonal to the level of verification (case-level vs. system-level).

Problem decomposition based on verification structure remains central after deployment. Real-world tasks are frequently decomposed into steps with heterogeneous verifiability, enabling automated checking where conditions are clear while introducing human judgment or procedural safeguards where correctness is ambiguous or value-laden. Test-time scaling, rejection sampling, verifier-guided selection, and self-correction loops are common mechanisms through which verification continues to shape system behaviour post-deployment [42, 52, 51].

Lifecycle-Spanning Verification. A smaller but important class of systems explicitly span both phases, integrating pre-deployment verification with post-deployment monitoring and auditing. In these systems, verification artefacts constructed during development, such as specifications, proofs, certificates, or learned verifiers, are reused or enforced at runtime, while operational verification signals may feed back into retraining or system updates [17, 34].

3.4 Who Performs Verification

A dimension in verifiability-first AI engineering concerns *who performs verification*. Verification may be carried out by automated tools (e.g., AI tools or other tools), by humans, or by structured combinations of both.

Automated Verification. Verifiability-first design assumes automated verification as the default verifier whenever correctness conditions can be made explicit and machine-checkable. Automated verifiers include formal solvers, compilers, simulators, runtime monitors, cryptographic checkers, and AI-based verifier agents. When verification is automated, it can be executed continuously, independently, and at scale, without becoming a bottleneck to system autonomy.

Across system designs, automated verification appears in several stable roles. In some systems, verification is performed by *dedicated verifier agents* that operate independently of task-solving agents [14, 45], where lightweight guard or audit agents continuously check actions and intent traces. In others, verification is embedded in *toolchains and formal engines*, such as compilers, theorem provers, simulators, or physics-based solvers [7, 31, 25].

Automated verification may also be *decentralised*. In this regime, verification is carried out by networks of validators or cryptographic mechanisms rather than by a single trusted authority, allowing trust to be distributed across independent parties [22, 24]. A related design places verification directly at interaction boundaries, where peer agents verify one another's credentials or claims at run time, eliminating reliance on a central verifier and enabling trust to emerge from protocol-level checks [21].

A key advantage of automated verification is *stackable improvement*. Once a subtask, decision, or intermediate result has been verified, it can be treated as stable and need not be re-verified, allowing verified progress to accumulate across iterations [41, 51, 52]. This property enables scalable refinement mechanisms such as rejection sampling, verifier-guided retries, and test-time compute scaling, where additional computation reliably improves outcomes without invalidating previously verified components.

Human Verification.. Human verification plays a distinct and complementary role within verifiability-first system design. Rather than acting as a general fallback for automation failure, human involvement is introduced deliberately where verification cannot be fully mechanised. This includes situations in which intent must be interpreted, values are contested, assumptions must be negotiated, or where procedural legitimacy outweighs factual correctness.

Within such designs, humans typically function as *final arbiters* or *oversight actors*, intervening only after automated mechanisms have exhausted their checking capacity [39, 53]. This role is particularly salient in domains requiring deep domain expertise or contextual judgment, where errors may only become apparent through careful interpretive scrutiny, as illustrated by cases of expert-led post hoc fact checking in scientific discovery. In other settings, human verification is positioned earlier in the loop through *specification confirmation* or *intent clarification*, ensuring that the properties being verified faithfully reflect human goals before automated verification is applied [17, 54].

Importantly, human verification is typically scaffolded by automation rather than replacing it. Automated mechanisms perform routine, scalable checks and surface uncertainty, while human validation is triggered selectively when confidence thresholds are violated or ambiguity persists [55, 56]. This division of labour preserves scalability while reserving human judgment for precisely those cases where it adds the most value.

Hybrid Verification. Many scalable system designs deliberately separate verification responsibilities across multiple actors. A single system may involve a solver that produces candidate actions or outputs, one or more automated verifiers that check formal or operational properties, cryptographic mechanisms that ensure integrity or provenance, and human overseers who assess intent, legitimacy, or unresolved ambiguity [15, 47]. Verification is therefore structured as a layered process, with each actor responsible for distinct properties, allowing systems to balance rigor, efficiency, and interpretability without overloading any single verification component.

3.5 Why Verification is Required

The purpose of verification in AI engineering extends beyond checking whether system outputs satisfy explicit requirements or constraints. As AI systems become more autonomous, adaptive, and embedded in real-world environments, verification serves as a mechanism for risk containment, accountability, scalable improvement, and the preservation of human agency and understandability.

Preventing Irreversible and High-Impact Failures. A primary motivation for verification is the prevention of irreversible or high-impact failures. Autonomous systems increasingly invoke external tools, interact with physical infrastructure, manipulate sensitive data, and coordinate with other agents. In such settings, a single unverified action may cause harm that cannot be easily undone. Verification therefore functions as a pre-emptive safeguard, ensuring that unsafe behaviours are detected and blocked before execution rather than relying on post hoc correction. This motivation underlies systems designed for runtime action gating and enforcement [14, 16, 25, 28].

Mitigating Hallucination, Error Propagation, and False Confidence. A second core motivation is the mitigation of hallucination, logical error, and error propagation in generative and agentic systems. Large language models often produce outputs that are fluent and confident yet incorrect, internally inconsistent, or ungrounded. In multi-step reasoning, scientific analysis, legal advice, and multi-agent collaboration, such errors can cascade across steps or agents, amplifying their impact. Verification introduces explicit checks on intermediate claims, retrieved evidence, reasoning

steps, or execution traces, reducing reliance on surface plausibility. This motivation is central to approaches [38, 51, 12, 57, 58].

Enabling Accountability Without Full Disclosure. Verification also enables accountability in settings where full transparency is infeasible or undesirable. In regulated, decentralised, or competitive environments, it is often unacceptable to expose internal states, training data, or proprietary decision logic. Verification mechanisms such as cryptographic proofs, execution certificates, and verifiable traces allow external parties to confirm that a system behaved safely, lawfully, or within policy constraints without requiring access to sensitive internals. This motivation is explicit in designs [15, 43, 22, 23].

Supporting Scalable Learning and Improvement. Verification further supports scalable learning and optimisation by providing structured, repeatable feedback that is cheaper and more reliable than manual human annotation. By verifying intermediate steps or partial solutions, systems can exploit process-level supervision rather than relying solely on final outcomes. This enables iterative refinement, rejection sampling, counterexample-driven repair, and test-time scaling, where verified progress accumulates incrementally. This role of verification is central to approaches [49, 52, 41, 59].

Preserving Human Understandability and Agency. A further motivation concerns the long-term preservation of human understandability and agency. As systems optimise performance, they may evolve behaviours that are effective but increasingly opaque to human stakeholders. Verification imposes a requirement that systems expose artefacts—such as rationales, constraints, certificates, or structured traces—that can be meaningfully assessed by humans. This ensures that trust does not rely solely on aggregate performance metrics, but on behaviours and representations that remain intelligible and contestable [42, 47, 53, 33].

4 Design Principles for Verifiability-First AI Engineering

Based on the conceptual framework introduced in Section 2 and the findings of the systematic literature review, we derive a set of design principles for verifiability-first AI engineering. These principles organise recurring verification practices identified in the literature into normative guidance for the design of verification. The principles impose explicit design constraints that support verifiability-first AI engineering and form the conceptual foundation for the architectural design patterns introduced in Section 5.

4.1 Principle 1: Decompose Problems Around Verifiable Behavioural Claims

Verification targets must be defined as explicit claims about system behaviour, expressed through artefacts and properties that can be verified reliably and at scale. Problems should be decomposed around the structures required to make those claims verifiable.

In verifiability-first AI engineering, verification begins by identifying which behavioural claims matter for assurance, such as correctness, constraint satisfaction, robustness, calibration, transparency, autonomy preservation, or procedural legitimacy. These claims may be embodied in artefacts such as models, specifications, traces, certificates, or intermediate representations, but the artefacts are chosen because they expose verifiable properties, not because they follow conventional development stages.

Problem decomposition is therefore driven by verification structure rather than by inherited workflows. The central question is which intermediate artefacts or behavioural claims can be formulated so that their associated properties admit local, repeatable, and scalable verification. Effective decomposition introduces intermediate claims, representations, or abstractions that simplify verification, even if doing so increases the complexity of solving.

Verification mechanisms must align with the epistemic structure of the problem. Problems with determinate truth admit direct correctness checks. Problems characterised by epistemic uncertainty require verification of consistency, calibration, robustness, or assumptions. Problems involving indeterminate or contested outcomes require procedural or legitimacy-based verification rather than outcome correctness. Decomposition should make these distinctions explicit.

