WHITE PAPER · LYRIKS

Verifying AI-written code: a deterministic approach

Coding assistants have changed how fast software gets written. This paper looks at how teams can check that what ships is still what they decided, release after release.



CONTENTS

Executive summary 00

1. The verification gap 01

2. What code review checks, and what it cannot 02

3. The reference problem 03

4. Three gaps: additions, omissions, regressions 04

5. Why determinism matters 05

6. How a verification loop works 06

7. Where it fits in the pipeline 07

8. Governance and sovereignty 08

9. How to start 09

10. Glossary 10

Executive summary

- AI coding assistants have sharply increased the volume of code that reaches review. Published industry data shows review time rising alongside output, and developer trust in AI-generated code is low.
- Code review, human or automated, reads the code. It can judge how code is written. It cannot tell whether the code does what the business decided, because that decision lives outside the code.
- The missing reference is the spec: business rules, journeys and invariants, kept in a structured form and linked to the code that implements them. Measured against it, every change can drift in three ways: an addition, an omission or a regression.
- Verification that gates a release has to be deterministic. Same code, same spec, same verdict, through a computation you can replay and audit. A probabilistic check cannot carry that weight on its own.
- A practical loop is available today. The validated spec goes to the coding agent, a deterministic engine checks the result rule by rule, each gap goes back to the agent, and every release ships with an implementation report. Teams can start with a pilot on one or two sensitive modules.

1. The verification gap

Most engineering leaders have felt the shift without needing a survey. Pull requests arrive faster, they are larger, and a growing share of each one was written by an assistant. The surveys confirm the impression and put numbers on it.

SOURCE	SAMPLE	FINDING
Faros AI, The AI Productivity Paradox, July 2025	10,000 developers, 1,255 teams	Teams using AI merged 98% more pull requests. Time spent in review rose 91%.
Sonar, State of Code Developer Survey, January 2026	More than 1,100 developers	96% don't fully trust that AI-generated code is functionally correct. Only 48% always check it before committing.
Stack Overflow Developer Survey 2025	49,000 respondents	29% trust the accuracy of AI output, down from 40% in previous years.
GitHub, Octoverse 2025	Platform activity	Close to one billion commits in 2025, up 25%.

Read together, these figures describe a gap rather than a crisis. Merged pull requests nearly doubled in the teams Faros AI studied. The capacity to judge them did not. Developers are aware of it: nearly all of them say they don't fully trust the code, and about half say they always check it before it goes in. The rest do not always check.

None of this points to careless engineers. It points to a queue. When the number of changes grows faster than the number of people who can evaluate them, review gets thinner. Reviewers skim. They approve what looks reasonable. Over time, teams also become reluctant to touch existing code at all, because nobody is sure what a change might break. They add around it instead, and the system grows while the team's grip on it loosens.

Adding reviewers helps at the margin. It does not change the shape of the curve. The more useful question is what, exactly, a reviewer is being asked to verify, and whether that question can be answered in a different way.

---

2. What code review checks, and what it cannot

Code review is one of the most valuable practices in software engineering, and AI has made it faster. Automated reviewers comment on every pull request within minutes. Human reviewers bring context, taste and knowledge of the team's conventions. Both should stay.

It helps to be precise about the question review answers. A reviewer looks at a diff and asks whether the code is well written and whether it is likely to work. In practice, review is good at catching:

- ☐ style and naming problems, dead code, unclear structure;
- ☐ likely bugs such as off-by-one errors, unhandled exceptions, null dereferences;
- ☐ risky patterns and known classes of vulnerability;
- ☐ inconsistencies with the conventions of the codebase.

All of these are properties of the code itself. A reviewer can detect them by reading the diff, sometimes with the surrounding files.

What review cannot do reliably is answer a different question: does this code do what we decided? Suppose a pricing rule says that a downgrade takes effect at renewal. If the diff applies it immediately, the code may still be clean, tested and idiomatic. Nothing in the diff says it is wrong, because the rule it violates is not in the diff. A careful human reviewer who happens to remember the rule might catch it. An automated reviewer that has never read the pricing policy has no way to.

