Patent Claims — LUCID

PATENT CLAIMS

System and Method for Iterative Formal Verification of AI-Generated Code Using a Hallucination-Verification Loop

---

INDEPENDENT CLAIMS

Claim 1 (Method — Core Iterative Loop)

A computer-implemented method for improving correctness of source code generated by an artificial intelligence model, the method comprising:

(a) receiving, by one or more processors, source code generated by a generative language model in response to a task description;

(b) extracting, by a claim extraction module executed by the one or more processors, a plurality of testable claims from the generated source code, each testable claim representing a decidable predicate about an expected behavior of the generated source code;

(c) verifying, by a formal verification engine executed by the one or more processors, each of the plurality of testable claims by executing the generated source code against a test specification in a sandboxed execution environment, the formal verification engine producing a verification result for each claim, the verification result comprising a verdict selected from the group consisting of: PASS, PARTIAL, FAIL, and NOT APPLICABLE;

(d) computing, by the one or more processors, a specification gap metric based on the verification results, the specification gap metric quantifying a fraction of applicable claims that did not receive a PASS verdict;

(e) when the specification gap metric exceeds a convergence threshold, generating, by a remediation module executed by the one or more processors, a structured remediation plan comprising, for each claim with a FAIL or PARTIAL verdict: an identification of the specific test failure, error messages and stack traces from execution, a diagnosed root cause, and a proposed code modification;

(f) regenerating, by the generative language model, updated source code incorporating the structured remediation plan while preserving portions of the source code associated with claims having a PASS verdict; and

(g) repeating steps (b) through (f) with the updated source code for a predetermined maximum number of iterations or until the specification gap metric falls below the convergence threshold.

Claim 2 (System — API-Based Verification Service)

A system for verifying and improving source code generated by artificial intelligence models, the system comprising:

one or more processors;

a non-transitory computer-readable storage medium storing instructions that, when executed by the one or more processors, cause the system to:

expose an application programming interface (API) configured to receive code verification requests and return verified code responses;

implement a generator module configured to interface with one or more large language models to generate source code from task descriptions, the generator module being model-agnostic and compatible with a plurality of different large language models;

implement a claim extraction module configured to decompose generated source code into a plurality of testable claims, each claim being a decidable predicate about an expected behavior of the source code;

implement a formal verification engine configured to execute the generated source code against test specifications in a sandboxed execution environment, the formal verification engine producing a structured verification result for each claim comprising a verdict and, for failing claims, error messages, stack traces, and execution diagnostics;

implement a remediation module configured to generate structured remediation plans from verification failures; and

implement an iterative loop controller configured to orchestrate repeated cycles of claim extraction, formal verification, remediation, and regeneration until a convergence criterion is satisfied or a maximum iteration count is reached.

Claim 3 (Method — Specification-Level Variant)

A computer-implemented method for reducing a specification gap between an AI-generated specification and an actual codebase, the method comprising:

(a) receiving, by one or more processors, an AI-generated specification comprising a plurality of claims about expected properties of a software system, wherein the specification is generated by prompting a large language model to hallucinate a comprehensive description of a software system;

(b) extracting, by a claim extraction module, individual testable claims from the AI-generated specification, each claim being categorized by type and assigned a severity level;

(c) verifying, by a formal verification engine, each testable claim against an actual codebase state through one or more of: static analysis, test suite execution, bounded model checking, type checking, or API conformance checking;

(d) computing a specification gap metric as one minus the ratio of passing applicable claims to total applicable claims;

(e) generating a structured remediation plan that identifies which verification failures to address, prioritized by claim severity and feasibility;

(f) applying code modifications to the codebase according to the remediation plan;

(g) regenerating an updated specification conditioned on the modified codebase; and

(h) repeating steps (b) through (g) until the specification gap metric converges below a target threshold.

DEPENDENT CLAIMS — METHOD (depending from Claim 1)

Claim 4 (Monotonic Convergence Property)

The method of Claim 1, wherein the iterative repetition of steps (b) through (f) produces a sequence of specification gap metrics that is monotonically non-increasing when the generated source code comprises isolated functions without cross-dependencies between claims, such that no task that receives a PASS verdict at iteration k loses the PASS verdict at any subsequent iteration k' > k.

Claim 5 (Formal Test Generation from LLM Output)

The method of Claim 1, wherein the test specification against which the generated source code is verified is itself generated by the method, the test generation comprising:

(a) receiving the plurality of testable claims extracted in step (b);

(b) translating each testable claim into one or more executable test cases using a language model operating at a deterministic temperature setting;

(c) validating the generated test cases for syntactic correctness and executability; and

(d) executing the generated source code against the generated test cases.

