The "Software = Code + Data" 3.0 Era

Daniel Huang, Base26

June 2025

1 Introduction

Technology—from the software on our computers to the artificial intelligence (AI) and robotics powering a Waymo car—is fundamentally reshaping our lives. We can now have complex conversations with Large Language Models (LLMs) and ride through San Francisco's winding streets without a human driver. Yet the promise of technology hinges on converting the unruly, unpredictable physical world into digital data that machines can understand. But can the real world ever be truly digitized? And how do today's advances build on century-old principles of computer science? We will structure our exploration as a series of questions.

1. *What underpins digital technology* (Section 2)? We review *Turing Machines*—the simple yet powerful theoretical devices that define computation—laying the groundwork for digital information processing at a speed and scale impossible in the physical world.

2. *Are we creating digital twins or digital zombies* (Section 3)? We examine why our "digital twins" are often incomplete reflections of reality—more like "digital zombies"—by exploring three key challenges: discretization, complexity, and completeness. This is why one could say that "software is eating the world," albeit with some indigestion.

3. *How is AI different from conventional software* (Section 4)? We explore the idea that conventional software and AI occupy opposite ends of the `Software = Code + Data` spectrum. Whereas conventional software is 99% code and 1% data (i.e., code heavy or `Software 1.0`[1]), AI is 1% code and 99% data (i.e., data heavy or `Software 2.0`[1]), opening fundamental new possibilities for solving problems.

4. *What does the future hold for technology* (Section 5)? We imagine how the emergence of LLMs points to a `Software = Code + Data 3.0`[1] era, enabling us to creatively explore the messy middle between code-heavy and data-heavy systems.

[1]We are borrowing Andrej Kaparthy's terminology: `Software 1.0` is code-heavy, `Software 2.0` is data-heavy software, and `Software 3.0` is the spectrum in between enabled by the emergence of LLMs.

2 The Nature of Digitization: From Messy Reality to Perfect Rules

What underpins digital technology? To answer our first question, we must begin at the very foundation of modern computing. The journey from a physical world to a digital one starts not with silicon chips, but with a theoretical device conceived nearly a century ago.

2.1 Turing Machines: What is a Computer?

This device is the *Turing Machine* (TM). Conceived by the British mathematician Alan Turing, the TM is a device designed to study computation. Turing developed this idea in 1936, nearly a decade before John von Neumann would outline the practical architecture of a digital computer—*the von Neumann architecture*—consisting of a central processing unit (CPU) and memory.

A TM consists of three core components. First is an *infinite piece of graph paper* (i.e., *tape*) that serves as the machine's scratchpad (memory). Each individual cell is labeled with its coordinates. Second is a *stylus* (i.e., *head*), like a pencil with an eraser, that can read a symbol from a cell, write a new one, and move in the coordinate plane of the graph paper. Finally, there is a finite set of instructions—a *rule book*—that dictates the stylus's actions that serves as the machine's *code* (i.e., program). An instruction, for instance, might state: "If the current cell is blank, write the symbol `1` and move one cell to the right. Then, proceed to instruction `#42`."

A *computation* is the mechanical execution of these instructions, starting from instruction `#1`, until the machine encounters a `HALT` instruction. The input and output of a TM computation can be written on designated parts of its scratchpad.

It seems almost absurd that this simple device—which you could implement with a long strip of paper and a very patient rule book follower—captures everything from rendering video games to solving protein folding. After all, can this simple and mechanistic process truly define the entire universe of computation? As it turns out, the *Church-Turing* thesis answers this in the affirmative. It states that anything that can be effectively computed by any means can also be computed by a TM.[2] Another way to frame this is that the choice of programming language does not matter, at least theoretically speaking. Thus, all roads lead back to paper, pencil, and a rule book.

2.2 The Universal Machine: Code = Data

Since we described a TM as a physical machine with a specific rule book, we might wonder if we need to build a different machine for each task. Fortunately, we do not have to, thanks to the idea of a *Universal Turing Machine* (UTM).

[2] At the risk of getting somewhat technical, this is the set of (partial) functions $\Sigma^* \rightharpoonup \Sigma^*$ where Σ^* is the set of finite words over a set of finite symbols from Σ.

A UTM is a TM that can simulate any other TM. Its existence relies on the following observation: the description of any other machine—its rule book—can be written onto the tape and fed to the UTM as input. The UTM then reads this data—another TM's code—and simulates the machine it describes. In other words, the UTM treats the rule book not as fixed hardware, but as software—data that can be executed as code. This establishes a simple yet profound proposition:

$$\texttt{Code = Data}.$$

This equivalence is what makes a computer a general-purpose tool. A single piece of hardware implementing a UTM can execute any program because its code can be treated as data for it to execute.