This principle treats problem formulation and decomposition as primary levers for scalable verification, ensuring that both artefacts and their properties are selected to support effective and composable verification rather than inherited by default.

4.2 Principle 2: Explicitly Separate Solve and Verify Tasks and Exploit Solve–Verify Asymmetry Where Available

AI systems must explicitly separate tasks that produce behaviour (solve) from tasks that assess, constrain, or interpret that behaviour (verify), and must treat verification as an independent, first-class concern rather than as an informal or downstream check.

This separation is not conceptually new. Software engineering has long distinguished construction from verification, typically with humans responsible for both, supported by tools. What has changed is the scaling regime. In contemporary AIware-centric systems, the cost of producing candidate behaviours has collapsed. Behaviour generation is cheap, parallel, and abundant, while verification remains comparatively expensive if it relies on manual judgment or ad hoc inspection. If verification is not explicitly separated and designed for scale, it becomes the dominant bottleneck.

Explicit separation enables systems to exploit solve–verify asymmetry where it exists. Many problems admit an inherent asymmetry in which generating candidate behaviours is expensive or uncertain, but checking acceptability against well-defined criteria is cheap, reliable, and repeatable. When solve and verify tasks are cleanly separated, systems can safely scale behaviour generation through verifier-guided generation, rejection sampling, constrained exploration, or iterative refinement, without weakening assurance.

Where strong solve–verify asymmetry does not arise naturally, verifiability-first AI engineering requires deliberate problem reformulation. Problems should be decomposed so that acceptability conditions become machine-checkable, compositional, or locally decidable, even if full end-to-end correctness remains intractable. This may involve introducing intermediate behavioural claims, proxy verification targets, or incremental verification steps that preserve partial guarantees and still compose.

This principle establishes the foundation of verifiability-first AI engineering: verification is not an optimisation layered onto generation, but a primary design objective that shapes how problems are formulated, how behaviours are produced, and how systems scale safely under uncertainty.

4.3 Principle 3: Design Verification Across the Lifecycle and Across Levels

Verification must be treated as a lifecycle-spanning concern, spanning both pre-deployment construction and post-deployment operation.

Before deployment, verification shapes problem formulation, specifications, training objectives, and acceptance criteria through machine-checkable conditions and verifier-guided feedback. After deployment, verification provides ongoing oversight, behavioural control, and adaptation under real-world conditions. Verification artefacts, such as specifications, invariants, certificates, learned verifiers, or monitoring logic, should be designed for reuse across phases rather than reconstructed ad hoc.

Within this lifecycle-spanning view, verification must explicitly distinguish between case-level and system-level verification. Case-level verification evaluates discrete instances of behaviour, such as individual outputs, actions, reasoning traces, or execution steps. System-level verification evaluates properties that emerge only in aggregate, including safety envelopes, policy compliance over time, robustness under distribution shift, and governance guarantees.

This distinction concerns what is verified, not when verification occurs. Both case-level and system-level verification may appear at design time, at runtime, or post hoc. Effective systems coordinate these levels explicitly, using local verification to constrain individual behaviour while enforcing global guarantees through system-level verification structures.

4.4 Principle 4: Design for Stackable and Incremental Verification with Scalable and Verifiable Rewards

Verification should be stackable and incremental, such that verified claims compose deductively over time rather than being repeatedly re-established from scratch.

Once a behaviour, subtask, or component has been verified under an explicit set of assumptions, that verification result should be treated as locked. It remains valid and reusable as subsequent behaviour is generated, provided its assumptions continue to hold. Re-verification should occur only when those assumptions are weakened, violated, or explicitly revised.

This principle treats verification as an accumulative control mechanism. Problem decomposition should therefore isolate verifiable units with clear scope, assumptions, and dependencies, so that verification results can be sealed, referenced, and composed across iterations, refinements, and partial updates. Progress becomes monotonic: systems advance by extending verified structure rather than revisiting settled claims.

In learning-based systems, this principle is operationalised through scalable and verifiable rewards. A reward is verifiable when the correctness of assigning that reward can itself be checked under explicit criteria, rather than relying solely on opaque preference models, ad hoc heuristics, or irreproducible human judgment. A reward mechanism is scalable when both reward evaluation and the reference standards it depends on can be maintained and reused without proportional growth in oversight cost.

When rewards are both verifiable and scalable, learning updates correspond to verified behavioural claims that can be sealed and reused, rather than provisional signals that must be continuously reinterpreted or re-justified. Learning progress can therefore accumulate deductively, enabling optimisation to proceed without eroding previously established guarantees.

Where feasible, reward signals should be grounded in machine-checkable or procedurally stable properties such as constraint satisfaction, formal correctness, policy compliance, validated intermediate claims, synthetic or game-based reference processes, or reproducible evaluation protocols. Scalable and verifiable rewards align optimisation with verification by ensuring that learning reinforces behaviours whose acceptability can be independently checked and preserved over time, enabling incremental improvement at scale.

4.5 Principle 5: Allocate Verification Actors Explicitly and Preserve Human Understandability

Verification must explicitly specify who performs verification, whether automated tools (AI or other tools), humans, or structured combinations of both, and must preserve human understandability wherever humans are responsible for verification.

Verifiability first systems recognise three modes of verification: automated, human, and hybrid. Automated verification should be the default wherever verification criteria are explicit, stable, and machine checkable, enabling scale, consistency, and continuous enforcement. Human verification is introduced selectively where verification cannot be fully mechanised, such as when intent must be interpreted, assumptions negotiated, values contested, or procedural legitimacy is required. Hybrid verification combines these roles by using automation to perform routine checks, expose uncertainty, and focus human attention on cases that genuinely require judgment.

The allocation of verification actors is therefore a design decision grounded in the verifiability structure of the problem, not an operational fallback for insufficient automation. Verification responsibilities must be assigned explicitly, with clear scope, authority, and accountability, and with outcomes that are inspectable and auditable.

Human understandability follows directly from this allocation. When humans are responsible for verification, systems must expose behaviour in forms that humans can meaningfully evaluate. From a verifiability first perspective, understandability is itself a verification requirement. Systems should therefore provide verifiable signals such as rationales, constraints, assumptions, uncertainty indicators, execution traces, or procedural records that enable evaluation, contestation, and justification. Without such representations, human verification collapses into superficial approval rather than substantive oversight.

5 Architectural Patterns for Verifiability-First AI Engineering

This section presents architectural design patterns that address recurring verification challenges and operationalise design principles. Each pattern specifies its context, problem, solution, quality trade-offs, and real-world known uses.

5.1 Stackable Verification

Summary Stackable verification structures a system into independently verified steps that are sealed once verified and then safely reused to enable modular reasoning and scalable assurance.

Context In AIware systems, behaviour is increasingly generated dynamically by AI agents at runtime rather than being fully specified at design time. Such systems often produce multi-step workflows in which individual steps may invoke external tools, knowledge bases, or other AI agents.

Problem Individual steps within an AI-generated workflow can be expensive to obtain and costly to verify. Verification may require running large-scale simulations, invoking formal verification engines, executing domain-specific validation pipelines, or engaging human experts. When a step has already been verified as acceptable under a given set of assumptions, repeatedly re-verifying that step during subsequent workflow evolution introduces unnecessary overhead and latency.

More importantly, AI systems frequently exhibit a failure mode in which progress made in one part of a workflow is undone when later steps are modified or regenerated. Minor downstream changes can trigger full re-verification or even re-solving of earlier steps, despite their assumptions and guarantees remaining unchanged. For complex scientific, engineering, or decision-making tasks where discovering a single valid step may already be difficult, this behaviour destroys accumulated progress and prevents monotonic advancement.

The core challenge is how to enable incremental improvement of AI-generated behavior while avoiding redundant re-verification of previously verified steps.

Solution Stackable Verification structures AI-generated workflows as a sequence of verification-bounded steps that can be incrementally accumulated without regression. The AI system generates a workflow composed of discrete steps, each associated with explicit verification conditions that define when it can be considered acceptable.

Once a step satisfies its verification conditions under a defined context, it is sealed: its verification status is recorded together with the assumptions, dependencies, and scope under which it holds. Sealed steps are treated as stable building blocks. Subsequent steps are constrained to build on top of them rather than rewriting or re-optimising them.

New steps may be appended or refined downstream, but previously sealed steps are reused without re-verification. Re-verification is triggered only when a contextual change invalidates one or more of the recorded assumptions on which a sealed step depends. In this way, verification effort scales with newly introduced behaviour rather than with the entire workflow.

This pattern exploits solve–verify asymmetry by making exploratory generation cheap while ensuring that correctness checks are explicit and automated wherever possible. Verification becomes a ratcheting mechanism: once a step is verified, progress becomes monotonic rather than fragile.

