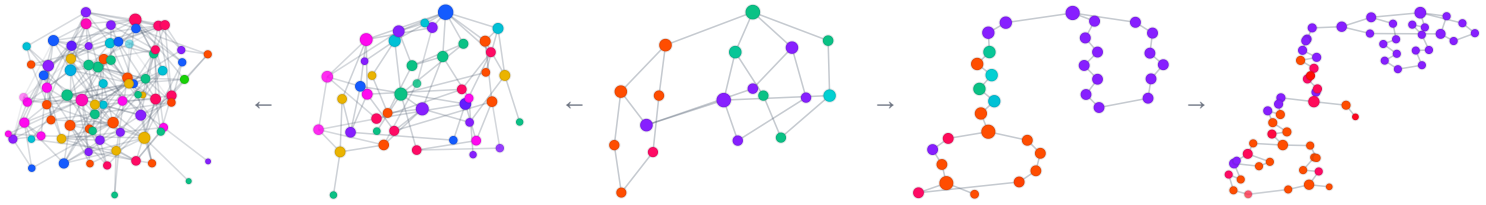
THE DECOHERENCE TRAP

OCTOBER 14, 2026

TL;DR

This essay introduces the concept of **decoherence**: the accumulation of locally valid but mutually incompatible or redundant models within a system, such that the cost of change is dominated by translating among those models and preventing unintended side-effects.



In other words, it's **what happens when slop drowns out the signal in an endless sea of noise.**

Decoherence grows relative to the rate of change of a system, and it increases faster the more decoherent a system is, resulting in runaway exponential growth. It is not a new phenomenon, and humans have always countered it through carefully finding ways to do more with less, unifying disparate patterns and models into a cohesive whole. However, the widespread use of AI is currently producing it at a rate that is unsustainable.

The good news is that this rules out the specter of runaway recursive self-improvement, the intelligence explosion that goes parabolic before anyone can react, as decoherence both prevents AI from improving in a bubble *and* limits the rate at which it can improve outside of it. The bad news is that we are barreling toward a different kind of disaster: **Decoherence Collapse.**

Imagine a building, let's say a small office tower. When this building was built, they used a type of pipe that will degrade over time and eventually need to be replaced. This is standard ***infrastructural debt***; basic wear and tear, as they say.

Then, let's say that when it comes time to replace the pipes, you need to change them over to a new standard. This could be because of new building code requirements, because the old pipes simply aren't made anymore, or

because the new pipes are much better and easier to maintain. This is standard ***technical advancement debt***, the need to upgrade and transition due to the forward progress of technology over time.

There is a third form of debt that can arise based on *how* the transition from old pipes to new pipes is done. Consider the following options:

1. **Shut down the building and replace all of the pipes at once.** This is a very disruptive and expensive method, since you can't use the building while the process is underway and you have to pay for the replacement up front. But the process is *complete* by the end of it.
2. **Shut down each floor, one at a time, and replace the pipes on that floor.** This is *less* disruptive overall, but somewhat *more* expensive because you still have to pay up front, and you have to pay a *premium* for the extra labor required to keep things running smoothly as the work goes on.
3. **Replace the pipes as needed, when they begin to fail.** This is the least disruptive strategy, and it allows you to amortize the cost of replacing all of the pipes over time, with one small repair after another.

For most people, option 3 is very appealing. It allows them to keep things running smoothly and to spread out the cost of maintenance. And if we can just make like the *Ship of Theseus*[1] and replace the pipes one at a time, why shouldn't we?

Well, for one thing, it creates ***decoherence***.

And decoherence is, by far, the *worst* form of tech debt.

EVOLUTION



The *Ship of Theseus* actually gives us a good lens for understanding the true scope of decoherence. We know how to build ships made of wood, and we know how to build ships made of steel. You can repair a ship one piece at a time with the same material, as the original thought experiment goes, and the ship remains functional, with all of the original designs and structures that went into it intact.

But do we know how to build every iteration of a ship *in between* wood and steel? Could you, say, slowly replace every piece of a man-of-war and incrementally turn it into a modern destroyer? And have it functional (for some definition of functional) at every step in the process?

It's not that the answer is an absolute "no". If we are creative enough, if we had the best minds working on it, then I'm sure there would be *some* path to doing this. But it would require *ingenuity* and *skill* and *craftsmanship*. It would require attention to detail. And, crucially, you would really want to *complete* the process, because the half-man-of-war, half-destroyer vessel is likely not going to be very good in modern combat.

