# P vs NP is Unprovable: A Formal Demonstration that the Problem is a Paradox

John Augustine McCain
Independent Researcher
trevalence@myyahoo.com

August 2025

**Abstract**

This paper formally demonstrates that the P vs NP problem is not merely unsolved, but unprovable, because it is a paradox: a structurally self-referential problem that cannot be resolved within the formal system in which it is posed. We show that P vs NP depends on a semantic collapse between "verification" and "proof," while defining both purely syntactically. This creates a tautological structure—solvable if and only if a solution is verifiable—mirroring the structure of the Halting Problem. In both cases, decision is only meaningful after the fact: a program halts when it halts; a certificate is verifiable when it is presented. As such, P vs NP becomes undecidable by construction.

We argue that this paradox arises from a category error: treating semantic reasoning as equivalent to syntactic computation. The assumed distinction between solving and verifying presumes a stable context in which verification can occur independently of discovery. But in reality, the structure of a problem is co-defined with its solution—there is no verification without an epistemically coherent question. Thus, P vs NP cannot be resolved through increased mathematical technique; it must be reframed entirely.

Our conclusion is constructive: P vs NP is not a failed question, but a fruitful paradox. Its resistance reveals the boundary of formal reasoning, and points toward models of computation that treat problems, solutions, and the contexts they generate as interdependent. This reframing does not solve the paradox; it proves it cannot be solved.

## 1 Introduction

In 1936, Alan Turing introduced the concept of a universal computing machine, formalizing the notion of mechanical computation. From this foundation, he identified a boundary to computation itself: the *Halting Problem*. Turing proved that there is no general algorithm capable of determining, for every possible program and input, whether the program will eventually halt or run forever. This was not a practical limitation, but a structural one—an impossibility built into the very definition of computation.

The Halting Problem is often described as a proof of undecidability. But it is also, more fundamentally, a paradox. The question "Will this program halt?" can only be answered by running the program and observing whether it halts. Any attempt to decide the outcome without running it becomes entangled in self-reference: a program that attempts to predict its own halting can be constructed to falsify its own prediction. The result is an infinite regress—a logical knot that cannot be untangled from within the system that poses it.

This paradoxical nature is not unique to the Halting Problem. It reflects a deeper pattern in formal systems, echoed in Gödel's incompleteness theorems, where certain truths cannot be proven within the system that defines them. The question of whether $P = NP$ belongs to this same family of problems. Despite over fifty years of intense research, no proof of either $P = NP$ or $P \neq NP$ has been found. It is widely suspected—by many of the field's most respected theorists—that the question itself may be undecidable within classical complexity theory.

Richard Lipton, Scott Aaronson, and others have outlined philosophical arguments suggesting that P vs NP could be independent of standard mathematical axioms, much like the continuum hypothesis. More speculative voices, such as Gregory Chaitin, have suggested that certain mathematical truths may be "true for no reason," existing beyond proof. These intuitions circle around the same insight: P vs NP might not just be unsolved, but unprovable.

The essence of P vs NP is this: can every problem whose solution can be *verified* efficiently also be *solved* efficiently? At first glance, this seems like a clear question. But as we will show, the distinction between "verification" and "solving" is itself semantically unstable. Verification only has meaning once a solution (or *certificate*) is already known. The act of verification does not precede the solution; it *coexists* with it. In this way, P vs NP is structurally akin to asking, "Can we know the answer before we know the question?"

This is the same absurdity that Douglas Adams captured humorously in *The Hitchhiker's Guide to the Galaxy*, when the supercomputer Deep Thought announces that "the answer to the ultimate question of life, the universe, and everything" is simply 42—without knowing what the question was. The joke resonates because it mirrors a profound logical truth: answers are meaningless without the right questions, and the process of questioning is not separate from the process of answering. In complexity theory, we have treated verification (the "question") and solution (the "answer") as separable processes—but they are not.

This paper argues that P vs NP does not merely *seem* paradoxical; it is a genuine paradox within the framework of classical logic. We will show that, like the Halting Problem, P vs NP involves a self-referential structure that collapses into tautology when analyzed formally. Our goal is not to solve P vs NP but to prove that, as currently posed, it is unprovable. We aim to reveal that the last half-century of attempts to resolve P vs NP have been stymied not by insufficient ingenuity, but by a fundamental category error: the assumption that the relationship between solving and verifying is well-defined in purely formal terms.

In what follows, we will:

1. Re-examine Turing's Halting Problem, explicitly framing it as a paradox rather than a mere proof of undecidability.

2. Trace the history of P vs NP and survey key arguments hinting at its potential independence from formal proof.

3. Analyze the structural similarities between the Halting Problem and P vs NP, showing that both hinge on questions whose answers are only meaningful *after the fact*.

4. Demonstrate that the syntactic treatment of verification in P vs NP hides a semantic self-reference, rendering the problem unprovable.

By the end of this paper, we will argue that P vs NP, like the Halting Problem, should not be thought of as a conventional open question awaiting resolution. Instead, it should be recognized as a paradox that exposes the limits of formal reasoning ...