2.3 Computability: What a Computer Can (and Cannot) Do

The principle that `Code = Data` is stunningly powerful. It might even lead us to believe that computation is limitless, since each rule book can be arbitrarily complex. However, a startling realization emerged alongside the very theory of computation: some problems, while clearly describable, can never be solved. Such problems are called *undecidable*.

The most famous of these is the *Halting problem*. The problem asks: is it possible to write a single program that can analyze the source code of any other program and predict with absolute certainty whether that program will eventually halt or run forever given an input? The impossibility of solving the Halting problem relies on the paradox of self-reference enabled by the equivalence of code and data.

To see why, imagine such a program, `Halts`, existed. We could then use it to construct another mischievous program called `Paradox` to demonstrate that such a program can only exist in our imagination. `Paradox` is designed to do the exact opposite of what `Program` would do when fed itself as input—remember that `Code = Data` so we can both treat `Program` as both code and input data. If `Program` halts on its own code, `Paradox(Program)` enters an infinite loop; if `Program` loops on its own code, `Paradox(Program)` immediately halts.[3] The logical trap is sprung when we ask the critical question: what happens if we run `Paradox` on itself, i.e., `Paradox`(`Paradox`)?[4]

1. If `Paradox` is supposed to halt, its own logic dictates that it must loop.

2. If `Paradox` is supposed to loop, its own logic dictates that it must halt.

This impossibility forces us to conclude that our initial assumption was wrong.

[3]This technique is a variation of *diagonalization*. Given an infinite table whose rows and columns are labeled by programs, and cells indicate whether the program in that row halts on the input given by the column, `Paradox` does the opposite along the diagonal of this table.

[4]The intersection of the `Paradox` row and `Paradox` column in the infinite table reveals a contradiction.

The undecidability of the Halting Problem carries an essential message: computation has absolute, knowable limits. Consequently, even before we get to practical concerns like speed or memory, we find that some problems by their very nature elude digitization.

3 Digital Twins or Digital Zombies?

Armed with the fundamentals of TMs, we can now examine *digitization*: the construction of digital models of everything physical from maps of cities to simulations of natural processes. However, the very act of translating reality into the discrete symbols of a computer forces a series of compromises—compromises that can corrupt the model, raising a critical question: *are we creating digital twins or merely digital zombies?*

3.1 Discretization Challenges

The first challenge we encounter when translating real-world phenomena into digital form is *discretization.* At its core, a computer operates on discrete symbols, like `0` and `1`. By contrast, the world we aim to model is often continuous. Consequently, we must translate these real-world objects and phenomenon into symbols. This problem might seem trivial at first, but consider a simple object: a one-foot by one-foot floor tile.

While its side length is easily represented by the number 1, its diagonal measures $\sqrt{2}$. How can a computer store this number? The problem is that $\sqrt{2}$ is irrational, possessing an infinite, non-repeating decimal representation (1.41421356...). We are immediately forced to make a choice: we could truncate the number, storing only a finite number of decimal places. Or we could include a symbol for $\sqrt{\cdot}$ and write additional software to handle computations with this extra symbol.[5]

While the truncation approach for a floor tile would likely be harmless in an e-commerce application, the same limitation becomes critical in scientific computing, aeronautics, or financial modeling, where the smallest rounding errors can cascade and compound. Thus, this simple example reveals a fundamental crack in the concept of a perfect digital twin. The computer must work with an approximation, not the true value.

3.2 Completeness Challenges

Discretization forces us to accept that our digital models are approximations. However, a second, deeper challenge arises when digitization forces us to confront undecidable problems. This may seem like a purely theoretical concern, but as it turns out, our smartphones navigate this challenge every day.

[5]There is a third option: work with computable reals encoded as infinite streams of numbers. However, the computability and complexity theory is less standard and nice. In particular, the Church-Turing thesis no longer holds.

A phone's operating system (OS) constantly manages other programs—the browser, email client, and camera app. Occasionally, some apps may hang. What should the OS do? In truth, the undecidability of the Halting problem demonstrates the OS cannot know for certain whether an app is in an infinite loop or just performing a very long calculation. The system doesn't try to solve this unsolvable problem. Instead, it has an escape hatch: you, the user. When a program becomes unresponsive, you act as an external agent, deciding to force-quit the application or restart the phone.