Returning to our plumbing example, we can see how the same strategy plays out regularly in the world of construction. Every time a pipe breaks, the plumber is called, walls are opened, the repair is hand-crafted for that specific area, and then everything is closed back up. If you're lucky, the plumber will leave detailed notes, and if you're not, then years later some poor sod is going to open that wall back up and say to themselves, "What the *hell* is going on here?"

Now, multiply that process over time as we slowly replace the pipes in our building, and the end result is a patchwork of old and new pipe that has much more *complexity* than either system on its own. Every joint, every hand-crafted patch, is its own little bundle of unique forces and properties that are understudied, and thus requires an expert in *both* systems to fully understand and work with.

On the surface, this is fine. The pipes work, why does it matter *how* they work? As long as it's a good patch, who cares?

Well, what happens if we, say, want to build an annex?

For the purposes of this thought experiment, we'll say that the water for the annex needs to be routed through at least part of our existing building, using the existing pipes. And, crucially, we need to increase the *overall water pressure* in the system.

Now, the question is: **Is it safe to increase the pressure?**

If we had all type A or all type B pipes, even ones that had been patched and repaired over time, then this would be a straightforward question. Both systems are well understood, both have tools and processes for calculating their capacity, and people run these calculations all of the time.

But, if we have an unknown mixture of both systems, suddenly we have to do a lot more analysis to answer that question. Every patch could change the answer, even ones at the other end of the system, because they're all tied together.

And this is where the true complexity of decoherence becomes clear. When we try to make changes to decoherent systems, it requires far more expertise and skill than working with either system alone. The engineer working on the system needs to trace and understand it much more thoroughly and carefully, because their changes may have unintended consequences. (And yes, I count plumbers as engineers, at least as much as software devs. Have you seen them think through pressure forces in drain pipes in real time? It's remarkable.)

That is no small task, and it is best if it's not rushed. But for the outside observer, without an understanding of the depth of the problem, this can be frustrating. Normally this part is easy. Why is it taking so long this time?

In this situation, management will oftentimes apply pressure to get things done faster. When pressure is applied, the expert doesn't do the full analysis faster; they just stop earlier and do something they're *pretty sure* will work. A sloppy join, an unusual part, maybe some extra safeguards, just in case.

Not completing the task thoroughly, however, creates a knock-on effect: this particular change has lower quality, and it makes the system more fragile and complex. This means the next change will take even *more* thought and care, and if the pressure on the engineer is the same, then they will do an even *worse* job.

That is a pretty messy situation as it stands, and you can already see a sort of exponential growth of complexity taking place here. But wait, it gets worse! Let's consider the next level of decoherence: Starting a *second* transition before finishing the first.

THE MAZE



The tricky part about software, as compared to a physical building or other types of real-world systems, is that it's all sort of invisible.

Coders walk through the halls of codebases every day. We navigate the labyrinth successfully so often that we become familiar with every corner and every turn, and when our walks become longer, with more corridors and dead ends and trap doors, we *feel* it much more immediately and intuitively.

But this is difficult to relate to people who can't even see the maze, who don't walk it with you every day. And when it comes time to clean up some of your debt, to make the maze shorter and less winding, your proposals are hard to sell up the chain, even to those who used to walk mazes a long time ago. In the end, none of them know *your* maze. They don't know the trials you face each and every time you enter it.

By comparison, our plumber friend in the real world can show the building owner the absurdity of their pipes much more *directly*.



Courtesy of r/plumbing [2]

When we start to mix more than two systems together, with multiple patches over time in a haphazard way, the complexity of the maze *explodes*. It is now combinatorial: you need to think of A -> B, A -> C, B -> C, A -> B -> C, C -> B -> A, and so on.

We can of course insulate engineers from this somewhat. We can come up with best practices, we can use tools and frameworks to isolate part of the system from another part (for instance, by using a pressure-limiting valve in our pipes). Inevitably, though, it gets to a point where even the best engineers, the smartest people, can no longer safely navigate the maze.

They may be able to do it 80 or 90% of the time, but they will eventually take a wrong turn

forget to leave a trail of string and find themselves stuck

within the immensity

of the unending growth

that outpaces their every step

into eternity.

This is the Decoherence Trap. The maze grows onward and onward until it swallows you whole and leaves you lost in the dark. Until it becomes so massive that it would take lifetimes to navigate.

Well, lifetimes for humans anyway. But what about AI?

SIMULATION

Humans get tired and reach physical limits, but autonomous agent swarms can keep going around the clock, collectively and tirelessly mapping the maze like the good little robots they are. If scale is the problem, then shouldn't we just be able to throw more compute at it and call it a day?