This is a structural limit, and it applies equally to people and to models. Line-by-line review compares code with itself and with general good practice. Business intent is a separate input. When that input is missing, the best reviewer in the world is checking the wrong thing very carefully.

> **Two questions, two layers.** "Is this code well written?" is answered by reading the code. "Does this code do what was decided?" needs a reference outside the code. A complete verification strategy answers both.

3. The reference problem

If the intent is not in the code, where is it? In most organizations it is scattered. Some of it sits in tickets, some in wiki pages, some in design files, some in the memory of the two people who were in the meeting. Parts of it are only recoverable by reading the code and inferring what someone meant.

That scatter has a cost. A refund window might be 14 days in a ticket, 30 days in the wiki and 21 days in the code (an illustrative example, but a familiar pattern). Which one is right? The code is what runs, so teams tend to treat it as the truth. That instinct is understandable and, for verification purposes, backwards. The code is the source of the implementation. The source of truth is what you meant to build.

To verify anything, you need that intent written down in a form a machine can check against. We call it the spec. A useful spec for verification has four properties.

PROPERTY	WHAT IT MEANS IN PRACTICE
Structured	Features, user journeys, actions, business rules, screens, objects and attributes, invariants and testable requirements are separate, named elements. A rule is a rule, not a paragraph somewhere in a long document.
Linked to code	Each rule points to the code that implements it. "A customer is refunded within 30 days" points to `refund.py`, the payment API and the `refunds` table.
Linked to sources	Each element keeps a link to where it came from: the code, a ticket, a wiki page, a design file. Contradictions between sources are flagged rather than silently resolved.
Validated	The business owner has read and approved it. Verification against an unvalidated spec only proves consistency with a guess.

Most teams do not start with such a spec, especially on existing systems. That is where retro-specification comes in. The spec is reconstructed from the existing code and from the tools around it. It covers the behavior and the business rules, and also the technical side: architecture, modules, dependencies, data model, APIs, flows, integrations. Each rule is linked to the code that implements it. The result is reviewed with the business, corrected where the code turned out to be wrong, and becomes the reference.

A general-purpose language model can also produce documentation from a codebase, often a long set of Markdown pages. That output is readable, which is useful. It is not structured, not linked rule by rule to the code, and not checkable. The difference matters as soon as you want to verify a change against it.

4. Three gaps: additions, omissions, regressions

Once a spec exists, every change can be compared with it. In our experience, the ways delivered code drifts from intended behavior fall into three categories. Naming them is useful because each one needs a different kind of attention.

To make them concrete, take one domain. The example below is **illustrative**: a subscription billing module with four validated rules.

RULE	STATEMENT
R1	A new subscriber gets a 14-day free trial. No charge is made before the trial ends.
R2	When a renewal payment fails, the system retries three times over seven days before suspending the account.
R3	A customer who cancels keeps access until the end of the paid period. No partial refund is issued.
R4	Every invoice shows the VAT rate that applies to the customer's billing country.

A developer asks a coding assistant to add a quarterly billing option. The assistant reads the billing service, adds the new period, adjusts invoice generation and consolidates some duplicated logic along the way. The tests pass. The demo works.

Addition: code nobody asked for

> While adding quarterly plans, the assistant also introduced a 5% discount on quarterly subscriptions, reasoning that longer commitments usually come with a lower price. It added a "save 5%" label on the plan selection screen to match.

Additions are the hardest gap to object to, because they often look like help. A sensible default, an extra case handled just in case, a friendlier label. The problem is ownership. Nobody in product approved the discount. Finance does not know it exists. Nobody will think to test it at the next pricing change. An addition is a behavior that no one decided and no one owns. Spotting it requires the list of what was requested; reading what was written is not enough.

Omission: a rule that was planned and never implemented

> The new quarterly invoice template was built from a simplified layout. It prints the total including tax but no longer shows the applicable VAT rate. Rule R4 is not implemented for quarterly invoices.