Claim 6 (Multi-Iteration Improvement with Context Preservation)

The method of Claim 1, wherein the regeneration in step (f) preserves context from prior iterations by providing to the generative language model: the original task description, the current source code, the verification results from the current iteration, the structured remediation plan, and the source code from one or more prior iterations, thereby enabling the generative language model to make targeted modifications rather than generating code from scratch.

Claim 7 (Early Exit on Full Convergence)

The method of Claim 1, further comprising:

after step (c), determining whether all applicable claims received a PASS verdict; and

when all applicable claims received a PASS verdict, returning the current source code without performing steps (e) through (g), thereby conserving computational resources for tasks that are resolved before reaching the maximum iteration count.

Claim 8 (Precision-Weighted Specification Gap)

The method of Claim 1, wherein computing the specification gap metric in step (d) further comprises:

assigning a precision weight to each claim based on its severity level, wherein critical claims receive a higher precision weight than low-severity claims; and

computing a precision-weighted specification gap as a weighted sum of verification failures divided by a sum of precision weights of all applicable claims;

whereby critical verification failures contribute more to the specification gap metric and are prioritized in the remediation plan of step (e).

Claim 9 (Sandboxed Execution with Resource Limits)

The method of Claim 1, wherein the sandboxed execution environment of step (c) provides:

process isolation, wherein the generated source code executes in a separate process with restricted permissions;

network isolation, wherein the generated source code cannot access external networks;

filesystem isolation, wherein the generated source code can only access a designated temporary directory;

resource quotas comprising CPU time limits, memory limits, and disk quotas; and

timeout enforcement, wherein test execution is terminated after a configurable timeout period.

Claim 10 (Containerized Verification for Complex Tasks)

The method of Claim 9, wherein for tasks involving modification of an existing codebase, the sandboxed execution environment comprises a containerized environment that replicates the target repository's dependency configuration and continuous integration environment, enabling verification against the exact runtime environment of the target software.

Claim 11 (Remediation Using Formal Verification Feedback)

The method of Claim 1, wherein the structured remediation plan of step (e) is generated by providing to the remediation module:

the specific test names that produced FAIL verdicts;

the complete error messages and stack traces from each failing test;

the relevant source code sections identified from the stack traces; and

the original task description;

and wherein the remediation module produces a structured output comprising, for each failure: a root cause diagnosis, a specific code modification proposal, and a confidence assessment of whether the proposed modification will resolve the failure without introducing regressions.

Claim 12 (Geometric Convergence Rate)

The method of Claim 1, wherein under conditions that remediation is non-expansive, strictly contractive on failures with contraction rate gamma, and that regeneration introduces fewer new failing claims than are fixed (with novelty bound beta where gamma + beta < 1), the specification gap metric converges geometrically according to:

```
specification_gap(t) <= (gamma + beta)^t * specification_gap(0)
```

Claim 13 (Claim Categorization and Hierarchical Verification)

The method of Claim 1, wherein the plurality of testable claims extracted in step (b) are organized into a hierarchical structure comprising:

Level 1 claims directed to individual function correctness;

Level 2 claims directed to API contract conformance;

Level 3 claims directed to feature completeness;

Level 4 claims directed to security and privacy guarantees; and

Level 5 claims directed to legal and regulatory compliance;

wherein higher-level claims depend on lower-level claims, and verification failure at a lower level propagates to constrain the maximum achievable pass rate at higher levels.

Claim 14 (Cost-Efficient Verification)

The method of Claim 1, wherein the formal verification step of step (c) has a computational cost that is independent of the parameter count of the generative language model, such that a formal verification invocation for source code generated by a model with N billion parameters has the same computational cost as a formal verification invocation for source code generated by a model with M billion parameters, where N is not equal to M.

DEPENDENT CLAIMS — SYSTEM (depending from Claim 2)

Claim 15 (Usage-Based API Billing)

The system of Claim 2, further comprising:

a metering module configured to track the number of verification requests processed;

a billing module configured to charge API consumers on a per-verification-request basis, with pricing tiers based on request volume; and

a rate limiting module configured to enforce per-consumer rate limits.

Claim 16 (Multi-Language Support)

The system of Claim 2, wherein the formal verification engine is configured to support source code written in a plurality of programming languages, the formal verification engine selecting an appropriate sandboxed execution environment and test framework based on the programming language of the submitted source code.