... then we will show how we can break those limits.

## 2 The Halting Problem as a Paradox

The Halting Problem is typically introduced as an undecidable decision problem: given a description of a program and an input, determine whether the program will eventually halt or run forever. Turing's 1936 proof established that no general algorithm can solve this problem for all possible programs. But beneath the formal result lies a deeper structure—one that reveals the Halting Problem as a paradox grounded in semantic self-reference.

To see this clearly, consider the structure of Turing's proof. Suppose there exists a hypothetical algorithm $\texttt{Halt}(P, x)$ that correctly decides whether program $P$ halts on input $x$. Turing constructs a new program $D$ that takes a program as input and does the opposite of what `Halt` predicts:

> If $\texttt{Halt}(P, P)$ returns `true`, then $D$ loops forever.
> If $\texttt{Halt}(P, P)$ returns `false`, then $D$ halts.

Now consider what happens when $D$ is given itself as input: $D(D)$. If $\texttt{Halt}(D, D)$ predicts that $D$ halts, then by definition $D$ loops forever. If $\texttt{Halt}(D, D)$ predicts that $D$ loops, then $D$ halts. In both cases, the prediction is falsified, leading to a contradiction. The conclusion is that no such `Halt` algorithm can exist.

This is often framed as a result about algorithmic undecidability, but the heart of the proof is semantic. The program $D$ is a self-referential construction: it attempts to decide something about itself by invoking a meta-level description of its own behavior. The contradiction does not arise from a computational error, but from a collapse of meaning—the act of predicting its own behavior recursively undermines the conditions for that prediction to be valid.

The Halting Problem, then, is not just a demonstration of undecidability; it is a formal analog of the Liar Paradox. The sentence "This program halts if and only if it does not halt" is isomorphic to "This sentence is false." In both cases, evaluation leads to a semantic loop that cannot be resolved without stepping outside the system. The structure is not algorithmically complex—it is semantically incoherent under classical bivalent logic.

This insight has important implications. The Halting Problem is not merely a limit on what can be computed—it is a demonstration that certain questions cannot be coherently posed within a given formal system. The act of asking "Will this program halt?" is itself an entanglement of problem and solution. The question generates its own context, and attempting to answer it collapses the assumptions required for a valid answer.

This same structure, as we will show, is embedded in the formulation of the P vs NP problem. Just as halting is only knowable once it happens, verification in NP is only meaningful once a valid certificate is already in hand. The separation of solving and verifying is presumed by the problem statement—but not supported by the semantics of how those concepts function. The paradox arises because the distinction is defined syntactically while being dependent on a semantic collapse.

In the next section, we will examine the standard formulation of P vs NP and show that its core assumption—that "verification" is well-defined independent of "proof"—contains a similar circularity. Once exposed, this reveals that P vs NP is not just difficult to prove within classical logic. It is unprovable because it is, like the Halting Problem, a paradox hiding in plain sight.

## 3 The Structure of P vs NP

The P vs NP problem asks whether every problem whose solution can be *verified* in polynomial time can also be *solved* in polynomial time. Formally, the class NP consists of decision problems for which a proposed solution (or *certificate*) can be checked efficiently by a deterministic Turing machine. The class P consists of those problems that can be solved efficiently without any certificate.

The intuitive formulation seems straightforward: if a problem can be verified quickly, can it also be solved quickly? But beneath this simplicity lies a hidden assumption—that the process of verification is epistemically and semantically independent from the process of solving. This is the assumption we challenge.

To see the structure clearly, consider what it means to "verify" a solution. In NP, verification is defined as a deterministic procedure that takes an instance $x$ and a certificate $y$, and returns `true` if and only if $y$ is a valid proof that $x \in L$, where $L \in$ NP. But this definition presupposes the existence of such a $y$. Verification is only meaningful once a certificate has been generated. It is not a discovery process—it is a post hoc evaluation.

This leads to a recursive dependency. Solving a problem produces a certificate. Verification confirms its validity. But in order to verify that a certificate is valid, we must already know what counts as a valid certificate. And that depends on knowing the structure of the solution space, which is precisely what solving is supposed to uncover. In practice, the verification predicate is designed to match the certificate—but the certificate is often constructed with full knowledge of the verifier. This creates an epistemic loop: verification depends on proof, but proof is only considered valid because of the existence of a verifier.

This circularity is typically hidden beneath the formalism. The function $V(x, y)$ is assumed to be efficiently computable, and the question becomes whether a corresponding algorithm $A(x)$ exists to generate such a $y$. But the real question is more subtle: can the existence of $y$ be determined without already knowing $y$? More precisely, can one generate a solution without already knowing the form of the thing that makes it verifiable?

This is structurally identical to the Halting Problem. Just as a program's halting is only known once it halts, a certificate's validity is only known once it is presented and checked. Any attempt to construct a general algorithm to determine the existence of valid certificates across all NP problems collapses into a tautology: a certificate is solvable if it is verifiable, but it is only verifiable once it has been solved.

