Can You Trust an AI to Do the Same Thing Twice?

A Plain-English Guide to How AI Computes — and Why Its Answers Can Vary

Kachyng Inc.

Part 2 of 8 · July 2026

Before We Begin: An Experiment

In September 2025, researchers at Thinking Machines Lab, an artificial intelligence research company, ran the same question — "Tell me about Richard Feynman" — through the same AI model one thousand times. They set the system to always select the single highest-scoring next word, removing every setting that deliberately introduces variety.[1]

The result was not one repeated answer. It was eighty distinct answers.

All 1,000 responses were word-for-word identical at first. AI models produce text in small pieces — a word or part of a word at a time — and every one of the 1,000 responses agreed on the first 102 pieces. At the 103rd, a microscopically small numerical difference changed which continuation was chosen, and from there the responses went their separate ways. When the researchers replaced the ordinary calculation routines with versions designed to behave the same way regardless of how busy the server was, all 1,000 runs produced exactly the same answer.

Albert Einstein famously objected to quantum mechanics with the remark that God does not play dice with the universe. Whatever the truth of that in physics, this paper makes a narrower and more practical claim about artificial intelligence:

> The variation observed in this experiment was not an unavoidable property of the AI model. It arose from engineering choices in the system that runs the model — specifically, calculations whose results changed depending on how much other work the computer happened to be doing at the same moment. That source of variation can be removed.

One caution before we begin. Not all variation in AI answers is of this kind. Sometimes a system is deliberately set to vary its answers. Sometimes a question is genuinely ambiguous and admits more than one reasonable reading. Sometimes the world simply changes between one asking and the next. Those are different phenomena, and this paper will keep them carefully separate.[2] Our

[1] The experiment. He, Horace, and Thinking Machines Lab, "Defeating Nondeterminism in LLM Inference," Thinking Machines Lab: Connectionism, September 2025 (thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/). The reported configuration: the prompt "Tell me about Richard Feynman" run 1,000 times against Qwen3-235B-A22B-Instruct-2507 in non-thinking mode, generating 1,000 tokens per run at temperature zero. The runs produced 80 unique completions; all completions were identical for the first 102 tokens, with the first divergence at token 103. With batch-invariant kernels enabled, all 1,000 completions were identical.

[2] Scope of the claim. The precise term for the phenomenon this paper addresses is inference nondeterminism: identical inputs to an inference system yielding non-identical outputs when no deliberate randomness is requested. This is distinct from (a) deliberate stochastic sampling, (b) semantic ambiguity in natural language, (c) changes in context, data, or

subject is the accidental kind of variation — the kind nobody asked for — and the engineering that produces and can eliminate it.

To see how it arises, we have to start at the very bottom: with how a computer stores a number. Every concept in this paper builds on the one before it. By the end, the chain from a single binary storage position to a diverging AI answer — and to what it takes to stop those particular dice — will be complete.

Executive Summary

Modern artificial intelligence depends on an enormous number of numerical calculations. These calculations are not generally performed with exact whole numbers. They use positive and negative fractional values that represent weights, activations, probabilities, similarities, and other learned relationships inside a neural network.

A computer cannot store every such value with unlimited accuracy. It has a fixed number of binary storage positions, called bits, available to describe each number. Those bits must be divided among several jobs:

- whether the number is positive or negative;
- how large or small the number is;
- and how many meaningful digits are retained.

This creates a fundamental engineering trade-off between storage, numerical range, precision, memory use, and computational speed.

When a computer cannot retain every meaningful digit, it rounds the value. Because rounding happens after intermediate steps, changing how a long calculation is divided into groups — and the order in which those groups are combined — can slightly change the final result.

Large language models perform billions or trillions of such operations. Most rounding differences are immaterial. But when two possible next words have nearly identical scores, a tiny numerical

external state between runs, and (d) model or infrastructure version changes. The paper's claims about removability apply to (batch-sensitive) inference nondeterminism, not to AI output variability in general. The property that the same input always produces the same output is formally called determinism; the main text deliberately uses the everyday word repeatability instead.

difference can change which one is selected. That first difference can then cause the rest of the response to diverge.

This paper develops that chain from first principles. It closes with four questions every business should ask of an AI system that acts on its behalf: did it understand what you meant, did it compute the same answer, did it execute the same business process, and did the customer receive the same result (Figure 1, Section 21)? Making the arithmetic repeatable answers only one of them.

The main text uses ordinary English throughout. Readers who want the formal terminology, exact technical caveats, and sources will find them in the footnotes at the bottom of each page.

1. How Does a Computer Store Anything at All?

At the lowest useful level, a computer stores information in bits.