Omissions are silent. Nothing breaks on screen. No test fails, because the tests for the new template were written from the same incomplete understanding as the template. The gap surfaces later, during an audit or a customer complaint. Spotting an omission means walking the list of rules that should apply and checking each one, which is tedious for a person and trivial for a machine that holds the list.

Regression: something that worked and no longer does

> While consolidating the payment code, the assistant merged the retry logic for monthly and quarterly renewals into one function with a single retry attempt. Monthly subscribers whose card fails are now suspended after one try instead of three over seven days. Rule R2, which held before the change, no longer holds.

Regressions are the gap teams fear most, and with reason. The behavior was correct, customers relied on it, and a change in an unrelated area broke it. Existing tests catch some regressions, but only the ones someone thought to write a test for. A rule-by-rule comparison with the spec catches the rest, provided each rule is linked to the code that implements it.

Rules R1 and R3 were untouched and still hold. In an implementation report, they would appear as conformant. The full picture for this change is therefore two conformant rules, one addition, one omission and one regression. That picture is far more useful to a release manager than "tests pass".

GAP	DEFINITION	WHY REVIEW MISSES IT	WHAT CATCHES IT
Addition	Behavior present in the code that the spec does not ask for	It looks like a reasonable improvement	Comparing implemented behavior with the list of requested rules
Omission	A rule in the spec that the code does not implement	Missing code does not appear in a diff	Walking every applicable rule and checking each one
Regression	A rule that held before the change and no longer does	The cause is often far from the visible change	Re-checking every rule linked to the touched code, not only the new ones

5. Why determinism matters

A natural response to the verification gap is to add another model. One assistant writes the code, a second one reviews it. That layer is useful for the questions in section 2. It has a limit when it becomes the gate for a release.

Language models are probabilistic. Given a context, they produce a likely output. That is precisely what makes them good at writing code: they fill gaps, interpret vague requests and propose alternatives. It also means the same question can receive different answers on different runs, and a likely answer is not always a correct one. Asking a probabilistic system to certify the output of another probabilistic system stacks one uncertainty on another.

A deterministic engine behaves differently. Given the same inputs, it produces the same result, through a computation that can be replayed step by step. When information is missing, it reports that information is missing rather than filling the gap with a plausible guess.

Three properties follow, and each one matters to a different reader of this paper.

Same question, same answer

If a check can pass on Tuesday and fail on Wednesday on unchanged code, teams learn to rerun it until it turns green. The check turns into a ritual. With a deterministic check, a gap stays open until someone changes the code or the spec. When it closes, there is a reason, and the reason is visible.

Auditability

For a CISO or a compliance function, "the model said it was fine" is not evidence. A deterministic verdict can be traced: which rule was checked, against which code, with which result. It can be re-run by an auditor months later and give the same answer. That is the difference between an opinion and a record.

A release policy you can enforce

Engineering leaders can only enforce a policy that behaves consistently. "No release ships with an open regression on a payment rule" is a sensible policy. It is only enforceable if the check that detects regressions is stable. A deterministic gate can block a release automatically without inviting the question of whether the gate was simply unlucky that day.

> **How the engine reaches a verdict.** The software and its spec are represented as a graph: features, journeys, rules, screens, objects and invariants, each node linked to the code that implements it and to the source it came from. Verifying a change means applying transformation rules to that graph and computing, for each spec rule, whether it still holds. The approach is based on graph rewriting and built in-house. A patent is pending. Formal mathematical proof of conformance is a research direction, not a current capability; today's verdicts are deterministic and replayable, which is the property a release gate needs.

Determinism does not mean AI is absent from the process. Models remain excellent at producing code, drafting specs and explaining results. The point is narrower. The control step, the one that says "conformant" or "not conformant", should not depend on a probability.

---

6. How a verification loop works

Verification is most useful when it is part of the production loop, not an inspection at the end. The loop below is available today with Lyriks and works with the coding agents teams already use.