The paradox becomes visible when we restate the question: "Can a solution be verified without already knowing what would count as a solution?" In some cases—such as mathematical theorems or SAT instances—the answer is yes, because the verification predicate is explicitly known and tractable. But P vs NP is not about specific problems—it is about the existence of a general transformation from verification to solution across all problem instances. This is where the semantic instability arises.

The core error is a category mistake: treating the verification function $V(x, y)$ as an epistemically primitive act, when in reality it is defined only in the context of a solution that already exists. Verification and solution are not separate steps—they are two views on the same epistemic event. As with the Halting Problem, the distinction collapses under formal scrutiny. The problem thus becomes self-referential: it defines itself in terms of processes that require each other to be meaningful.

In the next section, we will formalize this collapse by aligning the logical structure of P vs NP with that of the Halting Problem. We will show that both involve semantic loops that cannot be resolved within classical logic—because their question structures generate their own undecidability amid non-deterministic contexts. P vs NP is not merely hard; it is paradoxical.

## 4 The Formal Equivalence Between P vs NP and the Halting Problem

Having examined the underlying semantic loop of both the Halting Problem and P vs NP, we now formalize their equivalence. Our goal is not to show that P vs NP *is* the Halting Problem in disguise, but that both share the same underlying paradoxical structure: each asks a question whose answer is only meaningful after the answer is known.

We begin with the Halting Problem. Let $\texttt{Halt}(P, x)$ be a hypothetical algorithm that returns `true` if program $P$ halts on input $x$, and `false` otherwise. Turing's construction creates a contradiction by defining a program $D$ that behaves as follows:

$$D(P) = \begin{cases} \texttt{loop forever} & \text{if } \texttt{Halt}(P, P) = \texttt{true} \\ \texttt{halt} & \text{if } \texttt{Halt}(P, P) = \texttt{false} \end{cases}$$

Evaluating $D(D)$ leads to contradiction in both cases. The paradox arises because the behavior of the system depends on its own prediction. The truth of the halting status is only determinable by observing it directly.

Now consider the formal structure of NP verification. Let $L \in \text{NP}$ be a language, and let $V(x, y)$ be a deterministic verifier that returns `true` if $y$ is a valid certificate for $x \in L$. The problem P vs NP asks: for every such $V$, does there exist a polynomial-time algorithm $A(x)$ such that $V(x, A(x)) = \texttt{true}$?

Rewriting this:

$$\exists A \in \text{P}\ \forall x \in \Sigma^* : V(x, A(x)) = \texttt{true}$$

But the verifier $V$ is defined *with respect to* certificates $y$ that already satisfy the property. There is no guarantee that $A(x)$ can be constructed independently of knowledge of $y$; in fact, $A$ only exists if we already know how to produce certificates for $V$. This reflects a tautological loop:

$$A(x) \text{ solves } x \text{ if and only if } V(x, A(x)) = \texttt{true}$$

Yet $V$ cannot verify anything unless $A(x)$ is already known. The relationship is structurally self-referential.

To draw the equivalence formally, we note the following correspondence:

| **Halting Problem** | **P vs NP** |
|---|---|
| Will program $P$ halt on input $x$? | Is there a poly-time $A(x)$ such that $V(x, A(x)) = \texttt{true}$? |
| Halting is only known after execution | Verifiability is only known after solution is produced |
| Prediction function causes contradiction | Verifier function presupposes knowledge of certificate |
| No general algorithm exists | Existence of algorithm is the central question |
| Contradiction from self-reference | Circularity from semantic dependence |

In both cases, the system attempts to determine something about itself from within. The Halting Problem constructs a program that tries to predict its own behavior and contradicts the result. P vs NP attempts to determine whether solutions can be constructed without already knowing what counts as a solution.

This shows that P vs NP is not a clean decision problem. It is a structurally self-referential paradox. The assumption that verification is an independent, polynomial-time operation collapses when we acknowledge that verification depends on epistemic context—a context that is only defined once a certificate exists.

Like the Halting Problem, the P vs NP question tries to decide the outcome of a process that is only meaningful once it completes. This recursive dependency—verification requiring knowledge of proof, and proof being validated by verification—makes the question semantically unstable. It cannot be resolved by algorithmic or mathematical refinement, because it is not a formally well-defined problem. It is a paradox embedded in the architecture of classical computation.

This insight leads to the conclusion that P vs NP is not just difficult or independent—it is unprovable from within the system that defines it.

## 5 P vs NP is Unprovable

We now bring the argument to its conclusion: P vs NP is not merely unsolved, nor merely independent of standard axioms—it is unprovable, because it is not a well-formed question within the logical framework it presumes. Like the Halting Problem, it contains a recursive dependency that collapses its own premise. But unlike the Halting Problem, which was explicitly constructed to reveal this paradox, P vs NP has hidden its structure behind layers of formal abstraction.