A bit is one storage position that can contain either:

```
0
```

or:

```
1
```

An 8-bit value has eight such positions:

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

A 16-bit value has sixteen. A 64-bit value has sixty-four.

The easiest way to understand bits is therefore:

```
A bit count tells us how many binary storage positions
are available to describe a value.
```

More positions allow the computer to preserve more information.

But the positions are not always used only for the digits of the number. Their use depends on the type of number being stored.

2. Is Counting the Same as Measuring?

Before going further, it is necessary to separate two kinds of numbers.

2.1 Integers

Integers are whole numbers:

```
5        24        1,000        -87
```

They are used for counting discrete things:

- 12 employees;
- 200 orders;
- 3 failed transactions;
- 10,000 records.

An integer representation can store a whole number exactly, provided that the number falls within the available range.

For example, a computer might store the integer 25 as a binary pattern. No fractional portion must be preserved.

2.2 Fractional values

Now consider:

```
0.348217        -1.28734        12.3456789        0.00001827
```

These numbers contain fractional information. They may describe:

- a measurement;
- a probability;
- a percentage;
- a coefficient;
- a model weight;
- a similarity score;
- or an intermediate result of a calculation.

In this paper, phrases like "decimal value" and "fractional value" are used in the ordinary human sense: a number with a fractional part, as a person would write it. Inside the computer, these quantities are not stored as written decimal digits; they are stored in a binary format that this paper will build up step by step.[3]

These values raise a different question:

```
How many meaningful digits should the computer retain?
```

That is the beginning of precision.

3. Why Doesn't AI Just Use Whole Numbers?

A neural network is not primarily counting objects. It is representing learned relationships.

A model may need to express ideas such as:

- how strongly one feature should influence another;
- how much attention one word should give another word;
- how active a particular internal feature is;
- or how likely one possible next word is compared with another.

Those relationships are rarely exact whole numbers.

A simplified internal calculation might look like:

```
0.348217 x 0.987632 + 0.004218
```

A model contains billions of learned values of this general kind. These values are commonly called weights. A weight can be thought of as a learned adjustment:

```
Weight A =  0.348217
Weight B = -1.28734
Weight C =  0.004218
```

When the trained model is put to work producing answers — a stage engineers call inference — it repeatedly multiplies and adds these values.

[3] "Decimal values." GPUs operate on binary floating-point approximations of real-valued quantities, not on decimal-digit representations. The main text's usage is pedagogical shorthand.

That is why the arithmetic of fractional numbers is central to artificial intelligence.

4. Which Digits Actually Matter?

Consider the following values:

```
1234        12.34        1.234        0.1234        0.001234
```

Each contains the same meaningful sequence:

```
1  2  3  4
```

Each therefore has four significant digits.

The zeros before the first non-zero digit do not add precision. They only show where the decimal point is positioned.

For example, 0.00000123456 has six significant digits:

```
1  2  3  4  5  6
```

It does not have eleven significant digits merely because many zeros appear after the decimal point.

Significant digits answer:

```
How many meaningful digits are available to describe the value?
```

5. Are Decimal Places the Same as Meaningful Digits?

These two terms are commonly confused.

Decimal places

Decimal places count the number of digits after the decimal point. For example, 12.345 has three decimal places.

Significant digits

Significant digits count the meaningful digits beginning with the first non-zero digit. The same number, 12.345, has five significant digits.

Now consider:

```
0.00012345
```

It has eight decimal places but only five significant digits.

This distinction matters because computer precision is generally better understood in terms of significant digits, not a fixed number of digits after the decimal point.

The decimal point can move. The number of meaningful digits remains limited.

6. How Much Detail Can a Computer Remember?

Precision answers:

```
How much numerical detail can the computer retain?
```

Suppose an imaginary system can retain only four significant decimal digits.

The value 0.348217 must be rounded to approximately:

```
0.3482
```

The value 12.345678 must be rounded to approximately:

```
12.35
```

The value 123,456 must be rounded to approximately:

```
123,500
```

The decimal point moves, but the system still retains only about four meaningful digits.

Notice what this implies. Such a system can tell the difference between 1.234 and 1.235. But at a much larger scale, the finest distinction it can make is correspondingly larger: it can tell 1,234,000

from 1,235,000, but nothing finer. The gaps between the values the system can represent grow as the values themselves grow.[4]

Precision is therefore not the same as the number of decimal places. It is the amount of meaningful numerical information retained — a roughly constant number of meaningful digits, wherever the decimal point happens to sit.