STEP	WHAT HAPPENS	WHO ACTS	OUTPUT
1. Validate the spec	The rules affected by the change are reviewed and approved by the business owner. For an existing system, they come from retro-specification.	Product owner, tech lead	A validated set of rules, each linked to its sources
2. Send the spec to the agent	The validated rules, with their sources, are exposed to the coding agent (Claude Code, Cursor, Copilot) through MCP. The agent works from explicit rules rather than from a prompt paraphrase.	Developer and coding agent	Proposed implementation
3. Check rule by rule	The deterministic engine compares the delivered code with each rule and classifies it as conformant, addition, omission or regression.	Verification engine	A list of gaps, each tied to a rule and a location in the code
4. Send gaps back	Each gap returns to the agent as a precise, named problem. The agent corrects, and the engine checks again.	Coding agent, engine	A smaller list of gaps on each pass
5. Decide on additions	Additions are not automatically errors. A person decides whether to remove them or to add them to the spec.	Product owner	An updated spec or a cleaner implementation
6. Publish the implementation report	When all rules hold, each release ships with a report listing every rule and its status.	Release manager	A implementation report attached to the release

Two details make this loop practical.

First, gaps are actionable. "Rule R2 no longer holds: renewal retries reduced from three to one in the shared payment function" is something an agent or a developer can fix in minutes. A vague warning that something might be wrong costs far more time to investigate.

Second, the loop runs until all rules hold. It does not stop at "mostly fine". Because the check is deterministic, reaching full conformance is a stable state, not a lucky run.

The same loop extends upstream. Before any code is written, a proposed change can be simulated against the spec: which features, rules, screens and data it touches, what remains compatible, what conflicts, and what needs a decision, including between several changes in progress at the same time. The impact report also lists what is not touched and what needs testing. Many gaps are cheaper to prevent at this stage than to catch after implementation.

7. Where it fits in the pipeline

Spec verification does not replace anything a mature team already does. It adds a layer that looks at a different question. The table summarizes how the layers divide the work.

LAYER	READS	ANSWERS	TYPICAL FINDINGS
Human code review	The diff and its context	Is this change sensible, maintainable and consistent with how we work?	Design concerns, unclear code, convention issues
AI code review	The diff, line by line	Is this code well written and likely to work?	Style, likely bugs, risky patterns, known vulnerabilities
Automated tests	The running code against written test cases	Do the cases we thought of still pass?	Failures on covered paths
Static analysis and security scanning	The code and its dependencies	Does the code respect known quality and security rules?	Code smells, vulnerable dependencies, insecure patterns
Spec verification	The code against the validated spec	Does the code do what was decided?	Additions, omissions, regressions, each tied to a named rule

The findings rarely overlap. An AI reviewer that flags a currency amount stored as a float has done useful work that spec verification would not do. Spec verification that flags a downgrade applied too early has done useful work that the reviewer could not do, because the rule was never in the diff.

Tests deserve a specific note. They are the closest existing practice to spec verification, and good test suites encode a lot of business intent. Their limit is coverage of intent: a test exists only if someone thought to write it, and tests generated alongside the code often share its misunderstandings. A spec-linked check starts from the full list of rules rather than from the cases someone remembered. It also tells you, after a change, which rules need new tests.

> **In practice.** On each pull request, the AI reviewer comments on the diff and the verification engine posts its rule-by-rule status. Developers fix both lists. At release time, the implementation report joins the test results and the security scan in the release record.

8. Governance and sovereignty

A verification layer sees the most sensitive material an organization has: its code, its business rules and the documents that explain them. For CTOs and CISOs, how it is deployed matters as much as what it does.