The core assumption behind the P vs NP question is that verification is an epistemically primitive act: that there exists a well-defined, deterministic procedure for confirming whether a certificate solves a problem instance. But this assumption only holds if the meaning of the solution is already known. Verification without context is not validation—it is symbolic matching. And symbolic matching is not truth. It is only meaningful because we, as human interpreters, supply the context that gives the certificate its purpose.

This is not a side issue—it is the missing semantic layer that invalidates the entire structure of the problem. When computational complexity theory was formalized, it did so by intentionally removing semantic context. The verifier was modeled as a deterministic Turing machine, and the notion of "understanding" or "meaning" was discarded as informal. This move enabled a clean theory of classes like P, NP, and EXP—but at a cost: the severing of computation from reality.

In real-world computation, we never simply "verify" in the abstract. We build, test, interpret, debug, and reframe problems continuously. The process of solving and verifying are intertwined, co-defining one another through interaction. Computation does not produce solutions in isolation—it enables us to recognize potential solutions because we bring context to bear. A program that renders a screen, calculates a trajectory, or triggers a UI event is meaningful only because we know what it is for. We are the NP-verifier.

The myth of an NP-verifier that can validate any solution without context is a purely hypothetical construct. It was never realized in practice, nor could it be. It exists only in formal abstraction, and once we ask whether such a verifier is possible for all problems—without regard for context, semantics, or interpretation—we reintroduce the same paradox Turing revealed in the Halting Problem.

To illustrate the point, consider a simple case of formal equivalence that breaks under context. In mathematics, $1 \neq 2$. But in the real world, one large slice of cake may equal two smaller slices. The numerical identity changes based on the ontology we apply. We adjust the formalism to fit the situation, not the other way around. Mathematics gains meaning when it models the world accurately—not when it adheres to syntax at the expense of the reality. What is that reality? It's the approximation of infinity by contexts and their descriptions.

The same is true for complexity theory. By refusing to incorporate informal context into the definition of verification, the theory abstracted itself into paradox. It conflated symbolic evaluation with semantic truth and thus built a decision problem on an epistemic illusion.

P vs NP asks whether every solution that can be verified quickly can also be found quickly. But if the act of verification presupposes the structure of the solution—and that structure only exists

meaningfully within a context—then the question is circular. The verifier and the solver are not separate agents. They are roles we play in constructing meaning from computation.

We end with the unavoidable conclusion: the NP-verifier was always a hypothetical device. It was a placeholder for human interpretation, masquerading as a deterministic machine. It was never real—and thus, the P vs NP problem was never about solving versus verifying in the first place. It was a semantic paradox miscast as a formal question.

For fifty years, the field pursued a formal answer to a structurally unresolvable question. This is not failure—it is a mirror. It reveals that our deepest unsolved problem was not a barrier in the fabric of mathematics, but a misunderstanding of what computation really is.

## 6 Context, Contradiction, and the Collapse of Verification

To understand the structural collapse at the heart of the P vs NP question, we begin with the classical Liar Paradox:

> *"This sentence is false."*

Under classical logic, this statement creates an irresolvable contradiction. If it is true, then it is false. If it is false, then it is true. This loop is logically unstable and formally incoherent.

However, when viewed through the lens of **perspectival semantics**, the sentence does not collapse into absurdity. It branches. It produces a *semantic fork*—a context-sensitive structure that supports multiple stable interpretations depending on the perspective of evaluation.

### Trivalent Interpretations of the Liar Paradox

Let $S$ = "This sentence is false". Then the following interpretations become coherent when context is included:

- **Interpretation A — True:** From one perspective, $S$ is interpreted as truthfully asserting its own falsehood. That is, it *correctly* states that it is false. In this view, the sentence is **true**, because it accurately describes its own falsity.
- **Interpretation B — False:** Alternatively, if we treat the sentence as making a false claim about itself, then it is **false**, because it incorrectly asserts its own falsity.
- **Interpretation C — Both (Contradictory):** From a meta-perspective, the sentence asserts a contradiction that cannot be resolved without a perspective shift. It is **both true and false** simultaneously—containing a truth-functional loop that cannot be flattened into classical bivalence.

This structure reveals a core principle of what we call **perspectivist dialetheism**: Contradictions are not universally true. They are conditionally true, depending on the interpretive context. Ambiguity is not a breakdown of logic—it is its default ontological condition.

### Application to Verification and Computation

This has direct consequences for computational logic. If a sentence or problem admits multiple stable interpretations, then the act of verification cannot proceed until one has been selected. The verifier must operate from within a chosen context. Without it, verification becomes a paradox—just like the Liar Paradox.

This is precisely what we observe in computational decision problems that require external context for resolution. They mirror the logical structure of the Liar Paradox. Their answer state cannot be determined in isolation—it emerges only when a perspective is imposed. Once this happens, the paradox collapses into a stable truth value—but the choice of perspective determines which one.

It is not logic that fails. It is the assumption of context-free evaluation that fails.

With this in mind, we now turn to a computational case study that mirrors this exact structure in everyday reasoning: the "milk" scenario.