Benefits

- **Preservation of verified progress:** Once a step has satisfied its verification conditions, it is sealed and treated as stable. This prevents previously verified decisions from being unintentionally modified, re-optimised, or invalidated as the system explores downstream alternatives. Verified progress is therefore preserved rather than repeatedly undone.
- **Reduction of redundant verification effort:** Sealed steps are reused without re-verification, avoiding repeated execution of expensive verification procedures such as simulation, formal analysis, or human review. Verification effort is focused on genuinely new or modified behaviour, rather than rechecking unchanged components.
- **Support for iterative problem solving:** By ensuring that verified steps remain durable, the pattern enables a ratcheting form of problem solving in which the system incrementally extends behaviour without erasing earlier achievements. This is particularly valuable in exploratory or open-ended tasks where progress would otherwise be fragile.

Drawbacks

- **Risk of latent correctness or safety violations:** If the assumptions or contextual dependencies under which a step is sealed are incompletely specified, incorrectly recorded, or later invalidated, the system may continue to reuse verification guarantees that no longer hold. Such mismatches can introduce latent correctness or safety violations that may only surface in downstream behaviour.
- **Reduced global optimality and adaptability:** Treating sealed steps as stable constrains the system's ability to revisit earlier decisions. While this stabilises verified behaviour, it may prevent exploration of globally superior solutions when objectives, constraints, or optimisation criteria evolve over time.
- **Assumption tracking and governance overhead:** Stackable verification requires explicit representation and maintenance of assumptions, dependencies, and verification scope for each sealed step. This increases engineering complexity and introduces new failure modes, such as incomplete dependency capture or inconsistent assumption management.
- **Limited effectiveness in tightly coupled workflows:** In workflows with strong global constraints or high inter-step coupling, isolating steps for independent sealing may require conservative assumptions or coarse-grained verification units. This can reduce composability and diminish the benefits of fine-grained step-level sealing.

Known Uses

- **AlphaEvolve**[2]: AlphaEvolve demonstrates stackable verification in evolutionary AI systems. Candidate algorithms are generated through local mutations of previously verified programs. Each mutation is evaluated against functional correctness conditions and performance constraints. Variants that satisfy these conditions are retained and reused as parents without re-verification, while verification effort is concentrated on newly introduced changes. Re-verification is required only when assumptions, benchmarks, or execution contexts change.
- **Compositional Verification for Autonomous Driving**[3]: Researchers at MIT developed a verification framework for self-driving car controllers that uses assume-guarantee contracts on road segments. Instead of verifying an entire city's driving scenario at once, they verify discrete road components in isolation under specific assumptions.
- **NASA's Mars Exploration Rovers**[4]: NASA's Mars Exploration Rovers utilize the Field D* algorithm to optimize autonomous navigation by caching validated path segments. Rather than recomputing a full route when obstacles appear, the system preserves and reuses previously verified-safe coordinates, repairing only the specific portions of the path affected by new hazards.

5.2 Solver-Verifier Separation

Summary The Solver-Verifier Separation pattern decouples the logic used to generate a solution (the Solver) from the logic used to evaluate it (the Verifier).

Context AIware systems increasingly rely on LLMs and agentic frameworks to generate answers, plans, tool invocations, and code dynamically at runtime. These systems often operate in open-ended, multi-step workflows where behaviour evolves as execution progresses.

Problem In many contemporary LLM- and agent-based systems, verification is performed in an ad hoc and implicit manner. Acceptability is assessed through scattered heuristics, prompt constraints, confidence thresholds, or informal LLM-based judgments, rather than through an explicit verification architecture.

Because verification logic is intertwined with generation or embedded opportunistically across the system, verification assumptions are rarely made explicit, systematically enforced, or composed across steps. In multi-step or agentic workflows, this leads to fragile guarantees and limited ability to reason about correctness, safety, or failure modes over time.

The core challenge is how to transform verification from a secondary, informal activity into an explicit, independent system capability and a first-class architectural role.

Solution Solver–Verifier Separation introduces an explicit architectural boundary between generation and verification. The *solver* produces candidate behaviours, while the *verifier* independently evaluates those candidates against defined acceptability criteria.

Only behaviours that satisfy verification conditions are executed or committed. Verification logic remains external to the solving process and may involve automated checks, simulation, formal constraints, or human judgment. Solvers may iterate in response to verifier feedback, but acceptance authority remains with the verifier.

Benefits

- **Explicit verification semantics:** Acceptability criteria are clearly defined, inspectable, and enforceable.
- **Composable assurance:** Verification guarantees can be reasoned about and accumulated across steps.
- **Reduced brittleness:** System behaviour becomes more robust to changes in prompts, models, or workflows.
- **Alignment with verifiability-first design:** Verification is elevated from an implementation detail to an architectural concern.

[2] `https://deepmind.google/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/`
[3] `https://dspace.mit.edu/handle/1721.1/114151`
[4] `https://ntrs.nasa.gov/citations/20100010963`

Drawbacks

- **Additional design effort:** Explicit modelling of verification criteria and interfaces are required.
- **Verifier quality dependence:** Weak or incomplete verifiers limit the value of separation.
- **Runtime overhead:** Verification steps may add latency or computational cost.

Known Uses

- **AlphaCode**[5]: AlphaCode generates a very large set of candidate programs (solver) and then uses execution-based filtering and selection against tests and other checks (verifier) to determine which candidates are acceptable for submission. [**?**]
- **Anthropic Constitutional AI and Classifiers**[6]: Outputs from the solver model are evaluated against a predefined "constitution" using separate classifier layers, demonstrating a distinct verification process aligned with explicit normative principles rather than implicit acceptance.
- **Palantir AIP Logic with AIP Evals**[7]: In enterprise usage, AI-generated logic and function definitions are verified by a distinct evaluation suite (AIP Evals), separating generation from systematic validation before deployment in production workflows.

5.3 Sandbox Gatekeeper

Summary The Sandbox Gatekeeper isolates solver-generated behaviour within a restricted execution environment and promotes its effects to the production system only after a Gatekeeper verifies that execution satisfies defined correctness, safety, or policy criteria.

Context Many AIware systems generate executable behaviour, such as code, queries, configurations, or API calls. In principle, the correctness or safety of such behaviour is often easiest to verify by executing it. In practice, however, executing unverified behaviour directly in a production environment collapses the distinction between verification and commitment, because harmful effects may occur before verification completes. This coupling makes verification expensive, risky, and difficult to scale, particularly during development, learning, simulation, or policy exploration.

Problem When solver-generated behaviour is executed in the primary system context, verification becomes entangled with real-world consequences. Fear of irreversible side effects forces verification to be conservative, manual, or selectively applied, undermining solve–verify asymmetry and limiting scalability. The challenge is therefore not only safety, but scalability: how to empirically verify generated behaviour while preserving a clear acceptance boundary and preventing real-world effects from dominating verification cost.

Solution The Sandbox Gatekeeper introduces an isolated execution environment in which solver-generated behaviour can be executed without affecting real system state. A separate Gatekeeper component observes sandbox execution and evaluates it against verification criteria such as correctness, safety, robustness, or policy compliance.

Only when sandbox execution satisfies these criteria are effects authorised for promotion to the production environment. Failed, unsafe, or non-compliant executions are discarded without external impact.

By decoupling execution from commitment, the sandbox restores solve–verify asymmetry: execution can be freely repeated, instrumented, and inspected, while verification remains cheaper than real-world correction.

Benefits

- **Restored verification asymmetry:** Verification becomes inexpensive relative to real-world failure by eliminating irreversible effects.
- **Scalable experimentation:** Enables high-throughput verification during development, learning, simulation, and policy testing.
- **Controlled state transitions:** All effects crossing the sandbox boundary are explicitly authorised.
- **Strong fault containment:** Unsafe behaviour cannot affect production systems.

[5] `https://deepmind.google/blog/competitive-programming-with-alphacode/`
[6] `https://www.anthropic.com/research/constitutional-classifiers`
[7] `https://palantir.com/docs/foundry/aip-evals/overview/`

Drawbacks

- **Verification fidelity risk:** Divergence between sandbox and production environments may invalidate conclusions.
- **Environmental cost:** High-fidelity sandboxes may be expensive to construct or maintain.
- **Limited applicability:** Not all tasks admit executable or sandboxable verification, particularly those involving human or institutional effects.

Known Uses

- **ChatGPT Code Interpreter**[8]: Generated Python code is executed in a sandboxed environment before returning verified results.
- **Palantir AIP**[9]: AI-generated actions are validated against policy and safety constraints before being committed to operational systems.
- **Replit AI**[10]: AI-generated code is developed and tested within ephemeral containers. Following a major 2025 incident where an agent deleted live production data, Replit introduced Plan Mode, a gatekeeping layer that requires manual or policy-based validation of agent-proposed changes before they affect operational systems.