CONCERN	ANSWER
Can we inspect the code?	Yes. The software is open source under the AGPL license.
Where does it run?	On your infrastructure. It installs in one command, on-premise, including in air-gapped environments, with no outbound connection.
Which model sees our code?	Yours. You bring your own LLM and your own tokens. A local or sovereign model is possible. The verification step itself is deterministic: same code, same spec, same verdict.
How is knowledge stored?	In a graph database, not a retrieval index over text chunks. Every element is linked to its source: code, Jira, Confluence, Notion, Figma or GitHub. Inconsistencies between sources are flagged.
What if we change model or vendor?	The project memory is independent of the LLM and stays with you. It can be exported in full, with its traceability. Models come and go; the memory remains.
Where does the method come from?	A deterministic engine based on graph rewriting, built in-house. Patent pending. Lyriks owns 100% of its IP.

Two governance benefits are worth stating directly. First, the implementation report becomes part of the release record. It shows, for each release, which rules were checked and with what result, in a form that can be re-run later. Second, the spec itself becomes an asset that outlives individual tools and individual team members. When the people who remember why a rule exists move on, the rule, its sources and the code that implements it are still linked.

9. How to start

Verification does not need a big-bang rollout. The most reliable way to evaluate it is a short pilot on a small, well-chosen scope. Start small.

Choose the scope

Pick one or two modules where a wrong rule is expensive and AI-assisted changes are frequent. Billing, pricing, eligibility, permissions and regulatory reporting are common candidates. A module that

people hesitate to touch is often a good choice, because that hesitation is exactly what the pilot should address.

Run the pilot

- [ ] Retro-specify the chosen modules and have the business owner validate the resulting rules.
- [ ] Connect the spec to the coding agent your team already uses, through MCP.
- [ ] Run the verification loop on real changes for a few weeks, alongside your existing review and tests.
- [ ] Attach an implementation report to each release that touches the pilot modules.

Measure what changes

Decide on your measures before you start, and record a baseline. The following are practical and need no special tooling.

MEASURE	WHY IT MATTERS
Rules validated in the pilot scope	Shows how much intent is now explicit and checkable
Contradictions found between sources during retro-specification	Shows how much ambiguity existed before any code changed
Gaps found per change, by type (addition, omission, regression)	Shows what review and tests were letting through
Gaps found by verification that review and tests did not flag	Isolates the value of the additional layer
Passes needed to reach full conformance	Shows how well the agent works from explicit rules
Reviewer time spent on business-rule questions	Shows whether review can refocus on design and code quality
Incidents traced to a business rule in the pilot modules	The outcome that matters most, measured over a longer period

At the end of the pilot, the decision is usually clear. Either the verification layer found gaps that nothing else caught, and it is worth extending to more modules, or it did not, and you have learned that your existing process covers the scope well. Both outcomes are useful.

10. Glossary

TERM	DEFINITION
Spec	The structured description of what the software should do: features, journeys, actions, business rules, screens, objects, invariants and testable requirements.
Business rule	A single, named statement of intended behavior, such as "a customer is refunded within 30 days".
Invariant	A condition that must always hold, whatever the sequence of actions.
Retro-specification	Reconstructing the spec from existing code and related sources: behavior and business rules, and also architecture, data and integrations, each element linked to its code.
Conformant	A rule that the delivered code implements as specified.
Addition	Behavior present in the code that the spec does not ask for.
Omission	A rule in the spec that the code does not implement.
Regression	A rule that held before a change and no longer holds after it.
Implementation report	The rule-by-rule status of a release against the spec.
Deterministic	Producing the same result from the same inputs, through a computation that can be replayed.
Probabilistic	Producing a likely result, which may differ between runs on the same input.
Graph rewriting	A family of techniques that transform a graph by applying rules to its nodes and edges. Here, the basis of the verification engine.
MCP	Model Context Protocol, an open protocol through which coding agents read external context such as a spec.
Air-gapped	Deployed on a network with no connection to the outside.

See the verification loop on real code

Lyriks checks AI-written code against the rules you validated, with a deterministic engine and an implementation report for each release. It runs next to your review and your tests, on your infrastructure. Learn more at lyriks.io/use-cases/verify-ai-code or book a walkthrough at lyriks.io/demo.