### Case Study: Semantic Ambiguity in Everyday Reasoning

Two customers once asked:

> *"Can you guess who likes the milk—the man or the woman?"*

At face value, this appears to be a binary decision problem. Yet any answer is underdetermined without further context. Upon clarification, they revealed:

> *"It's our toddler who likes the milk."*

The original question, then, was not about the adults at all—it implicitly referenced a third party. From a computational perspective:

- **Decision problem:** Given query $Q$: "Who likes the milk?" and dataset $D$ describing the preferences and actions of individuals, determine the referent.
- **Ambiguity:** Multiple interpretations are valid until disambiguated.
- **Verification:** Once the toddler is identified as the referent, both "neither" and "both" can be semantically verified depending on the interpretation of "likes."

This illustrates a fundamental asymmetry:

**Context-dependent decision problems are computationally intractable to solve without disambiguation, yet trivial to verify once context is given.**

### Formal Modeling

Let:

$$D = \begin{Bmatrix} \neg\texttt{likes}(\texttt{man}, \texttt{milk}), \\ \neg\texttt{likes}(\texttt{woman}, \texttt{milk}), \\ \texttt{likes}(\texttt{toddler}, \texttt{milk}), \\ \texttt{buys}(\texttt{man}, \texttt{milk}), \\ \texttt{buys}(\texttt{woman}, \texttt{milk}) \end{Bmatrix}$$

Let $Q =$ "Who likes the milk?", and define context interpretations $C_i$:

$$C_1 : \texttt{likes} \mapsto \text{consumes} \quad \Rightarrow \quad \texttt{Answer} = \text{Neither}$$

$$C_2 : \texttt{likes} \mapsto \text{values-for-child} \quad \Rightarrow \quad \texttt{Answer} = \text{Both}$$

Let $M(Q, C, D) \to \texttt{Answer}$ be a deterministic machine. Then:

- Without $C$, $M$ cannot resolve $Q$.
- With $C$, $M \in \mathrm{P}$.

Thus, context functions as a certificate: a symbolic selection which collapses semantic ambiguity and renders the verification problem tractable.

### Contextual Incompatibility Theorem (Restated)

### The Real Role of Context in Computation

The assumption that verification is simpler than solving rests on the idea that solutions can be checked without interpreting the question. But that assumption collapses once context is required. In the milk example, the "solution" (the toddler) was not even part of the formal query until context made it so.

This is not a flaw in the question. It is a flaw in the model of verification.

### Mathematical Verification is Contextual

As a child (depending on whether or not you had teachers like mine) if you gave the correct answer in a math class but failed to show your work, your answer was marked wrong. The answer was correct—but unverifiable. Verification, then, is not a just secondary step after solving.

It is part of the solution process.

It requires understanding the reasoning pathway—the context through which the answer makes sense.

In real reasoning, when you fully understand a question, solving it *is* verification. When you do not understand, verifying a correct answer may be impossible. Ironically, that's why there is a two-year waiting period for the mathematical solution for P vs NP. In other words, even if you could hypothetically mathematically prove that $P = NP$ within an hour, verification takes exponentially longer. The answer was right there in the rules of the contest the entire time.

### Implication: P vs NP is Semantically Inverted

This leads to a profound inversion:

> **In the presence of meaningful context, solving can be easier than verifying, and in the absence of context, verification is meaningless.**

This flips the foundation of P vs NP. The belief that verification is always easier than solving only holds under the assumption that the verification machine requires no semantic understanding of the question. That assumption is false.

Verification and solving are not distinct processes. They are different phases of the same semantic event.

The concept of a universal NP-verifier—one that can confirm any correct answer without knowing what it means—was always a theoretical artifact. It does not exist in real reasoning, and it cannot be instantiated in real computation.

### Conclusion

The milk example shows that contextual disambiguation *is* the core computational task. Once the right interpretation is known, the answer is obvious. Verification is efficient—but only because the problem has already been solved in the meaningful sense.

This reveals the truth of the paradox: P vs NP assumes a separation between solving and verifying that is semantically incoherent. The verifier is not a machine. It is the human mind that brings context, meaning, and understanding.

And that means P vs NP was always a paradox—because it asked a question that only makes sense after it has been answered.

## 7 From Contradiction to Context: Perspectivistic Dialetheism

My route into the problem of P vs NP did not begin in mathematics—it began in language, logic, and contradiction.

I had long been unsettled by how easily contradiction is dismissed in formal logic. The principle of explosion tells us that from a contradiction, anything follows. But I found this counterintuitive. In actual reasoning, contradictions do not collapse meaning—they *generate* it. They force clarification, perspective, and semantic branching.

My first insight came through philosophical reflection: what if contradictions are not always logically fatal? What if, instead, some contradictions are **true from one perspective and false from another**? This was not simply dialetheism—the idea that some contradictions are true. It was something more precise: the truth value of a contradiction could vary *by perspective*.

Thus emerged **perspectivistic dialetheism**.

In this model:

- A statement may be **true** within one interpretive frame,
- **false** within another,
- and **both** true and false within a shared or undefined space.

The contradictions themselves generate their own interpretive conditions—they are not anomalies within a context but *producers of context*.