This need for a human "escape hatch" in computing points to a deeper truth about the limits of formal systems. Our solutions for complex, real-world challenges are often necessarily incomplete. This insight is formalized by *Gödel's incompleteness theorems*. In the 1930s, Kurt Gödel proved that any sufficiently powerful mathematical system[6] will contain true statements that cannot be proven within that system.[7] Just as our OS needs an external user to handle a hanging app, formal systems often require a perspective outside the system to grasp their full truth. In our quest to model the world, we inevitably create digital and logical abstractions that, by their very nature, cannot be perfectly complete.

3.3 Hard Complexity Challenges

Even if a problem can be perfectly digitized and is decidable—that is, solvable in principle—it may still be impractical if the answer takes a billion years to compute. This is where *complexity theory* comes in. It provides a framework for distinguishing between problems that are computationally "easy" and those that are "hard".

Problems considered easy belong to a class called `P` (for polynomial time). Many problems considered hard, however, belong to a class called `NP` (for non-deterministic polynomial time), the hardest being called `NP-complete`.[8] The key difference can be understood by comparing the act of finding a solution to the act of verifying one. Consider a complex Sudoku puzzle: trying to find the solution could take many hours and/or hints. Yet, if someone hands you a completed puzzle, you can verify that it is correct in minutes. This gulf between the difficulty of finding a solution and the ease of checking one is the core intuition behind the `P` versus `NP` question, one of the most famous open problems in computer science.[9]

[6]The canonical system is Peano arithmetic which describes arithmetic on natural (i.e., counting) numbers.

[7]Peano arithmetic is powerful enough to encode the operation of TMs since each TM's description can be encoded as a number (e.g., with Gödel numberings). This is yet another version of `Code = Data`.

[8]The introduction of complexity theory enables us define the *strong Church-Turing* thesis: all other models of computation can be *easily* simulated by a TM, i.e., in time `P`. Quantum computation may provide a counter-example to the strong Church-Turing thesis.

[9]While the question remains open, virtually every computer scientist believes that `P` does not equal `NP`, matching our experience that finding a solution is more difficult than verifying one.

Unfortunately, many of the most valuable real-world optimization problems—in logistics, finance, engineering, and drug discovery are `NP-complete`. For these problems, we cannot find optimal solutions in any reasonable amount of time. We must instead rely on approximations, heuristics, and “good enough” answers. This computational hardness means that even a perfectly discretized model can become a “digital zombie”—a model that looks right but is too complex to manipulate meaningfully.

3.4 Easy Complexity Challenges

But the challenges don’t stop with problems that are theoretically hard. Consider those that are, in principle, easy. A prime example comes from computational chemistry, where determining the ground state energy[10] of a system of atoms—a task that could help revolutionize materials and drug discovery—is known to be computationally hard.[11] To overcome this, scientists found clever workarounds: approximation algorithms that complexity theory considers easy because they run in polynomial time (class `P`). The catch: easy in theory can still be impossibly hard in practice.

To get a feel for the complexity of `P`, suppose after a long dinner, you forget where you parked your car on a street with N parking spaces. In the worst case, you must search all N spots—a linear, one-dimensional search. This is inconvenient but manageable. Now, let’s extend this to a two-dimensional parking grid with N rows and N columns. The worst-case search is now N^2 spaces. Finally, imagine a multi-story parking garage with N floors, each with an $N \times N$ parking grid. The search balloons to N^3 spaces. This already feels practically impossible for a large N. However, from a theoretical standpoint, all three scenarios—with runtimes searching for your car are proportional to N^1, N^2, and N^3—are in `P`.

Some of the best approximation algorithms for the ground state problem have runtimes with high-order polynomials (like N^5 or N^7), making even the three-dimensional garage search seem trivial in comparison. These algorithms require supercomputers running for weeks, and yet, they are only producing approximations of the true answer. Alas, being theoretically easy is often times practically hard.

4 The Code-Data Spectrum

In spite of the theoretical limits established at the dawn of computing, technological innovation has marched relentlessly forward. Human ingenuity has yielded practical solutions—or at least clever workarounds—for many of these impossible, hard, and even deceptively easy problems. Yet many challenges still

[10]Nature prefers lower-energy configurations which are more stable.

[11]In fact, it is hard even for a quantum computer even though atomistic interactions are described by quantum mechanics. It belongs to class `QMA-complete`, the quantum analog of `NP-complete`. Additionally, class `BQP`, the quantum analog of `P`, is conjectured to be incomparable to `NP-complete`.

elude us. This brings us to the current technological moment: AI, which is enabling us to solve problems we could not solve before. This raises the question: *How is AI different from conventional software*? The answer is that AI is software, but it represents a radical departure in design.