7. How Can One Computer Represent Both 0.000001 and 123,400,000?

A computer may need to represent both:

```
123,400,000
```

and:

```
0.000001234
```

The meaningful digits are similar:

```
1  2  3  4
```

What changes is the scale. One value is very large. The other is very small.

A fixed-point system places the decimal point in one fixed location. This can work well for values such as money:

```
$12.34        $500.00        $9.75
```

But it is inefficient when the system must represent numbers across an enormous range.

Floating point solves this by storing the number in separate conceptual parts:

- the sign;
- the meaningful digits;

[4] Relative precision. Floating-point formats provide approximately constant relative precision across their normal range: a roughly fixed count of significant digits, with the absolute spacing between adjacent representable values growing proportionally with magnitude. Precision is therefore not a fixed decimal-place count, and the four-significant-digit system in the text is an illustrative simplification. This is also why a format preserving roughly four significant digits is not limited to four digits after the decimal point: 1.234, 12.34, 123.4 and 0.001234 are all represented with similar relative precision, because the point floats with the scale.

- the scale of the number.

A human version resembles scientific notation:

```
1.234 x 10^8        or        1.234 x 10^-6
```

The significant digits remain 1.234. The scale part changes where the decimal point sits.

The decimal point effectively "floats," which is where the term floating point comes from.

8. How Are the Storage Positions Divided Up?

A floating-point format divides its available storage positions among three purposes.

8.1 Sign

One part records whether the number is positive or negative:

```
+1.234        -1.234
```

8.2 Scale

Another part records how large or small the number is — how far the point is shifted. This controls the numerical range.

8.3 Meaningful digits

The remaining part records the meaningful digits of the number.[5] This part largely controls the numerical precision.

A simplified diagram looks like:

```
[ Sign ] [ Scale ] [ Meaningful digits ]
```

The total number of bits is fixed.

- Giving more bits to the scale increases range.

[5] Stored layouts and the hidden leading bit. The part of a floating-point number that holds the meaningful digits is formally called the significand (historically, the mantissa); the scale part is the exponent. Stored field layouts: FP32 = 1 sign / 8 exponent / 23 fraction bits; FP16 = 1/5/10; BF16 = 1/8/7. For normal IEEE-style binary floating-point values, the leading binary digit is implicit rather than stored, so effective significand precision is one bit greater than the stored fraction-bit count (e.g., FP16 provides 11 bits of significand precision from 10 stored bits).

- Giving more bits to the meaningful digits increases precision.

This is a constrained storage-budget problem. Keep this budget in mind: it explains not only the difference between big and small formats, but also — as Section 10 will show — the difference between two formats of exactly the same size.

9. Is Reaching Far the Same as Measuring Finely?

A number format has both precision and range.

Precision

Precision asks:

```
How many meaningful digits can I retain?
```

Range

Range asks:

```
How large or how small a number can I represent?
```

A useful analogy is a measuring instrument.

A ruler that reaches 100 meters but is marked only every meter has:

- large range;
- poor precision.

A measuring instrument that reaches only 10 centimeters but is marked every micrometer has:

- small range;
- high precision.

Floating-point formats make a similar trade-off. A format may support very large and very small values while preserving only a limited number of significant digits.

10. What Do FP64, FP32 and FP16 Actually Mean?

The letters FP mean floating point. The number after them tells you the total number of storage positions each value receives.

FP64 uses 64 storage positions and preserves roughly 15 to 17 meaningful digits. FP32 uses 32 positions and preserves roughly seven. FP16 uses 16 positions and preserves roughly three to four. Smaller formats mean less memory per value, less data to move around, and — on most processors designed for AI work — much higher speed.[6] The price is fewer meaningful digits, and therefore earlier and coarser rounding.

One more detail completes the picture, and it follows directly from the storage-budget idea of Section 8: two formats of the same size can behave very differently. A 16-bit format called BF16 divides the same sixteen positions differently from FP16 — more positions for the scale, fewer for the meaningful digits.[7] It keeps less detail, but it can represent the same enormous span of magnitudes as FP32, and that trade suits the training of neural networks, which produces values across a very wide range.[8] The lesson generalizes:

```
The bit count sets the budget.
The split decides the behavior.
```

Everything else about number formats — the even smaller eight-bit and four-bit varieties, the exact layouts, the hardware differences — belongs in the notes, not the narrative.[9]

---

[6] Hardware-dependent performance. Relative throughput between formats depends on the processor. On GPUs optimized for AI workloads, FP64 throughput is commonly a small fraction of FP32/FP16/BF16 throughput; on other hardware the ratios differ. Memory-per-value comparisons hold universally; speed comparisons are hardware-dependent.