Claim 17 (Platform Integration Modes)

The system of Claim 2, wherein the API is configured to support a plurality of integration modes comprising:

an inline verification mode wherein the API receives generated source code and returns verified source code synchronously;

a background verification mode wherein the API receives generated source code, initiates verification asynchronously, and delivers results via a callback or polling mechanism; and

a CI/CD integration mode wherein the API is invoked as part of a continuous integration pipeline to verify source code before it is merged into a codebase.

Claim 18 (Verification Audit Trail)

The system of Claim 2, further comprising:

a logging module configured to generate an audit trail for each verification request, the audit trail comprising: the original generated source code, each extracted claim, each verification result with associated error messages, each remediation plan, and the final verified source code;

whereby the audit trail provides documentary evidence of the verification process for regulatory compliance purposes.

DEPENDENT CLAIMS — SPECIFICATION-LEVEL (depending from Claim 3)

Claim 19 (Hallucination-as-Feature Specification Discovery)

The method of Claim 3, wherein the AI-generated specification of step (a) is deliberately generated by prompting the large language model to describe a comprehensive specification for the software system without constraining the model to only describe existing functionality, such that the resulting specification includes claims about functionality that does not yet exist in the codebase, and wherein verification failures for such claims function as an automated requirements discovery mechanism that identifies missing functionality.

Claim 20 (Convergence Monitoring and Reporting)

The method of Claim 3, further comprising:

recording the specification gap metric at each iteration;

computing a contraction ratio as the ratio of successive specification gap metrics;

detecting convergence stall when the contraction ratio exceeds a threshold for a predetermined number of consecutive iterations; and

generating a convergence report comprising the per-iteration specification gap metrics, the contraction ratios, the residual failing claims, and a classification of residual failures as either remediable or structurally infeasible.

CLAIM DEPENDENCY TREE

```
Claim 1 (Independent - Method)
  |-- Claim 4  (Monotonic convergence)
  |-- Claim 5  (Formal test generation)
  |-- Claim 6  (Context preservation)
  |-- Claim 7  (Early exit)
```

```
  |-- Claim 8  (Precision weighting)
  |-- Claim 9  (Sandbox)
  |     |-- Claim 10 (Containerized verification)
  |-- Claim 11 (Remediation from formal feedback)
  |-- Claim 12 (Geometric convergence rate)
  |-- Claim 13 (Hierarchical claims)
  |-- Claim 14 (Cost-efficient verification)

Claim 2 (Independent - System)
  |-- Claim 15 (Usage-based billing)
  |-- Claim 16 (Multi-language)
  |-- Claim 17 (Integration modes)
  |-- Claim 18 (Audit trail)

Claim 3 (Independent - Specification-Level Method)
  |-- Claim 19 (Hallucination-as-feature)
  |-- Claim 20 (Convergence monitoring)
```

NOTES FOR PATENT ATTORNEY

1. **Prior Art Differentiation.** The key differentiator from all prior art is the use of *formal (execution-based) verification* in the iterative loop, as opposed to learned verification (LLM-as-judge, reward models, self-critique). The empirical evidence shows that only formal verification produces monotonic convergence; all learned verification approaches plateau or regress.

2. **Broadest Protection.** Claim 1 is drafted broadly to cover any iterative loop using formal (non-learned) verification of AI-generated code. The specific implementation details (claim extraction format, remediation plan structure, sandbox technology) are in dependent claims.

3. **The Convergence Property.** Claims 4 and 12 protect the monotonic convergence property itself — the fact that under stated conditions, the specification gap never increases. This is the most defensible and valuable claim because competitors cannot achieve this property without formal verification.

4. **API Delivery.** Claims 2, 15-18 protect the commercial delivery mechanism: an API-based verification service for AI coding platforms. This covers the business model, not just the algorithm.

5. **Specification-Level Variant.** Claims 3, 19-20 protect the broader application where LUCID operates on entire codebases against hallucinated specifications, not just individual function generation. This is the enterprise play.

6. **Hallucination-as-Feature.** Claim 19 specifically protects the concept of deliberately using hallucination to discover missing requirements — a novel application that inverts the conventional framing of hallucination as a defect.

7. **Total: 20 claims** (3 independent + 17 dependent). The independent claims cover the core method, the system/API delivery, and the specification-level variant. Dependent claims cover convergence properties, specific technical features, and commercial embodiments.