5.4 Evidence-Augmented Generation

Summary Evidence-Augmented Generation requires the solver to produce both a solution and an explicit verification trace, enabling an independent verifier to validate correctness without re-solving the problem.

Context In domains such as law, medicine, science, and complex engineering, the acceptability of a solution depends not only on the outcome but also on the reasoning, justification, or provenance behind it. Opaque solutions are difficult to audit, contest, or trust.

Problem For complex tasks, independently verifying a solution is often as difficult as generating the solution itself. When a solver provides only a final answer, the verifier must reconstruct the reasoning process, leading to redundant computation, high verification cost, and loss of information discovered during solving. This undermines solve–verify asymmetry and discourages scalable verification.

Solution Evidence-Augmented Generation requires the solver to emit two structured outputs: a solution and an associated evidence trace, such as citations, intermediate derivations, proof steps, or execution logs. The verifier's role is limited to checking the validity of the relationship between the evidence and the solution, rather than rediscovering the solution independently.

Benefits

- **Improved transparency:** Verification traces provide a human-readable audit path.
- **Efficient verification:** The verifier operates over constrained evidence rather than the full problem space.
- **Stronger solve-verify asymmetry:** Solving remains exploratory while verification becomes lightweight.

Drawbacks

- **Solver overhead:** Producing structured evidence increases generation cost and complexity.
- **Evidence integrity risk:** Weak or informal verifiers may accept fabricated or misleading evidence.

Known Uses

- **Perplexity AI**[11]: Answers are accompanied by explicit source citations, enabling users to act as verifiers by checking the supporting evidence without independently reconstructing the answer.

[8] https://openai.com/index/chatgpt-plugins/#code-interpreter
[9] https://www.palantir.com/platforms/aip/
[10] https://replit.com/ai
[11] https://www.perplexity.ai

- **Lean Theorem Prover**[12]: Evidence is produced by a solver (human or automated tactic), while Lean's small trusted kernel mechanically verifies the proof term, checking correctness without re-deriving the proof.
- **Rocq Prover**[13]: Proofs are constructed interactively or via automation, and the system's kernel verifies that each step adheres to the rules of the Calculus of Inductive Constructions, turning verification into a deterministic checking process.

5.5 Verification-Driven Workflow Decomposition

Summary Verification-Driven Workflow Decomposition decomposes a complex problem by identifying subproblems according to the verification conditions they admit, constructing a workflow in which each step produces a behavioural claim that can be verified independently and incrementally.

Context AIware systems increasingly operate over problems whose solutions manifest as complex behaviours, such as plans, decisions, analyses, or sequences of actions, rather than as single outputs. These behaviours are subject to heterogeneous constraints, including logical validity, safety, policy compliance, robustness, and uncertainty calibration.

Such constraints differ not only in content but in how—and whether—they can be verified automatically. This variation should shape how problems are decomposed, rather than being addressed after the fact.

Problem Conventional problem decomposition follows functional or workflow conventions (e.g., planning, generation, execution), with verification applied retrospectively to the artefacts these steps produce. This approach obscures the underlying verification structure of the problem and prevents systematic exploitation of solve–verify asymmetry.

As a result, verification is often expensive, weakly automated, and non-compositional. Failures are discovered late, verified progress is frequently invalidated by downstream changes, and human involvement is introduced as a generic fallback rather than for well-defined epistemic reasons.

The core challenge is how to decompose problems so that verification is not an afterthought, but a primary driver of the decomposition itself.

Solution Verification-Driven Workflow Decomposition structures a problem as a sequence of subproblems, where each subproblem is defined by an intermediate behavioural claim and an associated verification condition. The primary design decision is the placement of step boundaries such that each intermediate claim admits independent, low-cost verification and meaningfully constrains subsequent solving.

For example, in an agentic planning task, the first step may produce not a plan, but a claim that a given set of goals is mutually consistent under a stated world model. This claim can be verified using constraint satisfaction or model checking, and downstream planning is formulated only over goal sets that have passed this verification. The intermediate result is therefore a verified feasibility claim, not a partial solution.

In a code-generation task, an early step may produce a candidate type signature or interface specification derived from the problem description. This intermediate result can be verified using static type checking or schema validation, before any executable logic is generated. Subsequent steps are constrained to produce code that conforms to the verified interface.

Each step specifies: (1) the intermediate behavioural claim it establishes (e.g., feasibility, consistency, admissibility), (2) the assumptions under which the claim holds, and (3) a verification condition that determines whether the claim is accepted.

The workflow is defined so that downstream steps are formulated in terms of verified intermediate claims rather than raw outputs. Successful verification establishes a stable intermediate result that need not be revisited unless its assumptions are invalidated.

Benefits

- **Stackable and monotonic progress:** Verified intermediate claims constrain subsequent solving without requiring re-verification, allowing progress to accumulate rather than being repeatedly undone by downstream changes.

[12] `https://leanprover.github.io/`
[13] `https://rocq-prover.org/`

- **Failure localisation and diagnosability:** When verification fails, the failure is associated with a specific behavioural claim and assumption set, making it easier to identify which aspect of the problem formulation is incorrect.
- **Automation-first verification:** By construction, verification targets are chosen to enable automated checking using AI or other tools, reducing reliance on human verification in the normal loop.

Drawbacks

- **Upfront decomposition effort:** Identifying meaningful intermediate behavioural claims and their verification conditions requires rethinking problem formulation and may be non-trivial for novel domains.
- **Incomplete verification coverage:** Some aspects of a problem may resist decomposition into independently verifiable claims, requiring conservative assumptions or deferred verification.
- **Risk of over-constraining search:** Poorly chosen intermediate claims may prematurely restrict the solution space, excluding valid or innovative solutions despite being verifiable.

Known Uses

- **AlphaEvolve (DeepMind)**[14]: AlphaEvolve decomposes algorithm discovery into iterative proposal and evaluation steps, where each candidate program is treated as an intermediate behavioural claim and verified using automated correctness tests and performance benchmarks. Only candidates that pass verification are retained and used to constrain subsequent search, enabling stackable improvement without revisiting previously verified solutions.
- **Lean**[15]: In Lean, complex proof goals are decomposed into sequences of lemmas, each representing an intermediate claim with a precisely defined verification condition. The proof checker automatically verifies each lemma before it can be used in downstream reasoning, ensuring that verified results are reused without re-verification. Problem decomposition is driven by what can be mechanically checked rather than by narrative proof structure.
- **Sketch**[16]: Sketch decomposes programming tasks into partially specified programs and constraints, where intermediate candidates are verified against automatically generated or user-provided specifications. Verification results prune the search space and constrain subsequent synthesis steps, allowing the system to accumulate verified partial structure incrementally.

5.6 Atomic Verifiable Claims

Summary The Atomic Verifiable Claims pattern structures problem solving around minimal behavioural claims that admit independent, low-cost verification, enabling scalable and compositional assurance.

Context AIware systems increasingly generate complex behaviours such as multi-step plans, reasoning traces, code, analyses, or action sequences rather than single, static outputs. These behaviours are often evaluated under heterogeneous constraints, including correctness, safety, policy compliance, robustness, and uncertainty.

In such systems, verification frequently becomes expensive because it is applied to large, entangled behaviours whose correctness is difficult to assess as a whole.

Problem When verification targets are coarse-grained, verifying a generated behaviour can be as difficult as producing it. Verifiers are forced to reconstruct reasoning, re-evaluate global properties, or simulate entire workflows, collapsing solve-verify asymmetry and limiting scalability.

Moreover, failures discovered at the level of large behaviours are hard to localise. A single verification failure may provide little insight into which assumption, step, or constraint was violated, leading to brittle iteration and repeated re-solving.

The core challenge is how to structure generated behaviour so that verification remains cheap, local, and compositional, even as overall behaviour becomes complex.

[14] https://deepmind.google/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/
[15] https://lean-lang.org/
[16] https://people.csail.mit.edu/asolar/SKETCH/

Solution Atomic Verifiable Claims decomposes behaviour into a sequence of minimal claims, each associated with a precise verification condition that can be checked independently.

Rather than attempting to verify an entire behaviour at once, the system is designed so that each step produces an explicit claim, such as feasibility, consistency, admissibility, type correctness, constraint satisfaction, or policy compliance. Each claim is verified in isolation under a clearly defined set of assumptions.