4.1 The Two Faces of Software: Code and Data

The idea that `Code = Data` is profound but abstract. In practice, conventional software treats them as distinct components, a relationship better captured by the equation:

`Software = Code + Data.`

Here, `Code` represents the instructions written by engineers, while `Data` is the information the code acts upon, often stored separately in databases or files and accessed through queries. This framework is particularly useful for understanding *Good Old-Fashioned AI* (GOFAI), the dominant paradigm in the decades before the current AI revolution (i.e., deep learning). The approach involved domain experts collaborating with engineers to translate their knowledge into programmatic rules. The resulting AI was effectively 99% code and 1% data since the richness of the expert's experience had been distilled into logical rules.

The chess program Deep Blue is a powerful demonstration of this approach. Grandmasters worked with scientists and engineers to distill their expertise into a sophisticated evaluation function, a complex piece of code that could score the strength of any given board position. Paired with brute-force search, this method was sufficient to defeat world champion Garry Kasparov in 1997. Deep Blue was the pinnacle of the code-heavy paradigm, but its victory also highlighted its limitations: its intelligence was narrow, handcrafted, and incapable of learning on its own.

4.2 The Data-Heavy Revolution

The deep learning revolution flipped the GOFAI paradigm on its head. Instead of using human experts to distill data into code, the new approach uses a small amount of code to allow the data itself to be "programmed" into the weights of a neural network. This relationship can be expressed as:

`Neural Network = Inference Code + (Training Code + Data).`

While GOFAI was 99% code and 1% data, this new approach describes software where the code is minimal and the data is large. Following Andrej Kaparthy, we might call this `Software 2.0`.

Deep learning-based algorithms have enabled solutions to many problems previously intractable with code-heavy approaches, including image classification, the game of Go, and protein folding. But what enables this approach to go beyond that of conventional software? On the surface, the answer is that deep learning provides a practical way to build data-heavy software. To go deeper, we can turn to information theory and its algorithmic counterpart.

4.3 Information

Information theory, first proposed by Claude Shannon in 1948, provides a mathematical framework for quantifying information. Its fundamental unit of measurement is the *bit*, representing the resolution of uncertainty between two equally probable outcomes, labeled `0` and `1`. The theory's central insight is that information is a measure of surprise or unpredictability. A predictable sequence of symbols contains very little information whereas a highly unpredictable sequence contains more.

For example, a uniform pattern such as `00000000` is highly predictable—its low probability of being anything other than a 0 at any given position means that observing the sequence does little to reduce an observer's surprise. Conversely, a less predictable pattern like `01101001` contains more information. Because each symbol in the sequence could plausibly be a `0` or a `1`, the final pattern presents a greater degree of uncertainty.

The binary nature of a bit is the foundation of the base-2 system used in all digital computing. While a single bit can only distinguish between two possibilities, sequences of bits can be used to represent an exponentially larger set of outcomes. For instance, in the 8-bit ASCII standard, the capital letter `A` is represented by the sequence of bits `01000001`—its codeword.

By assigning a codeword to each element in a set of possibilities, we produce a *codebook*: a complete mapping of symbols to their unique codewords. This codebook acts as a language which we use to describe patterns in data. The key insight is that a good codebook compresses information—patterns that occur frequently can be assigned shorter codewords since they are less surprising.

4.4 Algorithmic Information

What if, instead of a fixed codebook like ASCII, a program itself could serve as the codebook? *Algorithmic Information Theory* (AIT) offers this more universal perspective by merging information theory with computation. It defines the complexity of a sequence—its *Kolmogorov complexity*—as the length of the shortest computer program that can produce it.[12] A sequence of a million zeros has low Kolmogorov complexity since we can write a short loop to generate it. In contrast, a truly random sequence has high complexity, as its shortest description is the sequence itself.

However, true Kolmogorov complexity is incomputable—we can never be certain that a given program is the absolute shortest one possible. This is where the practical principle of *Minimum Description Length* (MDL) becomes invaluable. MDL asserts that the best model for a sequence is the one that minimizes the sum of two terms: the length of the description of the model itself—its codebook—and the length of the sequence when encoded using that model. The trade-off between these two terms is a direct embodiment of *Occam's Razor*: a simpler explanation is preferred over a more complex one.

[12]By the Church-Turing thesis, the choice of programming language does not matter.

While the precise theoretical underpinnings of deep learning are still an active area of research, the MDL principle provides a powerful lens through which to view it. The training process can be interpreted as a sophisticated search for a compressed representation of patterns in the data. The crucial insight is that a neural network's architecture and weights constitute its codebook. Just as a programming language determines what programs can be written, the learned codebook determines what patterns can be expressed, ideally forming an efficient "programming language" for the patterns inherent in the data.