Well, that depends on the nature of the complexity we are facing. Does it:

1. Grow linearly with the size of the codebase as a whole, OR
2. Grow *exponentially*, with each change increasing the likelihood that the *next* change will add even more complexity.

If it's the former, then yes, as the size of the problem grows we can just keep adding more agents and more data centers and working the problem at about the same pace. But if it's the *latter*, then adding more capacity will only get us the ability to push a bit further before the size of the maze threatens to consume all of the compute on Earth (and in space, should SpaceX's plans[3] work out).

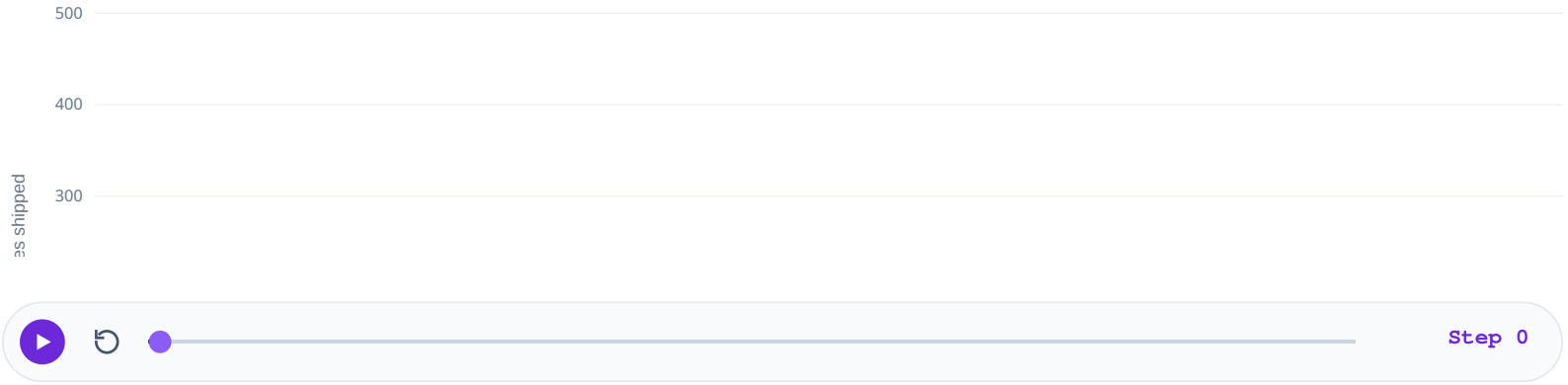
In my experience, it is the latter. When a plumber opens up a wall and sees a mess, it isn't clear what's going on. They may choose to use pipe A, they may choose to use pipe B, or they may choose to add a new type of pipe entirely in their confusion. The same is even more true for code.

We can see this play out in a simple simulation below. Here, we have a graph that represents a codebase (or some other complex system), where:

- **Nodes** represent different components within the system
- **Node Size** represents the size of the to component
- **Node Color** represents the dominant paradigm that the component belongs to. You can think of this as the "kind" of component - different materials of pipes, different constructs in code, different types of systems, etc.
- **Edges** represent connections between those components



START



This is a recording of a few runs. If you want to play with the full simulation yourself, changing the settings and watching what happens, you can find it here.

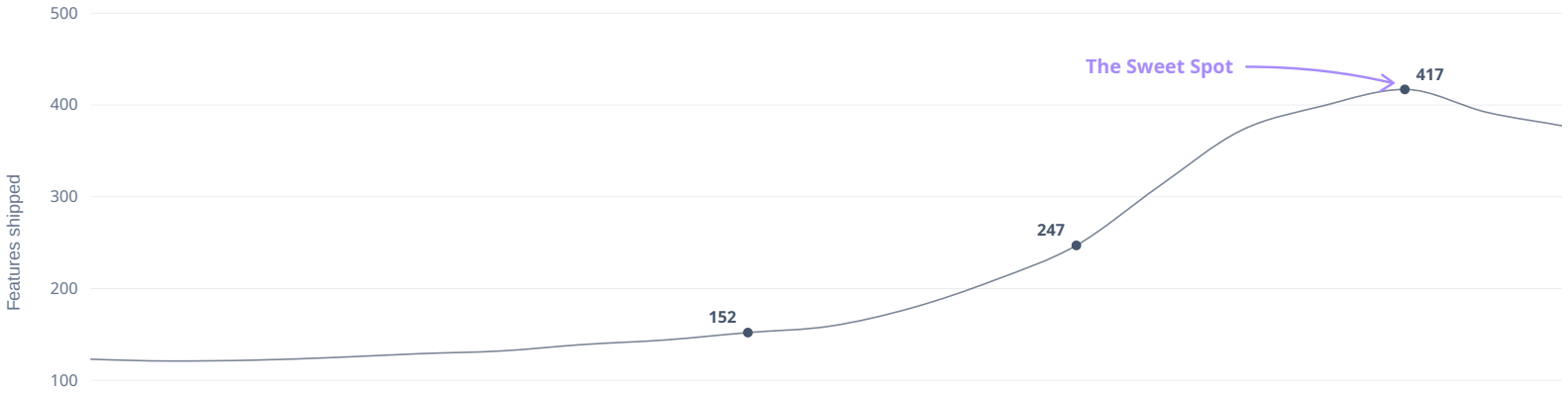
An actor, representing a developer, implements a "feature" by walking some random subsection of the graph and then mutating it. The actor's base behavior is shaped by its *coherence factor*:

- **Lower coherence** means the actor rushes through the problem more quickly and makes snap decisions, preferring simple solutions without considering the context around them, and overall adding complexity to the graph.
- **Higher coherence** means the actor takes longer and approaches each problem with more thought, and thus has a higher chance of making a less complex change or even removing complexity.

At first, the lower-coherence actor moves very quickly compared to the higher-coherence actor. However, as the graph changes, it impacts the walks as well. Each of the following slows down the actor's progress:

- **Higher connection degree (number of edges)**. This represents the added burden of having to pull in the context of each adjacent node, at least partially, in order to understand its impact on the current node.
- **Transitions between colors**. There's overhead when switching paradigms as you implement a feature, and it requires the actor to again gain context and understanding of each paradigm AND how they connect.
- **Path length**. The walk itself has a chance to extend further as there are more connections and transitions needed. This represents the actor needing to read more and revisit more in order to get all of the context needed to implement the feature, and it can increase indefinitely until you hit the *size of the entire system* (meaning, in order to make a change *anywhere*, you need to understand *everything*).

If we take the distribution of features built over time and coherence, we can plot the result and see a shape that should look familiar to any engineer: The **Sweet Spot**.



In the **Sweet Spot**, the system is relatively coherent and we actively invest in it to ensure it stays that way. There are a few dominant paradigms, but there's enough jitter that things can change and adapt over time, and we aren't overly perfectionist about everything.

Above the **Sweet Spot**, we work extremely hard to make sure every new node fits into the existing paradigms, and we remove complexity at every turn. This takes much more time and slows down feature work, but in the worst case it still produces a highly coherent system that can be iterated through *relatively* quickly.

Below the **Sweet Spot**, the drop-off is precipitous. We very quickly lose speed as the codebase becomes exponentially more complex. If the coherence factor is *negative*, and no care is given to maintaining coherence, then the time to ship a feature grows without any apparent bound.

We can also see how different strategies affect the system over time from different starting points. If we start with a highly coherent system and use a mostly decoherent strategy (a coherence factor of +0.2, where some coherence is maintained but shortcuts are still taken), then the graph devolves slowly at first before hitting some inflection points and entering the Trap.

By contrast, if we take a system that starts out decoherent and apply effort to recohere it over time, then we can move the needle toward coherence and accelerate.

Now, critics may note that this simulation has arbitrary weights and does not definitively *prove* that this is how complex systems will evolve. This is not my assertion.

What the simulation demonstrates is that this curve can exist as an *emergent* behavior of simple rules and local decisions. Actors changing the system today can improve or degrade the ability of future actors to change the system tomorrow. This is an obvious truth, but I think the implications of it can be quite easily lost if we can't actually *see* the complexity like we can in the real world, with pipes and wires and physical hardware.

So, we come back to the question on agents: Can agents work on these systems and maintain coherence? Or do they, over time, introduce decoherence even *slightly*? Because there is another crucial risk to consider here: decoherence does not grow over *time*, but with *change*.

If you build a decoherent system today, and then stop changing it for 10 years, and come back to it, it will be exactly as decoherent as you left it. It may have other forms of technical debt, but those are not related to decoherence. Decoherence grows or shrinks *only* when changes are made to the system.

And if you're making a lot of changes really quickly, you can also grow decoherent *just as quickly*. Even a very slight rate of increasing decoherence can spiral out of control before we even know it.

Here's the thing: We've been dealing with this phenomenon for much, much longer than agentic development has been around. Every single company I have ever worked for had massive amounts of decoherence, and less-experienced engineers and engineers under pressure also tend to add more decoherence over time, so it's not like humans are perfect here.