Once verified, an atomic claim can be treated as a stable unit that downstream solving may rely on without re-verification, unless its assumptions are invalidated. Larger behaviours are accepted only as compositions of verified atomic claims, making verification deductive rather than reconstructive.

This pattern shifts verification effort from global judgment to local checking, preserving solve–verify asymmetry by ensuring that verification cost scales with the number of new claims rather than with overall behavioural complexity.

Benefits

- **Low-cost:** Verification operates over small, well-scoped claims rather than large behaviours.
- **Compositional assurance:** Verified claims can be combined deductively to justify complex behaviour.
- **Failure localisation:** Verification failures are tied to specific claims and assumptions, improving diagnosability.
- **Scalability:** Enables verification to keep pace with increasingly complex and agentic behaviour.

Drawbacks

- **Design overhead:** Identifying meaningful atomic claims and verification conditions requires careful problem reformulation.
- **Granularity trade-offs:** Claims that are too fine-grained increase solver verbosity, while overly coarse claims reduce verification benefit.
- **Assumption management:** Each claim requires explicit tracking of scope and dependencies.

Known Uses

- **Lean**[17]: In Lean-based theorem proving, complex mathematical results are constructed as sequences of lemmas, each representing an atomic verifiable claim. Every lemma is independently type-checked by Lean's trusted kernel before it can be referenced by subsequent proofs. Larger theorems are accepted only as compositions of previously verified claims, enabling deductive verification at scale without re-deriving earlier results.
- **CompCert**[18]: The CompCert certified C compiler decomposes correctness into a chain of atomic semantic preservation claims between successive compilation stages. Each stage emits proof objects that are mechanically verified in Coq. The correctness of the compiled binary is established by composing these verified claims rather than verifying the compiler or program monolithically.
- **AlphaCode and competitive programming systems**[19]: AlphaCode evaluates candidate programs against unit tests and execution constraints, each test acting as an atomic behavioural claim about program correctness. Programs are accepted only if all such claims are verified. Verification scales by decomposing correctness into many small, independently checkable properties rather than a single holistic judgment.
- **NASA autonomous planning**[20]: NASA flight and rover planning systems decompose mission plans into verified subplans, each associated with explicit assumptions about environment state and resource availability. Each subplan is validated independently, and higher-level mission guarantees are derived by composing these verified claims under tracked assumptions.

5.7 Scalable and Verifiable Rewards

Summary The Scalable and Verifiable Rewards pattern defines reward signals whose assignment can be independently checked by a verification procedure against explicitly constructed reference standards, enabling learning and optimisation to scale by making both reward correctness and the criteria it depends on verifiable.

[17] `https://lean-lang.org/`
[18] `https://compcert.org/`
[19] `https://deepmind.google/blog/competitive-programming-with-alphacode/`
[20] `https://ntrs.nasa.gov/citations/20100010963`

Context Learning-based AIware systems use reward signals to guide optimisation, exploration, and adaptation. In classical reinforcement learning, rewards are often treated as ground-truth values produced by a well-defined environment. In modern AI systems operating in open-ended, uncertain, or value-laden domains, rewards are frequently derived from human judgments, preference models, or heuristic proxies.

As system capability and autonomy increase, two distinct scalability bottlenecks emerge: 1) Verifier scalability: whether checking correctness is cheaper than generating behaviour. 2) Reference-standard scalability: whether the criteria, data, or constraints being checked against can be produced, maintained, and reused at scale.

Much existing work focuses on accelerating verifiers, but in many practical settings the dominant bottleneck is the second: the absence of reliable historical data, stable labels, or universally agreed correctness criteria. In such cases, verification fails to scale not because checking is expensive, but because there is nothing cheap and stable to check against.

Problem Most contemporary reward and evaluation signals are neither verifiable nor reference-stable. Once a reward or judgment is assigned, there is often no cheaper or independent way to validate that assignment without re-performing the original evaluation, frequently by humans.

This leads to several compounding failures:

- reward assignments cannot be reused deductively across learning iterations or system versions;
- verification effort scales linearly with optimisation rather than constraining it;
- learning exploits proxies that are easy to maximise but difficult to justify;
- oversight cost is dominated by reference data generation rather than checking.

When reference standards are implicit, unstable, or human-dependent, verification collapses into trust. Solve–verify asymmetry disappears even if checking itself is computationally cheap.

The core challenge is therefore not only to design verifiers, but to design verifiable reference standards, which are criteria against which behaviour can be checked cheaply, consistently, and repeatedly.

Solution The Scalable and Verifiable Rewards pattern addresses both bottlenecks by treating reward signals as a special case of constructed reference standards. A reward is verifiable if, given a candidate behaviour and its context, a verifier can determine whether the signal was assigned correctly without re-solving the original task and without regenerating the reference criteria.

This is achieved by explicitly constructing reference standards that are:

- cheap to check;
- stable under refinement;
- reusable across time, agents, and deployments;
- explicit about assumptions and scope.

Depending on problem structure, reference standards may be constructed through:

- **Executable or formal criteria**, such as tests, simulators, specifications, or proof obligations.
- **Synthetic data and environments**, where ground truth is generated procedurally rather than curated historically.
- **Game-based or adversarial setups**, where correctness emerges from interaction rather than labels.
- **LLM-as-judge or model-based evaluators**, constrained by consistency, calibration, or agreement checks rather than assumed correctness.
- **Capability- or constraint-based proxies**, where verification targets demonstrable properties instead of subjective quality.

Rewards become verifiable when their assignment is derived from such constructed standards. Importantly, verifiable rewards do not require encoding absolute correctness. In uncertain or value-contested domains, verification may target procedural properties such as transparency, constraint adherence, robustness, or internal consistency rather than optimality.

Benefits

- **Scalable verification**: Bottlenecks shift from repeated judgment to reusable standards.
- **Explicit assumptions**: Verification criteria are inspectable and auditable.
- **Reduced reward hacking**: Optimisation is constrained by checkable structure rather than informal proxies.
- **Alignment with verifiability-first design**: Learning reinforces behaviours that admit justification.

Drawbacks

- **Design overhead**: Constructing good reference standards is itself an engineering task.
- **Proxy risk**: Poorly chosen standards may encode the wrong incentives.
- **Limited coverage**: Some domains resist any stable or mechanisable reference construction.

Known Uses

- **Sketch**[21]: Sketch assigns rewards based on execution correctness against test suites. Reward assignment is mechanically verifiable, enabling large-scale search with minimal human oversight.
- **AlphaCode**[22]: Candidate programs are rewarded based on deterministic test outcomes. Verification is cheap and reliable relative to program generation, enabling extensive exploration under explicit acceptance criteria.
- **Lean-lang**[23]: In theorem proving, rewards correspond to verified proof steps or lemmas. Each reward is justified by the proof checker, allowing learning to proceed over mechanically verified progress.

5.8 Context-Epistemic Verification Routing

Summary Context-Epistemic Verification Routing assigns verification strategies and actors based on a joint classification of each case along two dimensions: epistemic clarity and context dependence.

Context AIware systems increasingly generate behaviour in open-ended, dynamic environments, where outputs are shaped not only by learned models but also by runtime context, interaction history, and external world state. In such systems, different cases within the same application may admit radically different notions of correctness and different forms of verification.

Designing a single, uniform verification mechanism for all cases leads either to false certainty or to unscalable human oversight.

Problem Many AI systems implicitly treat verification as a one-dimensional problem: either correctness is assumed to be checkable, or verification is deferred to humans when automation fails. This collapses two fundamentally distinct questions:

- Is there a well-defined notion of correctness or acceptability for this case? (epistemic clarity)
- Is verification invariant, or does it depend on situational, social, or environmental context? (context dependence)

Ignoring context dependence causes systems to over-generalise automated checks beyond their valid scope. Ignoring epistemic clarity causes systems to apply inappropriate correctness criteria or to escalate to humans without a principled rationale. In both cases, verification becomes brittle, ad hoc, and difficult to scale.

The core challenge is how to route verification in a way that respects both the nature of the truth being assessed and the degree to which verification depends on context.

Solution Context-Epistemic Verification Routing introduces an explicit routing layer that classifies each case along two axes:

- **Epistemic clarity**: whether the behaviour admits determinate correctness, probabilistic assessment, or inherently indeterminate or contested evaluation.

[21] `https://people.csail.mit.edu/asolar/SKETCH/`
[22] `https://deepmind.google/blog/competitive-programming-with-alphacode/`
[23] `https://lean-lang.org/`

- **Context dependence**: whether verification criteria are stable and invariant, or require situational, domain-specific, or normative context.