[7] Stored layouts and the hidden leading bit. The part of a floating-point number that holds the meaningful digits is formally called the significand (historically, the mantissa); the scale part is the exponent. Stored field layouts: FP32 = 1 sign / 8 exponent / 23 fraction bits; FP16 = 1/5/10; BF16 = 1/8/7. For normal IEEE-style binary floating-point values, the leading binary digit is implicit rather than stored, so effective significand precision is one bit greater than the stored fraction-bit count (e.g., FP16 provides 11 bits of significand precision from 10 stored bits).

[8] BF16 adoption. BF16 retains FP32's exponent range while providing less significand precision than FP16. Its prevalence in training reflects the empirical finding that wide dynamic range (avoiding overflow/underflow) matters more than significand precision for most training workloads; actual format choice varies by model, accelerator generation, framework, and workload.

[9] Low-bit formats. The two common FP8 variants are E4M3 (4 exponent / 3 fraction bits, favoring precision) and E5M2 (5 exponent / 2 fraction bits, favoring range). Integer quantization schemes (INT8, INT4) store small integers together with per-group scale factors and sometimes zero-points; reconstructed value = stored integer x scale (+ zero-point

11. What Happens If We Deliberately Throw Away Some of the Digits?

Two related but distinct ideas are commonly described using bits.

11.1 Weight precision

A large language model (LLM) — an artificial intelligence model trained to predict and generate text — contains billions of stored weights. A weight may originally be:

```
0.34821749
```

If stored using a lower-precision representation, the model can only keep approximately:

```
0.348
```

The missing information has been discarded. This conversion of model values to a lower-bit representation is called quantization. A model described as 16-bit, 8-bit, or 4-bit is usually being described by how its weights are stored. In many low-bit formats the stored value is not a miniature fraction at all, but a small whole number kept alongside a shared multiplier — 87 stored, multiplied by a shared 0.004, reconstructs 0.348 — with the details varying by scheme.[10]

11.2 Calculation precision

The graphics processing unit (GPU) — the specialized parallel processor on which most AI arithmetic runs — may perform the arithmetic using another format. For example:

```
Weights stored in 8 bits
    converted into a 16-bit working format
    multiplied
    partial totals accumulated in 32 bits
```

adjustment). Newer block-scaled or microscaling formats share scale information across small blocks of values. "An 8-bit model" therefore does not identify a single representation.

[10] Low-bit formats. The two common FP8 variants are E4M3 (4 exponent / 3 fraction bits, favoring precision) and E5M2 (5 exponent / 2 fraction bits, favoring range). Integer quantization schemes (INT8, INT4) store small integers together with per-group scale factors and sometimes zero-points; reconstructed value = stored integer x scale (+ zero-point adjustment). Newer block-scaled or microscaling formats share scale information across small blocks of values. "An 8-bit model" therefore does not identify a single representation.

Expanding an 8-bit stored value into a 16-bit container does not restore the information previously discarded:

```
Original value:            0.34821749
Stored at low precision:   0.348
Expanded for calculation:  0.34800000
```

The larger working format gives the calculation more room, but it does not recreate the lost digits.

It is similar to displaying a low-resolution photograph on a larger screen. The screen has more pixels, but the missing image detail does not return.

12. Why Would Anyone Choose Less Precision?

LLMs perform an extraordinary number of calculations. Using fewer bits offers major benefits:

- less memory is required to store the model;
- less data must be moved between memory and the processor;
- more values can be processed simultaneously;
- inference can be faster;
- energy and hardware costs can be reduced.

Neural networks are often tolerant of small numerical approximations. A model may continue to perform effectively even when some weights and calculations use fewer significant digits.

The trade-off is that lower precision increases the size and frequency of rounding effects.

13. When Does Rounding Happen — at the End, or Along the Way?

Engineers routinely round measurements. For example, a measured value may be:

```
12.3456789 mm
```

but be reported as:

```
12.346 mm
```

depending on the required tolerance. The principle is familiar: more digits do not always represent useful or justified information.

The important distinction is when rounding occurs.

An engineer may retain additional digits through a calculation and round the final result. A floating-point system must ensure that every intermediate result fits inside its available format. It therefore behaves more like:

```
Calculate  ->  round to available precision  ->
calculate  ->  round again  ->  calculate  ->  round again
```

The computer is not choosing to be careless. Each intermediate value must fit into a finite storage representation.

14. Why Can't a Computer Store 0.1 Exactly?

Humans normally write numbers in base 10. Computers represent numbers using binary digits, 0 and 1.