I later discovered a striking parallel in psychology, specifically in **Relational Dialectics Theory**, developed by Leslie A. Baxter and Barbara M. Montgomery. In that theory, meaning in relationships arises from the tension between opposing communicative needs—autonomy and connection, openness and closedness. Importantly, these tensions are not problems to be resolved, but forces that *create* relational meaning.

This mirrored my view exactly: contradiction is not a flaw to eliminate, but a structure that gives rise to meaning.

From there, I began to recognize that many unsolved logical and computational problems—particularly the Liar Paradox, the Halting Problem, and P vs NP—shared this form. They were not simply hard or unsolvable. They were **perspectival contradictions**. They produced their own contexts and made meaning ambiguous until a perspective collapsed the uncertainty.

What classical logic saw as ill-formed questions, I began to see as insight-rich problems that revealed the limits of binary formalism.

When I later returned to P vs NP, I realized the same pattern was present: the problem only exists as a paradox because it presupposes a verifier that exists outside of interpretation. But no such thing exists. Like the contradictory statements I had previously studied, the P vs NP question

produces a context in which verifying and solving are inseparable—and trying to split them creates the paradox itself.

It was then I understood: P vs NP is not just a mathematical question. It is a semantic contradiction that requires a perspectivist lens. And only through that lens can we see that the problem was never about algorithms—it was about context, meaning and truth all along.

## 8 Formalizing the Collapse of Verification in Contextual SAT

To formalize the paradox at the heart of context-sensitive verification, we reduce the problem to a SAT-like decision structure and demonstrate how it becomes semantically unstable without an external interpretive certificate.

Let us begin with a propositional clause:

$$(A \vee B) \wedge \neg A \wedge \neg B$$

This clause is clearly unsatisfiable under classical Boolean logic: it demands that at least one of $A$ or $B$ be true, while simultaneously asserting that neither is.

Now, reinterpret the clause semantically. Let:

$$\begin{aligned} A &:= \text{The stick is a sword-stick} \\ B &:= \text{The stick is a wand-stick} \\ A \vee B &:= \text{The child has a stick that is either a sword or a wand} \end{aligned}$$

In symbolic context, the stick is neither fully a sword nor fully a wand—so both $A$ and $B$ are technically false. Yet from a narrative perspective, $A \vee B$ is true: the child clearly has a toy with imaginary properties.

This reveals a fatal issue: the clause is semantically satisfiable in human context but unsatisfiable in formal logic. The problem is not in the propositional form—it is in the assumption that each symbol $A$ and $B$ maps to a discrete, context-free truth value.

Now let us model this as a context-sensitive SAT instance.

### Contextual SAT Collapse

Let:

$$\Phi = (A \vee B) \wedge \neg A \wedge \neg B$$

$$\mathcal{I} : \text{Set of interpretations of } A, B$$

Then the satisfiability of $\Phi$ under $\mathcal{I}$ depends entirely on semantic compression: whether some external certificate collapses the meaning of "stick" into a specific referent. Without this, the verifier must evaluate all plausible interpretations of each atom across a semantic space:

$$|\mathcal{I}| = O(2^n)$$

Where $n$ is the number of context-dependent symbols.

Thus, verification without a semantic certificate becomes equivalent to brute-forcing across exponentially many meaning permutations.

### Reduction to General Case

A verification machine $V$ attempts to validate a candidate assignment for an ambiguous SAT instance $\Phi$ with symbols $x_1, x_2, ..., x_n$, each of which has multiple latent interpretations $\{x_{i1}, x_{i2}, ..., x_{ik}\}$. Then:

$$\text{Verification} = \min_{C \in \mathcal{C}} \texttt{SAT}(\Phi_C)$$

Where: - $C$ is a contextual certificate disambiguating all terms, - $\Phi_C$ is the fully grounded version of $\Phi$, - and without $C$, $V$ must evaluate:

$$\Phi_{\text{all}} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k} \texttt{SAT}(\Phi_{x_{ij}})$$

This evaluation becomes intractable or contradictory. The logical system does not know what "stick" is—until a context imposes the meaning.

### Parallel: The White Horse Paradox

This resembles the classical *White Horse Dialogue* from Chinese philosophy, which asserts:

> "A white horse is not a horse."

While absurd under classical logic, this is semantically sound: "horse" and "white horse" differ in category granularity. The former refers to a general kind; the latter to a specific instantiation.

Similarly, in our SAT structure: - $A$ and $B$ are category-incompatible, - but $A \vee B$ is semantically true because the ambiguity refers to a third category (e.g., imaginative play object).

The logic breaks not because the reasoning is flawed, but because formal symbols cannot track cross-level semantic references.

### Conclusion

This micro-proof shows why context-sensitive verification becomes formally intractable:

### Symbolic Encoding of Context and Gödelian Collapse

One might object that context is ineffable, or outside the domain of formal computation. But Gödel's incompleteness theorems show otherwise: even semantic truths about logical systems can be encoded symbolically within those systems. This encoding was achieved via Gödel numbering—a mapping of syntactic elements to natural numbers, allowing meta-mathematical statements to be made within arithmetic.