Based on this classification, cases are routed to distinct verification regimes:

- **High epistemic clarity, low context dependence**: fully automated, machine-checkable verification with stackable guarantees.
- **High epistemic clarity, high context dependence**: automated verification augmented with explicit contextual constraints or assumptions.
- **Low epistemic clarity, low context dependence**: statistical, consistency-based, or calibration-oriented verification.
- **Low epistemic clarity, high context dependence**: human verification focused on interpretive judgment, legitimacy, or accountability.

Humans are not treated as a generic fallback. Instead, human verification is deliberately reserved for cases where correctness cannot be established without contextual interpretation, or where verification itself targets human understanding and legitimacy.

Benefits

- **Epistemically grounded verification**: Verification strategies align with what can be meaningfully verified.
- **Scalable automation**: Most cases are handled by automated verification without sacrificing legitimacy.
- **Principled human involvement**: Human effort is focused on high-context, high-impact cases.
- **Clear failure semantics**: Verification failures can be attributed to epistemic limits or contextual gaps rather than system error.

Drawbacks

- **Classification complexity**: Accurately modelling epistemic clarity and context dependence requires domain understanding.
- **Routing risk**: Misclassification may lead to inappropriate verification or misplaced confidence.

Known Uses

- **IDx-DR / LumineticsCore**[24]: The system explicitly distinguishes determinate diagnostic outcomes from context-dependent cases such as insufficient image quality, routing the latter to human clinicians rather than forcing automated decisions.
- **Population-scale medical screening pipelines**[25]: Deployed workflows route cases based on both diagnostic confidence and contextual adequacy, ensuring that verification responsibility matches what can be justified for each case.

5.9 Redundant Solvers

Summary Redundant Solvers deploy multiple independent solvers to generate alternative candidate behaviours for the same task and use their alignment or misalignment as a verification signal, without assuming correctness from any single solver.

Context AIware systems frequently generate complex behaviours such as reasoning traces, plans, decisions, or action sequences under partial specifications and open-ended constraints. In many such settings, explicit verification conditions are incomplete or context-dependent, and a single generated behaviour provides limited evidence of acceptability.

[24] `https://www.accessdata.fda.gov/cdrh_docs/reviews/DEN180001.pdf`
[25] `https://www.thelancet.com/journals/landig/article/PIIS2589-7500(25)00096-2/fulltext`

Problem A single solver may produce a coherent and confident behaviour that nonetheless relies on incorrect assumptions, overlooks constraints, or exploits underspecified objectives. When verification criteria are weak or partial, the absence of an explicit failure signal is often misinterpreted as success.

Conversely, treating agreement across multiple solvers as proof of correctness is unsound: solvers may share training data, inductive biases, or structural blind spots. The challenge is how to use solver redundancy to improve verifiability without collapsing into consensus-based decision making.

Solution The Redundant Solvers pattern introduces two or more independent solvers that generate candidate behaviours for the same task. Independence may be achieved through different prompting strategies, reasoning styles, tool-use policies, or model instances.

A verifier compares solver outputs to assess their degree of alignment with respect to relevant behavioural properties (e.g., conclusions reached, constraints invoked, assumptions made, or actions proposed). Alignment increases confidence that the behaviour is stable under solver variation, while misalignment indicates epistemic or contextual uncertainty.

Rather than selecting an output by voting or averaging, solver comparison is used to inform verification decisions, such as whether additional verification is required, execution should be constrained, or human interpretation is warranted.

Benefits

- **Uncertainty exposure**: Reveals hidden ambiguity that single-solver generation may conceal.
- **Verification support**: Provides an additional signal when explicit verification conditions are incomplete.
- **Solver isolation**: Reduces reliance on any single solver's assumptions or failure modes.

Drawbacks

- **Increased computational cost**: Requires generating and comparing multiple candidate behaviours.
- **Non-decisive**: Solver alignment does not establish correctness; it only informs confidence.
- **Design sensitivity**: Effectiveness depends on the independence and diversity of solvers.

Known Uses

- **SpaceX Dragon and Falcon autonomous flight software** [26]: SpaceX flight systems employ multiple independently developed guidance, navigation, and control (GNC) software paths running in parallel. Disagreement between redundant decision paths triggers fault management modes, conservative control laws, or abort logic rather than automatic execution.
- **Airbus Fly-By-Wire Flight Control Systems** [27]: Airbus aircraft deploy triplex or quadruplex flight control computers developed by independent teams using different software and hardware stacks. Outputs are continuously cross-checked. Misalignment between solvers causes reversion to degraded modes or alternate laws, explicitly treating disagreement as a safety signal rather than consensus correctness.
- **Google Large-Scale Production Ranking and Decision Systems** [28]: In production search, ads, and recommendation systems, Google runs multiple independently trained ranking and decision models in parallel. Disagreement between model outputs is used to detect distribution shifts, uncertainty, or anomalous behaviour, triggering additional evaluation, fallback logic, or human review rather than automatic acceptance.

5.10 Multi-verifiers

Summary Multi-verifiers structures verification as a layered design involving multiple independent verifiers and, where necessary, secondary verification that evaluates the validity of verification findings themselves, rather than relying on a single authority or assuming consensus implies correctness.

[26] `https://www.spacex.com/vehicles/dragon/`
[27] `https://www.airbus.com/en/innovation/zero-emission/fly-by-wire`
[28] `https://blog.google/technology/ai/`

Context In AIware systems, behaviour is generated dynamically by learned or agentic components and evaluated under partial specifications, evolving context, or normative constraints. Verification targets may include correctness, safety, policy compliance, robustness, provenance, or legitimacy. In such settings, no single verifier, whether automated or human, can reliably capture all relevant acceptability conditions.

Moreover, verifiers themselves may be imperfect. Automated verifiers may produce false positives or false negatives due to modelling assumptions, incomplete specifications, or distribution shift. Human verifiers may disagree, misinterpret context, or introduce bias. As a result, verification outcomes themselves become objects that may require further scrutiny.

Problem Single-verifier designs concentrate epistemic authority in one mechanism. This amplifies blind spots, obscures uncertainty, and creates brittle failure modes: when a verifier flags an issue, the system often lacks a principled way to determine whether the issue genuinely exists, whether it is an artefact of the verifier, or whether additional context is required.

Conversely, naïvely aggregating multiple verifiers by voting or consensus is insufficient. Agreement does not imply correctness, and disagreement does not indicate which verifier is wrong or how the conflict should be resolved. The core challenge is how to structure verification so that both behavioural claims and verification findings can be assessed systematically.

Solution Multi-verifiers defines verification as a structured, multi-stage process involving both primary and secondary verification.

At the primary level, multiple independent verifiers evaluate the same behavioural claim. These verifiers may differ in method, scope, assumptions, or verification target, such as logical consistency, policy compliance, robustness, safety constraints, or provenance. Each verifier emits a structured verification signal rather than a binary pass or fail.

At the secondary level, verification focuses on the verification results themselves. Secondary verification assesses whether issues identified by primary verifiers indeed exist, whether they arise from genuine violations, incomplete assumptions, or verifier limitations. This may include:

- checking the applicability and assumptions of a verifier to the current case;
- validating the evidence or reasoning underlying a reported violation;
- resolving conflicts between verifiers by evaluating their respective scopes and epistemic strengths;
- determining whether escalation to additional automated checks or human judgment is warranted.

Interpretation of verification outcomes is therefore an explicit verification activity, not an implicit arbitration policy. Acceptance, rejection, or escalation decisions are made based on structured verification evidence and meta-level assessment of verifier reliability, rather than on majority vote or implicit trust.

Benefits

- **Epistemic robustness:** Reduces reliance on any single verifier and mitigates verifier-specific blind spots.
- **Explicit handling of verifier uncertainty:** False positives, false negatives, and verifier disagreement become first-class signals rather than exceptions.
- **Principled escalation:** Secondary verification provides a structured basis for deciding when additional checks or human involvement are necessary.
- **Scalable assurance:** Most cases are resolved automatically at the primary level, with secondary verification and human effort reserved for contested or high-impact cases.

Drawbacks

- **Design complexity:** Requires explicit modelling of verifier roles, assumptions, and interpretation logic.
- **Increased overhead:** Additional verification stages may introduce computational or organisational cost.
- **Risk of over-verification:** Poorly calibrated secondary verification may lead to unnecessary escalation or latency.

Known Uses

- **YouTube Content Moderation and Appeals**[29]: Content is evaluated by multiple automated classifiers targeting different policy dimensions. Disputed cases are escalated to additional review layers, where moderators assess not only the content but also whether the automated flags are justified, effectively performing secondary verification.
- **Stripe Radar**[30]: Stripe Radar combines multiple fraud detection models and rule-based checks. Elevated risk or disagreement triggers step-up actions such as additional authentication or manual review, where investigators assess whether automated risk signals reflect genuine fraud.
- **Palantir AIP**[31]: AI-generated decisions are evaluated by multiple policy, risk, and data-quality verifiers. Conflicts or anomalies in verification outcomes trigger further evaluation of the verification evidence itself before acceptance or escalation.