Some decimal fractions cannot be represented exactly using a finite number of binary fractional positions. This is similar to how one-third cannot be written exactly as a finite decimal:

```
1 / 3 = 0.333333...
```

In binary, a decimal value such as 0.1 also requires a repeating representation. The computer must eventually stop and store the nearest representable value. As a result, the stored value is extremely close to 0.1, but not mathematically identical to it.

This is why some systems display:

```
0.1 + 0.2 = 0.30000000000000004
```

The result is not evidence that the computer does not understand addition. It is the visible consequence of adding finite binary approximations.

15. Can the Grouping of a Calculation Change Its Answer?

In ordinary mathematics, a long addition can be grouped any way you like without changing the result:

```
(A + B) + C  equals  A + (B + C)
```

Inside a computer, that guarantee is no longer perfect. Because each intermediate result must fit into a finite representation and may be rounded, changing the grouping can change where the rounding happens — and therefore change the final stored value.[11]

Assume an imaginary system with limited significant-digit capacity. Let:

```
A = 1,000,000        B = -1,000,000        C = 0.1
```

The exact mathematical answer of adding all three is 0.1. But consider two groupings.

Grouping One

First calculate A + C. The exact result is:

```
1,000,000.1
```

If the system cannot preserve the final decimal digit at that scale — recall Section 6: the gaps between representable values grow with the value — it may store:

```
1,000,000
```

Then:

```
1,000,000 + (-1,000,000) = 0
```

Final stored result: 0.

Grouping Two

First calculate A + B:

[11] Associativity. The property lost in floating-point addition is associativity: (a + b) + c is not guaranteed to equal a + (b + c), because rounding occurs at each intermediate step. Floating-point addition of ordinary finite values remains commutative (a + b = b + a).

```
1,000,000 + (-1,000,000) = 0
```

Then:

```
0 + 0.1 = 0.1
```

Final stored result: 0.1.

The mathematical inputs did not change. Only the grouping changed. The different outcome occurred because rounding happened at different intermediate stages.

16. Do Thousands of Racing Processor Cores Make the Answer Random?

A GPU is designed to perform many calculations simultaneously, and a popular explanation says that this alone makes AI unpredictable: thousands of processor cores race along in parallel, they finish in a different order every time, and so the answer comes out different every time.

That explanation is mostly wrong, and it is worth being precise about why, because the real cause is more interesting.

A GPU does not normally produce a different answer merely because thousands of cores run at once. If the same calculation routine processes the same inputs using the same internal strategy, it will ordinarily reproduce exactly the same result, down to the last bit, run after run.[12]

Here is what the strategy involves. A single LLM request contains mathematical operations so large that the work must be divided into smaller pieces. A long sum, for example, might be divided into:

```
Group 1 subtotal        Group 2 subtotal
Group 3 subtotal        Group 4 subtotal
```

Those subtotals must then be combined. There are many mathematically valid ways to do it:

```
(Group 1 + Group 2) + (Group 3 + Group 4)
```

[12] Run-to-run determinism versus batch invariance. Individual GPU kernels (calculation routines) executing the same inputs with the same configuration are typically run-to-run deterministic; the widely repeated "concurrency plus floating point" explanation of LLM nondeterminism is, in the systems studied, not the primary cause. The Thinking Machines analysis attributes production nondeterminism primarily to lack of batch invariance: kernel dispatch and reduction strategies change with batch size, altering rounding order. Secondary sources of variation exist (e.g., atomic-operation ordering in certain kernels, automatic kernel selection, hardware or software version differences); the dominance claim is scoped to the inference systems and configurations studied.

or:

```
((Group 1 + Group 3) + Group 2) + Group 4
```

With exact arithmetic, every combining order gives the identical result. With finite-precision arithmetic — as Section 15 showed — the last few stored bits can differ between them.

So the crucial question is not "do the cores race?" It is:

```
What makes the software choose one dividing-and-combining
strategy on one occasion and a different one on another?
```

If the strategy never changed, the answer would never change. The next section identifies the everyday event that changes it.

17. What Changes When the Server Gets Busy?

AI providers process multiple requests together to use GPU hardware efficiently. This is called batching. A batch is simply a group of requests processed during the same period.

Each prompt remains logically separate — no request sees another's data. However, the size and shape of the overall workload can cause the inference software to choose a different strategy for dividing and combining the calculations associated with each prompt. The calculation may be:

- divided into a different number of sections;
- assigned to different work groups;
- or combined back together in a different order.

The mathematical operation remains equivalent. The intermediate rounding pattern changes. That can change the final few bits.

And here is the uncomfortable part: from an individual user's perspective, the batch is invisible and uncontrollable. How many other people happened to send requests in the same fraction of a second determines how busy the server is, which determines the strategy, which determines the rounding, which can determine the final bits of your answer. The other users are, in effect, the dice.