This principle is clearly illustrated by comparing modern neural networks to earlier AI systems. Whereas Deep Blue's programmers painstakingly encoded human chess knowledge into an explicit codebook of rules, a neural network discovers its codebook implicitly from data. It must balance two competing pressures: having enough complexity to capture the underlying patterns in the data, but not so much complexity that the model itself becomes a lengthy, over-specified description. This perspective brings the exploration full circle, returning to the concept that Software = Code + Data—Software 2.0 does not eliminate code; it hides it in the weights learned from data.

5 The "Software = Code + Data" 3.0 Revolution

What does the future hold for technology? In this final section, we examine how the emergence of LLMs points to a `Software = Code + Data 3.0` revolution. We are now entering the messy, yet exciting middle ground between code-heavy and data-heavy systems.

5.1 Large Language Models

Large Language Models (LLMs), such as ChatGPT, are now used by hundreds of millions of people today. While the complexity of building LLMs are astounding, we can abstract their core components in our framework as follows:

```
LLM = ((Training Code + Transformer + Web) + Inference Code).
```

Consequently, an LLM can be described as 0.01% code and 99.99% data—an exceptionally data-heavy piece of software—due to its training on web-scale data. However, this description fails to capture the dynamic interactivity that makes these models so compelling. This quality stems from two key properties: their *auto-regressive* and *stochastic* natures.

The auto-regressive property gives LLMs a self-referential flavor. Once the model is *prompted* (<`bos`>), it produces an output token which is appended to the input sequence to create a new, longer input. This new sequence is then used to generate the next token, continuing until a stopping condition is met (<`eos`> token). The table below illustrates this step-by-step generation of the phrase `Software = Code + Data`.

time	output	input
0	`Software`	`<bos>`
1	`=`	`<bos>Software`
2	`Code`	`<bos>Software =`
3	`+`	`<bos>Software = Code`
4	`Data`	`<bos>Software = Code +`
5	`<eos>`	`<bos>Software = Code + Data`

This self-consuming loop, where the model's output is continuously appended to its input, is what creates everything from a single sentence to the illusion of a multi-turn conversation.

The second key property is that LLMs are stochastic. Their non-deterministic nature means that they do not produce the same output every time for a given input. That is, instead of deterministically selecting the next best token, the model rolls a weighted die to choose from a range of options. At first glance, the addition of this randomness may seem to take LLMs out of the realm of standard computation with TM. However, this is not the case.

The theoretical model for this is the *Probabilistic Turing Machine* (PTM), a TM that includes a dedicated tape pre-filled with the results of random coin flips. Whenever the machine needs to make a random choice, it simply reads the next dice roll from this tape. In practice, this random tape is simulated by a pseudo-random number generator—a deterministic algorithm that produces sequences that appear random.[13] Suffice it to say, a PTM adds no additional computational power to a TM. But it is precisely this managed non-determinism coupled with self-reference that gives LLMs their dynamic and interactive qualities.

5.2 What's Old is New

This interactivity is most apparent in how an LLM acts as a "meta-prompter", producing responses that are themselves prompts. This hints at a form of *meta-programming*—a program that operates on programs—and connects the newest technology to a concept in computer science that has been there since the beginning: `Code = Data`. What's old is new again, with a twist:

`Prompt = Data = Code`.

The most immediate application of using prompts to generate code is that it can aid human software engineers in developing software—vibe coding. The next logical step is to interface and interleave LLMs with conventional software—AI agents and other human-in-the-loop systems. More speculatively, can an LLM be prompted to improve its own implementation, thus modifying its own code? This leads to the tantalizing idea of recursive self-improvement.

Could this process continue infinitely? Our journey through the theory of computation provides a definitive answer: no. Just as data has a point of

[13]Pseudo-random number generators have many other practical applications, most notably in cryptography.

maximum compression, any program—including an LLM's own source code—is fundamentally limited by its own incompressibility. Yet, the gap between where we are today and that theoretical (uncomputable) boundary is immense, leaving a vast and unpredictable landscape for transformation.

The most profound innovations will not come from pursuing "more data" or "better code" in isolation, but from creatively exploring the spectrum between them. The journey from using "English as a programming language"—code written in base-26[14]—to self-improving systems shows that the lines between programmer and prompter, code and data, are blurring. We are venturing boldly into the `Software = Code + Data 3.0` era.

[14] For simplicity, we are referring to the 26 letters of the English alphabet, without regard to case, numbers, or punctuation.