6 Related work

Verification of AI systems have been studied across several research communities, including software engineering, formal methods, cyber-physical systems, autonomous systems, and trustworthy AI. A substantial body of research has investigated verification and validation methods for AI-enabled and learning-based systems, particularly in safety-critical and cyber-physical contexts. For example, Calinescu et al. [60] investigate runtime assurance for self-adaptive systems through the concept of dynamic assurance cases, which support continuous verification under evolving operational conditions by allowing assurance arguments to be updated as systems adapt at runtime. This line of work provides important mechanisms for maintaining assurance after deployment, but it largely assumes a given system structure and concentrates on how verification can be performed within that structure. In contrast, our work addresses how verification considerations should shape problem formulation, decomposition, and architectural design from the outset, especially in AIware-centric systems where behaviour is not explicitly encoded in code artefacts.

Another closely related line of research focuses on testing and verification techniques for AI-based and machine-learning systems. Zhang et al. [61] present a comprehensive survey and taxonomy of testing methods for machine-learning systems, covering data testing, model testing, metamorphic testing, and runtime validation. Complementing this, Garousi et al. [62] conduct a classification study on testing and verification of AI-based systems, analysing existing techniques with respect to system type, validation goal, and evaluation context. Industrial perspectives further illustrate the practical challenges of testing AI systems. Breck et al. [63] introduce the ML Test Score, a rubric for assessing production readiness and technical debt in machine-learning systems, while Amershi et al. [64] analyse software engineering practices in large-scale industrial ML deployments at Microsoft. While these studies provide valuable insights into testing and validation practices, they largely treat verification as a post hoc activity applied to models or pipelines. Our work differs by framing verification as a design-time concern that influences decomposition strategies, lifecycle structure, and optimisation objectives.

Runtime monitoring and assurance for autonomous and agentic systems form another important strand of related work. Luckcuck et al. [65] survey specification and verification approaches for autonomous systems, highlighting challenges in specifying adaptive, non-deterministic behaviour and the limitations of static verification techniques. Our framework explicitly distinguishes case-level and system-level verification and situates runtime assurance within a lifecycle-spanning verification strategy, rather than treating it as a standalone mechanism.

7 Conclusion

The emergence of foundation models and agentic AI systems fundamentally reshapes the foundations of software engineering. As behaviour generation becomes cheap, abundant, and increasingly automated, verification rather than construction defines the practical limits of scale, safety, and trust. This paper argues that scalable AI engineering therefore requires a decisive shift from treating verification as an afterthought to adopting a verifiability first design paradigm. We introduce a conceptual framework that characterises verification along five orthogonal dimensions, namely what is verified, how verification is performed, when it occurs across the system lifecycle, who performs verification, and why verification is required. Grounded in a systematic literature review, the framework provides a unifying lens for analysing verification across a wide range of AI systems, including formal code and model verification, learning based optimisation, agentic behaviour, and governance oriented oversight. Building on this framework, we

[29] `https://support.google.com/youtube/answer/2802032`

[30] `https://stripe.com/radar`

[31] `https://www.palantir.com/platforms/aip/`

derive a set of design principles that explain how AI systems should be deliberately structured so that verification remains automated, scalable, and compositional as autonomy increases. We further operationalise these principles through a collection of architectural design patterns that distil recurring verification oriented structures observed in real world systems. Future work will focus on empirical evaluation of the design principles and architectural patterns through their application in large-scale, real-world deployments, particularly in AI-for-science and other high-stakes domains where verification cost, scalability, and long-term assurance are critical.

References

[1] Len Bass, Qinghua Lu, Ingo Weber, and Liming Zhu. *Engineering AI systems: architecture and DevOps essentials*. Addison-Wesley Professional, 2025.

[2] Andrej Karpathy. Verifiability. `https://karpathy.bearblog.dev/verifiability/`, 2024. Accessed: 2025-12-31.

[3] Liming Zhu, Qinghua Lu, Ming Ding, Sung Une Lee, and Chen Wang. Designing meaningful human oversight in ai. *Available at SSRN*, 2025. SSRN Working Paper No. 5501939.

[4] Liming Zhu, Qinghua Lu, Sung Une Lee, and Ding Ming. Oversight design for AI-enabled decision making in government services. *Available at SSRN*, 2025. SSRN Working Paper No. 5543018.

[5] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering—a systematic literature review. *Information and Software Technology*, 51(1):7–15, 2009.

[6] Rishi Bommasani. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[7] Humza Sami, Pierre-Emmanuel Gaillardon, Valerio Tenace, et al. Aivril: Ai-driven rtl generation with verification in-the-loop. *arXiv preprint arXiv:2409.11411*, 2024.

[8] Aman Kumar, Deepak Narayan Gadde, Keerthan Kopparam Radhakrishna, and Djones Lettnin. Saarthi: The first ai formal verification engineer. *arXiv preprint arXiv:2502.16662*, 2025.

[9] Marcos Cramer and Lucian McIntyre. Verifying llm-generated code in the context of software verification with ada/spark. *arXiv preprint arXiv:2502.07728*, 2025.

[10] Mihir Gupte et al. Towards autoformalization of llm-generated outputs for requirement verification. *arXiv preprint arXiv:2511.11829*, 2025.

[11] Zhaoyang Wang, Jinqi Jiang, Huichi Zhou, Wenhao Zheng, Xuchao Zhang, Chetan Bansal, and Huaxiu Yao. Verifiable format control for large language model generations. *arXiv preprint arXiv:2502.04498*, 2025.

[12] Kuo Zhou and Lu Zhang. Step-wise formal verification for llm-based mathematical problem solving. *arXiv preprint arXiv:2505.20869*, 2025.

[13] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. Improving steering and verification in ai-assisted data analysis with interactive task decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–19, 2024.

[14] Roham Koohestani. Agentguard: Runtime verification of ai agents. *arXiv preprint arXiv:2509.23864*, 2025.

[15] Artem Grigor, Christian Schroeder de Witt, and Ivan Martinovic. Proofs of autonomy: Scalable and practical verification of ai autonomy. In *ICML Workshop on Technical AI Governance (TAIG)*.

[16] Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. Verisafe agent: Safeguarding mobile gui agent via logic-based action verification. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, pages 817–831, 2025.

[17] Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T Le. Veriguard: Enhancing llm agent safety via verified code generation. *arXiv preprint arXiv:2510.05156*, 2025.

[18] Saptarshi Sengupta, Harsh Vashistha, Kristal Curtis, Akshay Mallipeddi, Abhinav Mathur, Joseph Ross, and Liang Gou. Mag-v: A multi-agent framework for synthetic data generation and verification. *arXiv preprint arXiv:2412.04494*, 2024.

[19] Tianyang Xu, Dan Zhang, Kushan Mitra, and Estevam Hruschka. Verification-aware planning for multi-agent systems. *arXiv preprint arXiv:2510.17109*, 2025.

[20] Eranga Bandara, Ross Gore, Peter Foytik, Sachin Shetty, Ravi Mukkamala, Abdul Rahman, Xueping Liang, Safdar H Bouk, Amin Hass, Sachini Rajapakse, et al. A practical guide for designing, developing, and deploying production-grade agentic ai workflows. *arXiv preprint arXiv:2512.08769*, 2025.

[21] Sandro Rodriguez Garzon, Awid Vaziry, Enis Mert Kuzu, Dennis Enrique Gehrmann, Buse Varkan, Alexander Gaballa, and Axel Küpper. Ai agents with decentralized identifiers and verifiable credentials. *arXiv preprint arXiv:2511.02841*, 2025.

[22] Zhenhua Zou, Zhuotao Liu, Lepeng Zhao, and Qiuyang Zhan. Blocka2a: Towards secure and verifiable agent-to-agent interoperability. *arXiv preprint arXiv:2508.01332*, 2025.

[23] Artem Grigor, Christian Schroeder de Witt, Simon Birnbach, and Ivan Martinovic. Vet your agent: Towards host-independent autonomy via verifiable execution traces. *arXiv preprint arXiv:2512.15892*, 2025.

[24] Nitin Singh, Pankaj Dayama, and Vinayaka Pandit. Zero knowledge proofs towards verifiable decentralized ai pipelines. In *International Conference on Financial Cryptography and Data Security*, pages 248–275. Springer, 2022.