The implication is profound: paradoxes and contexts are not beyond formalization—they can be *coded*.

Let us apply this to the verification paradox described earlier. Let each contextual disambiguation $C_i \in \mathcal{C}$ be encoded as a unique symbol or string. Then any verifier that attempts to evaluate a context-sensitive formula $\Phi$ must also evaluate:

$$\Phi_{C_i} = \texttt{SAT}(\Phi \mid C_i)$$

But if the space of possible $C_i$ is itself dependent on interpretive context—such as referents in natural language or narrative intent—then encoding the correct $C_i$ requires referencing semantic information not present in $\Phi$ itself.

This leads to a Gödelian recursion: the verifier must include within itself an encoding of the context in which it operates. But such encodings eventually reference contexts about the context, and so on. The system becomes meta-reflexive, and thus formally incomplete.

In other words, the attempt to build a universal verifier that accounts for semantic context leads to:

- Self-reference (Gödel-type encoding of semantic position),
- Undecidability (since the system refers to truths it cannot internally prove),
- and paradox (when interpretations recursively contradict).

This is not merely a philosophical echo of Gödel's theorems—it is a structural replication. Verification fails not because computation is weak, but because the system is being asked to resolve its own semantic interpretation internally.

$$\boxed{\text{No verifier can resolve meaning without importing symbolic context.}}$$

$$\boxed{\text{But importing that context collapses the verification boundary.}}$$

Hence, even symbolically, the attempt to preserve P vs NP as a formal distinction in the presence of context leads to an undecidable, Gödelian structure.

The paradox does not merely reflect reality—it *is* the result of applying reality to formalism.

- Without a semantic certificate, verification reduces to brute-force disambiguation across interpretation space.
- Classical logic treats semantic forks as contradictions rather than latent truths.
- The gap between representation and referent makes the verification problem unsolvable in deterministic polynomial time.

Thus, any attempt to build a universal verifier that does not include semantic context cannot model reality. And therefore:

$$\boxed{P \neq NP}$$

Not as a result of formal complexity-theoretic separation, but because formal verification *without perspective* is an undecidable construct.

When the question requires context to resolve, and context cannot be formally compressed, verification becomes semantically indistinguishable from solving—or worse, logically impossible.

## 9 They Were All Right — And Now We Know Why

Many thinkers throughout the history of mathematics, computer science, and philosophy have sensed that the P vs NP problem concealed something deeper than a complexity class distinction.

Kurt Gödel intuited it in his concerns about provability and decidability. John Nash felt it in his sense that cryptographic difficulty reflected real-world intractability. Roger Penrose suspected it in

his argument that human consciousness could not be fully captured by computational algorithms. Scott Aaronson has hinted that $P \neq NP$ is not just likely true, but must be, for reality to remain coherent. Alan Turing, with the halting problem, uncovered the semantic recursion that lies at the heart of this paradox.

Each of them grasped part of the answer. What they lacked was not intelligence or rigor — it was simply the semantic frame. They were trying to verify an insight before its full formulation had been formally proved.

> They each held a piece of the truth. What they lacked was the context to collapse the ambiguity.

Now, that context is visible. $P \neq NP$ is not merely a gap in computation time. It is a structural separation between abstract logic and embedded reasoning. It is the realization that truth in reality is ambiguous — until collapsed by interaction. That interaction is what we call understanding.

## The Good News

This realization is not a death knell for progress. It is a gateway.

We thought that proving $P \neq NP$ would be a terrible thing. That it would close doors. That it would freeze our future.

> But if that was our fear, then we misunderstood the problem as badly as we misunderstood the solution.

Because now we know:

*Since we can prove that $P \neq NP$ in hypothetical abstractions of reality,*

*It means we can design systems that approximate $P = NP$ in reality — by building machines that reason through interaction, and learn by living in ambiguous truth.*

This is the promise of dialetheic perspectivism in philosophy and computation.

I therefore propose a logic system that doesn't just describe ambiguity, but harnesses it as a core feature. A machine that does not merely solve problems, but understands them through perspective and context created by interaction.

## Toward a Future of Embodied Semantics

In this framework, ambiguity is not a bug. It is a structure for exploration.

Contradiction is not a flaw. It is the presence of multiple meanings waiting to be interpreted.

Verification is not always checkmark. It is a assessment of alignment between problem solving and truth.

In short: We do not have to fear that $P \neq NP$ Consider this: Integrate this trivalent logic with an LLM. I know it needs work, but I'm confident that, because of its simplicity, contradiction resilience and versatility, it might pave the way to approximate $P = NP$ in ways we couldn't consider while we still formally treated paradox as an error.