This is precisely what Thinking Machines Lab demonstrated in the systems it studied: varying server load, and therefore varying batch conditions, was identified as the primary source of the

user-visible variation in those inference systems — not racing processor cores.[13] This was the mechanism behind the Feynman experiment described at the start of this paper.

18. How Does One Changed Bit Become a Different Answer?

An LLM generates a response one token at a time. A token may be a word, part of a word, punctuation, or another text unit.

For every next-token decision, the model calculates a score for many possible tokens. For example:

```
Token A: 0.81234567891
Token B: 0.81234567890
```

The scores are almost identical. A microscopic change in the final numerical bits could produce:

```
Token A: 0.81234567889
Token B: 0.81234567890
```

Token B now has the slightly higher score. If the rule is to select the highest-scoring token, the model chooses Token B instead of Token A.

That first changed token becomes part of the text the model sees when predicting the next one. The later response may therefore follow a different path.

The chain is:

```
Different grouping of calculations
    -> different intermediate rounding
    -> different final bits
    -> a near-tied token ranking changes
    -> a different token is selected
    -> the rest of the response may diverge
```

[13] Run-to-run determinism versus batch invariance. Individual GPU kernels (calculation routines) executing the same inputs with the same configuration are typically run-to-run deterministic; the widely repeated "concurrency plus floating point" explanation of LLM nondeterminism is, in the systems studied, not the primary cause. The Thinking Machines analysis attributes production nondeterminism primarily to lack of batch invariance: kernel dispatch and reduction strategies change with batch size, altering rounding order. Secondary sources of variation exist (e.g., atomic-operation ordering in certain kernels, automatic kernel selection, hardware or software version differences); the dominance claim is scoped to the inference systems and configurations studied.

In the Feynman experiment, this is exactly what happened at token 103: a near-tie broke the other way, and 1,000 identical beginnings became eighty different endings.

19. What About the Variety We Ask For on Purpose?

The accidental variation described above should not be confused with deliberate variety.

An LLM assigns scores to possible next tokens. A setting commonly called temperature controls how much variety is permitted when selecting among them.

At a higher temperature, the system may deliberately choose a lower-ranked token to make its output more varied — useful for brainstorming or creative writing. At temperature zero, the system is instructed to always pick the single highest-scoring token.[14]

People therefore expect:

```
Same model + same prompt + temperature zero = same response
```

But if the underlying numerical scores differ slightly because of the execution strategy, the identity of the highest-scoring token can still change.

Temperature zero removes the deliberate dice. It does not automatically remove the accidental ones. That is why the Feynman experiment produced eighty answers with temperature set to zero: the deliberate variety had been switched off, and the variation that remained was pure engineering artifact.

[14] Greedy decoding. Selecting the single highest-scoring token at every step is formally called greedy decoding; temperature zero is its usual practical setting.

20. Should the Answer Depend on Who Else Is Using the Server?

Section 17 ended on an uncomfortable fact: the strategy — and therefore the answer — can depend on how many strangers happened to press Enter at the same moment. Put as a question, the fix suggests itself:

```
Should the answer depend on how many other people
happen to be using the server at the same time?
```

If the answer is no, then the strategy must be held fixed. The prompt must follow the same calculation structure regardless of the surrounding batch — engineers call this property batch invariance, and an inference system built to guarantee it is called batch-invariant. For example, the system may require that:

- values are divided into the same-sized groups every time;
- each group is combined internally in the same order;
- partial totals are merged through the same fixed sequence;
- the model's attention calculations use the same splitting rules;
- and exact ties are resolved by the same consistent rule.

The physical GPU cores performing the work do not need to be the same cores each time. What must remain fixed is:

- which values are grouped together;
- where rounding occurs;
- and in what order partial results are combined.

Thinking Machines Lab demonstrated this in practice: with batch-invariant calculation routines in place, all 1,000 repetitions of the Feynman prompt produced the identical completion under the tested configuration. The routines were released publicly as open-source software; the approach was demonstrated on one widely used open-source inference engine and was subsequently integrated by another.[15]

[15] Adoption. Thinking Machines released its batch-invariant kernels as an open-source companion library (github.com/thinking-machines-lab/batch_invariant_ops), demonstrated deterministic inference under vLLM, and the SGLang project subsequently integrated the approach for deterministic high-throughput inference: LMSYS Org, “Towards

The cost is reduced scheduling freedom. The fastest arrangement for a particular workload may not be used if it would change the calculation structure. Repeatability — the same input reliably producing the same output, every time[16] — therefore comes at a price. It introduces a trade-off among:

- repeatability;
- processing volume;
- response speed;
- hardware utilization;
- and cost.