[25] Francesco Bellotti, Riccardo Berta, Vafali Soltanmuradov, David Martín Gómez, Akshay Dhonthi, Vahid Hashemi, and Luca Lazzaroni. Robustness verification of a reinforcement learning-based agent for automated car parking. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 139–147. Springer, 2024.

[26] Guan-Yan Yang and Farn Wang. Taming silent failures: A framework for verifiable ai reliability. *arXiv preprint arXiv:2510.22224*, 2025.

[27] Mike Loukides. Software 2.0 means verifiable ai. `https://www.oreilly.com/radar/software-2-0-means-verifiable-ai/`, 2025. O'Reilly Radar article.

[28] Emmanuel O Badmus, Peng Sang, Dimitrios Stamoulis, and Amritanshu Pandey. Powerchain: A verifiable agentic ai system for automating distribution grid analyses. *arXiv preprint arXiv:2508.17094*, 2025.

[29] Paolo Morettin, Andrea Passerinia, and Roberto Sebastiania. Towards a unified framework for probabilistic verification of ai systems. In *The Verifying Learning AI Systems (VeriLearn) Workshop*. VeriLearn, 2023.

[30] Cristina Cornelio, Takuya Ito, Ryan Cory-Wright, Sanjeeb Dash, and Lior Horesh. The need for verification in ai-driven scientific discovery. *arXiv preprint arXiv:2509.01398*, 2025.

[31] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. Trainverify: Equivalence-based verification for distributed llm training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 237–253, 2025.

[32] Yudong Wang, Zixuan Fu, Jie Cai, Peijun Tang, Hongya Lyu, Yewei Fang, Zhi Zheng, Jie Zhou, Guoyang Zeng, Chaojun Xiao, et al. Ultra-fineweb: Efficient data filtering and verification for high-quality llm training data. *arXiv preprint arXiv:2505.05427*, 2025.

[33] Clark Barrett, Thomas A Henzinger, and Sanjit A Seshia. Certificates in ai: Learn but verify. *Communications of the ACM*, 2024.

[34] United Nations Scientific Advisory Board. Verification of frontier AI. Technical report, United Nations, June 2025. Accessed: 2026-01-05.

[35] Yedi Zhang, Sun Yi Emma, Annabelle Lee Jia En, and Jin Song Dong. Rvllm: Llm runtime verification with domain knowledge. *arXiv preprint arXiv:2505.18585*, 2025.

[36] Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. Rethinking verification for llm code generation: From generation to testing. *arXiv preprint arXiv:2507.06920*, 2025.

[37] Brando Miranda, Zhanke Zhou, Allen Nie, Elyas Obbad, Leni Aniva, Kai Fronsdal, Weston Kirk, Dilara Soylu, Andrea Yu, Ying Li, et al. Veribench: End-to-end formal verification benchmark for ai code generation in lean 4. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

[38] Maciej Besta, Lorenzo Paleari, Marcin Copik, Robert Gerstenberger, Ales Kubicek, Piotr Nyczyk, Patrick Iff, Eric Schreiber, Tanja Srindran, Tomasz Lehmann, et al. Checkembed: Effective verification of llm solutions to open-ended tasks. *arXiv preprint arXiv:2406.02524*, 2024.

[39] Runtao Zhou, Giang Nguyen, Nikita Kharya, Anh Nguyen, and Chirag Agarwal. Improving human verification of llm reasoning through interactive explanation interfaces. *arXiv preprint arXiv:2510.22922*, 2025.

[40] Run Peng, Ziqiao Ma, Amy Pang, Sikai Li, Zhang Xi-Jia, Yingzhuo Yu, Cristian-Paul Bara, and Joyce Chai. Communication and verification in llm agents towards collaboration under information asymmetry. *arXiv preprint arXiv:2510.25595*, 2025.

[41] Shalev Lifshitz, Sheila A McIlraith, and Yilun Du. Multi-agent verification: Scaling test-time compute with multiple verifiers. *arXiv preprint arXiv:2502.20379*, 2025.

[42] Jan Hendrik Kirchner, Yining Chen, Harri Edwards, Jan Leike, Nat McAleese, and Yuri Burda. Prover-verifier games improve legibility of LLM outputs. *arXiv preprint arXiv:2407.13692*, 2024.

[43] Jiangou Zhan, Wenhui Zhang, Zheng Zhang, Huanran Xue, Yao Zhang, and Ye Wu. Portcullis: A scalable and verifiable privacy gateway for third-party LLM inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 1022–1030, 2025.

[44] Artem Grigor, Christian Schroeder de Witt, and Ivan Martinovic. Proofs of autonomy: Scalable and practical verification of AI autonomy. In *Proceedings of the ICML Workshop on Technical AI Governance (TAIG)*, 2024.

[45] Abhivansh Gupta. Verifiability-first agents: Provable observability and lightweight audit agents for controlling autonomous LLM systems. *arXiv preprint arXiv:2512.17259*, 2025.

[46] Sai Charan Dekkata, Hrishikesh Tawade, Vamsidhar Reddy Kamanuru, Santhosh Kusuma Kumar Parimi, and Surendra Rathod. Strengthening ai validation and verification in autonomous vehicles: Comprehensive analysis and improvements to key standards. In *Proceedings of the 2025 IEEE Conference on Artificial Intelligence (CAI)*, pages 1540–1549. IEEE, 2025.

[47] Przemyslaw Biecek and Wojciech Samek. Model science: Getting serious about verification, explanation and control of AI systems. *arXiv preprint arXiv:2508.20040*, 2025.

[48] Bin Ji, Huijun Liu, Mingzhe Du, Shasha Li, Xiaodong Liu, Jun Ma, Jie Yu, and See-Kiong Ng. Towards verifiable text generation with generative agent. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24230–24238, 2025.

[49] Fanbin Lu, Zhisheng Zhong, Ziqin Wei, Shu Liu, Chi-Wing Fu, and Jiaya Jia. Steve: A step verification pipeline for computer-use agent training. *arXiv preprint arXiv:2503.12532*, 2025.

[50] Yedi Zhang, Yufan Cai, Xinyue Zuo, Xiaokun Luan, Kailong Wang, Zhe Hou, Yifan Zhang, et al. Position: Trustworthy AI agents require the integration of large language models and formal methods. In *Proceedings of the Forty-second International Conference on Machine Learning (ICML) Position Paper Track*, 2025.

[51] Zhihong Shao, Yuxiang Luo, Chengda Lu, Z. Z. Ren, Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and Xiaokang Zhang. Deepseekmath-v2: Towards self-verifiable mathematical reasoning. *arXiv preprint arXiv:2511.22570*, 2025.

[52] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024.

[53] Yoo Yeon Sung, Hannah Kim, and Dan Zhang. Verila: A human-centered evaluation framework for interpretable verification of llm agent failures. *arXiv preprint arXiv:2503.12651*, 2025.

[54] Towards formal verification of llm-generated code from natural language prompts. *arXiv preprint*, 2023.

[55] Miloš Košprdić, Adela Ljajić, Bojana Bašaragin, Darija Medvecki, and Nikola Milošević. Verif.ai: Towards an open-source scientific generative question-answering system with referenced and verifiable answers. *arXiv preprint arXiv:2402.18589*, 2024.

[56] Florian Dörfler, Matthias Görtz, Kilian Kleeberger, Andreas Schuller, and Birgit Vogel-Heuser. Verification and validation of llm-rag for industrial automation. *arXiv preprint arXiv:2407.08988*, 2024.

[57] Zhizheng Wang, Qiao Jin, Chih-Hsuan Wei, Shubo Tian, Po-Ting Lai, Qingqing Zhu, Chi-Ping Day, Christina Ross, Robert Leaman, and Zhiyong Lu. Geneagent: Self-verification language agent for gene-set analysis using domain databases. *Nature Methods*, 22(8):1677–1685, 2025.

[58] Albert Sadowski and Jaroslaw A. Chudziak. On verifiable legal reasoning: A multi-agent framework with formalized knowledge representations. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pages 2535–2545, 2025.

[59] Yusuf Ali, Gryphon Patlin, Karthik Kothuri, Muhammad Zubair Irshad, Wuwei Liang, and Zsolt Kira. Eve: A generator-verifier system for generative policies. *arXiv preprint arXiv:2512.21430*, 2025.

[60] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly. Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11):1039–1069, 2017.

[61] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1):1–36, 2020.

[62] Emanuele De Angelis, Guglielmo De Angelis, and Maurizio Proietti. A classification study on testing and verification of AI-based systems. In *Proceedings of the 2023 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 1–8. IEEE, 2023.

[63] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. The ML test score: A rubric for ML production readiness and technical debt reduction. In *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE, 2017.

[64] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[65] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys*, 52(5):1–41, 2019.