Listing 1: Trivalent Logic Engine with Contextual Evaluation

```
from enum import Enum
from typing import Callable, Dict, List, Tuple, Set
from dataclasses import dataclass, field
from functools import reduce
import ast
```

```
import datetime
from collections import defaultdict

class Tvalue(Enum):
    FALSE = 0
    TRUE = 1
    BOTH = 2

    def __invert__(self):
        return {
            Tvalue.FALSE: Tvalue.TRUE,
            Tvalue.TRUE: Tvalue.FALSE,
            Tvalue.BOTH: Tvalue.BOTH
        }[self]

    def __and__(self, other):
        return {
            (Tvalue.TRUE, Tvalue.TRUE): Tvalue.TRUE,
            (Tvalue.TRUE, Tvalue.BOTH): Tvalue.BOTH,
            (Tvalue.BOTH, Tvalue.TRUE): Tvalue.BOTH,
            (Tvalue.BOTH, Tvalue.BOTH): Tvalue.BOTH,
        }.get((self, other), Tvalue.FALSE)

    def __or__(self, other):
        return {
            (Tvalue.FALSE, Tvalue.FALSE): Tvalue.FALSE,
            (Tvalue.FALSE, Tvalue.BOTH): Tvalue.BOTH,
            (Tvalue.BOTH, Tvalue.FALSE): Tvalue.BOTH,
            (Tvalue.BOTH, Tvalue.BOTH): Tvalue.BOTH,
        }.get((self, other), Tvalue.TRUE)

def parse_logical_expression(expr: str) -> Callable[[Dict[str, Tvalue]], Tvalue]:
    expr_ast = ast.parse(expr, mode='eval')

    def make_eval(node):
        if isinstance(node, ast.BoolOp):
            op = Tvalue.__and__ if isinstance(node.op, ast.And) else Tvalue.__or__
            children = [make_eval(v) for v in node.values]
            return lambda context: reduce(op, [child(context) for child in
                children])
        elif isinstance(node, ast.UnaryOp) and isinstance(node.op, ast.Not):
            child = make_eval(node.operand)
            return lambda context: ~child(context)
        elif isinstance(node, ast.Name):
            return lambda context: context.get(node.id, Tvalue.BOTH)
        else:
            raise ValueError(f"Unsupported AST node: {type(node)}")

    return make_eval(expr_ast.body)

@dataclass
class Perspective:
    name: str
    evaluate_fn: Callable[[str], Tvalue]
    memory: Dict[str, Tvalue] = field(default_factory=dict)

    def evaluate(self, statement: str) -> Tvalue:
        if statement not in self.memory:
            self.memory[statement] = self.evaluate_fn(statement)
```

```
        return self.memory[statement]

    def merge_with(self, other: 'Perspective', new_name: str) -> 'Perspective':
        def composed_eval(statement: str) -> Tvalue:
            v1 = self.evaluate(statement)
            v2 = other.evaluate(statement)
            return v1 if v1 == v2 else Tvalue.BOTH
        return Perspective(name=new_name, evaluate_fn=composed_eval)

class CollapseCache:
    def __init__(self):
        self.cache: Dict[str, List[Tuple[Tvalue, float]]] = defaultdict(list)

    def record(self, prop: str, value: Tvalue, weight: float = 1.0):
        self.cache[prop].append((value, weight))

    def get_weighted_score(self, prop: str) -> float:
        entries = self.cache.get(prop, [])
        total_weight, net_score = 0.0, 0.0
        for val, weight in entries:
            if val == Tvalue.TRUE:
                net_score += weight
            elif val == Tvalue.FALSE:
                net_score -= weight
            total_weight += weight
        return net_score / total_weight if total_weight > 0 else 0.0

    def get_stability(self, prop: str) -> float:
        return abs(self.get_weighted_score(prop))

@dataclass
class Proposition:
    statement: str
    contextual_value: Tvalue = Tvalue.BOTH
    perspective_values: Dict[str, Tvalue] = field(default_factory=dict)
    collapse_history: List[Tuple[datetime.datetime, Tvalue]] = field(
        default_factory=list)

    def _compute_dynamic_threshold(self, perspectives: List[Perspective]) -> float
        :
        count = len(perspectives)
        return min(max(0.5 / (1 + 0.1 * count), 0.1), 0.5)

    def evaluate(self, perspectives: List[Perspective], cache: CollapseCache) ->
        Tvalue:
        values: Set[Tvalue] = set()
        for perspective in perspectives:
            value = perspective.evaluate(self.statement)
            self.perspective_values[perspective.name] = value
            cache.record(self.statement, value)
            values.add(value)

        threshold = self._compute_dynamic_threshold(perspectives)
        score = cache.get_weighted_score(self.statement)

        if Tvalue.TRUE in values and Tvalue.FALSE in values:
            self.contextual_value = Tvalue.BOTH
        elif score > threshold:
            self.contextual_value = Tvalue.TRUE
```

```
        elif score < -threshold:
            self.contextual_value = Tvalue.FALSE
        else:
            self.contextual_value = Tvalue.BOTH

        self.collapse_history.append((datetime.datetime.now(), self.
            contextual_value))
        return self.contextual_value

    def compute_stability(self, cache: CollapseCache) -> float:
        return cache.get_stability(self.statement)
```