These particular dice, in other words, are not a law of nature. They are a performance optimization — one that can be traded away when repeatability matters more than raw speed.

21. If the Arithmetic Is Repeatable, Is the Business?

Even perfectly repeatable model arithmetic does not automatically guarantee a dependable business outcome.

A software system may still be affected by:

- changing external data;
- time and date conditions;
- simultaneous database activity;
- network failures;
- duplicated requests;

Deterministic Inference in SGLang and Reproducible RL Training," September 2025 (lmsys.org/blog/2025-09-22-sglang-deterministic/).

[16] Scope of the claim. The precise term for the phenomenon this paper addresses is inference nondeterminism: identical inputs to an inference system yielding non-identical outputs when no deliberate randomness is requested. This is distinct from (a) deliberate stochastic sampling, (b) semantic ambiguity in natural language, (c) changes in context, data, or external state between runs, and (d) model or infrastructure version changes. The paper's claims about removability apply to (batch-sensitive) inference nondeterminism, not to AI output variability in general. The property that the same input always produces the same output is formally called determinism; the main text deliberately uses the everyday word repeatability instead.

- policy changes;
- human approvals;
- authorization state;
- or third-party system behavior.

And before any arithmetic runs at all, there is language. Two customers who want the same thing will phrase it differently. A system that interprets "cancel my last order" and "undo that purchase from this morning" as different requests has a dependability problem no amount of numerical repeatability can fix.

It is therefore useful to ask four separate questions of any AI system that acts on a business's behalf:

```
Did it understand what you meant?
Did it compute the same answer?
Did it execute the same business process?
Did the customer receive the same result?
```

Each question is a separate dimension of dependable behavior, and each must be engineered separately — they are not four boxes wired in a row. Engineers would label them the semantic, numerical, execution, and outcome layers; the questions matter more than the labels. Figure 1 shows all four, and where the industry stands on each.

Layer	Question	Status
Semantic	"Did it understand what you meant?"	Open problem
Numerical	"Did it compute the same answer?"	Solved — batch-invariant inference
Execution	"Did it execute the same process?"	Open problem
Outcome	"Did the customer get the same result?"	Open problem

Figure 1. Four questions to ask of any AI system that acts on a business's behalf. Only the second — same calculation, same answer — has a demonstrated general solution today.

Batch-invariant inference addresses only the numerical layer. A business deploying AI agents to take real actions — placing orders, moving money, changing records — needs all four. Each layer must be engineered deliberately; none arrives for free, and none substitutes for the others.

Of the four, the numerical layer is the one this paper has traced in detail — and, as Section 20 showed, it is the one the research community has demonstrated how to solve. The other three cannot be solved inside the model, because they do not live inside the model. Meaning must be resolved before the arithmetic can be trusted to matter; execution and outcome must be governed after it. They require engineering that sits between the AI agent and the business systems it acts upon.

Surprisingly, only one of these four questions has a generally demonstrated answer today. For the numerical layer — did it compute the same answer? — the remedy exists: fix the calculation structure, and one thousand runs produce one answer. The other three are not merely unsolved. Each is documented, in the published research and standards literature, as a recognized open problem.

For the semantic layer, a substantial body of published research shows that today's models routinely give different answers to differently worded versions of the same request. Measured effects are not subtle: studies report performance swings of up to 45 percentage points across rewordings that mean the same thing, and comparable instability from formatting changes alone.[17]

For the execution layer, the most widely used benchmark of tool-using AI agents measured exactly the property a business cares about: not whether an agent can succeed once, but whether it succeeds every time. A state-of-the-art agent that completed a retail customer-service task successfully about 61 percent of the time on a single attempt completed the same task successfully on all eight repeated attempts less than 25 percent of the time. The benchmark's authors concluded that methods are needed to make agents act consistently and follow rules reliably — a conclusion, not a solution.[18]

---

[17] Semantic-layer evidence. Paraphrase and prompt sensitivity is extensively documented. Representative findings: Cao et al. (2024) measured performance swings of up to 45 percentage points across semantically equivalent prompt formulations; Sclar et al. (2024) report few-shot accuracy swings of up to 76 percentage points from prompt-formatting changes alone; Hager et al. document 8-50 percent performance variance from phrasing and 5-18 percent from information ordering in clinical settings. Survey and measurement work (e.g., Errica et al., 2025; the PROSA framework) treats paraphrase inconsistency as a recognized, unsolved reliability issue rather than a solved one.

[18] Execution-layer evidence. Yao et al., "tau-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains" (arXiv:2406.12045, 2024) introduced the pass^k metric — the probability that an agent succeeds on all k independent trials of the same task. Reported results: a GPT-4o function-calling agent achieved approximately 61 percent single-attempt success (pass^1) on the retail domain and approximately 35 percent on the airline domain, with retail pass^8

For the outcome layer, the United States national standards body itself states that there is not yet consensus on standardized, verifiable methods for measuring the risk and trustworthiness of AI systems, and describes AI measurement as a developing field. Subsequent research has argued that today's mainstream agent evaluations are structurally unable to measure reliability at all, because they score single attempts rather than repeated behavior.[19]

Every organization deploying AI agents to take real actions is exposed to all three, whether it has examined them or not. They are, at the time of writing, open engineering problems — recognized, measured, and unsolved.

22. How Do We Get From a Single Bit to a Different Answer?

The complete conceptual chain is:

```
Bits
    -> available storage positions
    -> allocation among sign, scale and meaningful digits
    -> numerical range and precision
    -> finite precision requires rounding
    -> intermediate rounding depends on grouping
    -> the software's grouping strategy can change with the workload
    -> the final bits can differ
    -> a near-tied token decision can change
    -> the generated response can diverge
```

This does not mean every LLM response will differ. It means the same question can occasionally produce a different answer because the computer grouped and rounded the underlying calculations differently — a possibility that sits in the engineering, not in the mathematics.

falling below 25 percent. The authors state the findings "point to the need for methods that can improve the ability of agents to act consistently and follow rules reliably."

[19] Outcome-layer evidence. NIST AI 100-1, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)" (U.S. National Institute of Standards and Technology, January 2023) identifies the lack of agreed-upon, verifiable metrics for AI risk and trustworthiness as a principal measurement challenge, and characterizes AI measurement as a developing field. Subsequent research on long-horizon agents (e.g., "Beyond pass@1: A Reliability Science Framework for Long-Horizon LLM Agents," 2026) argues that prevailing single-attempt benchmarks are structurally unable to measure reliability, reinforcing that whole-system reliability measurement against a defined tolerance has no accepted standard method today.

23. So — Can You Trust an AI to Do the Same Thing Twice?

The question on the cover of this paper now has an honest answer: not automatically. For one of the four layers — the calculation itself — the answer can be made yes, by deliberate engineering. For the other three, nobody can yet make it yes for you.

How we arrived there is worth restating in five steps. Artificial intelligence is built on ordinary computer arithmetic operating at extraordinary scale.

The foundation is simple:

- every value receives a finite number of storage positions;
- those positions must describe the sign, scale and meaningful digits;
- the available meaningful digits determine precision;
- finite precision requires rounding;
- and rounding can make the grouping of operations matter.

LLMs perform enormous numbers of floating-point multiplications and additions involving positive and negative fractional values. These operations are divided across parallel hardware and later recombined. When the workload changes the grouping, the last few numerical bits can change. Most of the time this has no visible consequence. Occasionally it changes which token receives the highest score — and the answer diverges.

Einstein's dice make a fitting last image, provided we are precise about what the experiment actually demonstrated. It demonstrated that one important source of AI variability — arithmetic whose grouping shifts with server load — is not fundamental to the model at all. It is a consequence of implementation choices, and under controlled conditions it can be engineered away completely: one thousand runs, one answer.

That does not make every AI system automatically dependable. Deliberate variety settings, genuinely ambiguous language, changing data, external services, and shifting business state remain separate sources of variation — and each must be controlled at its own layer, from the numerical layer this paper has traced in detail, through the semantic, execution, and outcome layers of Section 21.

The central lesson is not that computers are unreliable. It is that reliability must be engineered, layer by layer:

```
Storage determines precision.
Precision determines rounding.
Grouping determines where rounding occurs.
Repeatability requires controlling that grouping —
and then controlling every consequential layer that follows.
```

AI plays dice today. It does not have to play them at every layer tomorrow.

About Kachyng

Kachyng is building Agentic Commerce Infrastructure: the identity, orchestration, and settlement rails that let autonomous AI agents transact on behalf of humans and enterprises. Its patent-pending IDX + KYA identity layer gives every agent an independent cryptographic identity bound to a human controller — with scoped delegation, runtime behavioral trust, and global revocation in seconds. The architecture is protected by a patent family of 31+ assets with priority dates spanning 2010 to 2026.

Proof materials — production walkthroughs, architecture diagrams, filed patent claims, and customer and processor evidence — are available under NDA. Contact us to schedule a briefing.

Kachyng Inc. · 535 Mission Street, San Francisco
kachyng.com · info@kachyng.com