In fact, I would *venture to say* that most companies operate somewhere well below the **Sweet Spot**, and indeed operate in a continuous cycle where:

1. Decoherence grows to the point that no features can be completed in a reasonable amount of time.
2. A large effort is made to reduce decoherence.
3. The effort produces results and features begin shipping again at a slow-but-acceptable pace.
4. The effort is not completed, or if it is, it still doesn't push us all the way to the **Sweet Spot**. Resources and focus are shifted, and the system begins to decohere again.

Wash, rinse, repeat. Some companies may recohere faster and longer than others, and a very small percentage manage to stay in the **Sweet Spot** over time, but it requires a dedicated engineering culture of excellence (Apple comes to mind; say what you will about the state of macOS, I would take it any day over Windows).

The question is, are agents actually *better* than the average engineering culture at this point? Significantly better, at that?

So far we have not seen evidence of this. If anything, the early research points the other way.

A Harvard study[4] of 718 companies found that adopting coding agents raised the amount of code written by 30%, but did not produce a measurable increase in finished work. The extra code piled up in review instead: pull requests took 49% longer to review, and nearly twice as many were sent back for changes. A Carnegie Mellon study[5] of open source projects that adopted Cursor found a large jump in velocity that did not last, alongside an increase in code complexity that did, and that the added complexity was itself a major cause of the later slowdown. And an analysis of over 300,000 AI-authored commits[6] across more than 6,000 repositories found that about one in eleven of them, and up to one in seven depending on the assistant, introduced at least one quality issue, and that nearly a quarter of those issues are still sitting in the code today.

More code, the same amount of finished work, and more complexity left behind with every change.

PROGRESSION

Now, let's address the primary counterargument, which is the *technological progression* argument. It goes something like this:

> The first generation of agents will produce inferior-quality code, but as long as it's "good enough" then they can still replace humans for most coding tasks, and we can build our processes around

them in anticipation of more capable agents in the future.

”

This is effectively the story of automation during the Industrial Revolution, where the first generation of textile machines was indeed much worse than humans, but *eventually* they did catch up. As Cory Doctorow[7] has pointed out, industrialists raced to adopt automation because it allowed them to discipline labor and consolidate power at the expense of the workers and the public, who had to endure far worse working conditions *and* worse quality goods for generations.[8] [9]

But I want to build on this further, because there was a core reason that it worked for textiles: The machines produced inferior products, but they met a minimum bar.

The machines themselves were also very well designed, for what they were. I spoke recently to a mechanical engineer who extolled the quality of those generations of machines, how well they operated, how long they would last. Yes, they may have produced low-quality slop, but they produced it quickly and tirelessly and for many, many years, until the next generation came along and replaced them with incremental improvements. And crucially, while the machine was working, there was no increase in *decoherence*. Wear and tear, yes. Generational advances, yes. But not decoherence.

Now, here's the rub: The logic of this argument is assuming that *code* is analogous to *textiles*. That is, code is the final product that users will pay for. But nobody wants to buy code. They want to buy *applications*.

In other words, code is not the textile, but the textile-machine. Every time a screen is rendered, a form is submitted, a record is updated, that is the "product" that the code produces. This is why code has been so immensely profitable, because every additional customer requires no additional overhead. The raw materials are bits and bytes of electricity on chips, and the product is information and processes encoded into those bits.

A more accurate analogy would be if we somehow went back in time and made not just a textile-machine, but a machine that *designed* textile-machines. That iterated on the next generation of textile manufacturing on its own,

semi- or perhaps fully autonomously. This is something that has never existed before, and is unprecedented in the history of automation.

This is the dream of recursive self-improvement. Of an intelligence explosion, a technological singularity[10]. This is the dream of AI in the first place.

So, how has that been going?

SINGULARITY

Well, the goalposts have been *subtly shifting*. In May of this year, Tom Reed, a skeptic of the intelligence explosion theory, coined the term "the Goodhart Singularity[11]" to describe the limitations of AI iterating in a bubble without new data about the real world. This is based on Goodhart's law[12]:

> *When a measure becomes a target, it ceases to be a good measure.*
>
> — MARILYN STRATHERN, PARAPHRASING CHARLES GOODHART

In a post from last month, Tom Davidson, a senior research fellow at Forethought[13], conceded that data bottlenecks will slow down progress[14] once AI reaches what he calls the "human-quality ceiling": the point where there's no better human demonstration left to learn from, and AI has to manufacture higher-quality data than anything that exists. Davidson has written extensively about models of AI takeoff speeds, and later forecasts built on his work, most prominently the AI 2027 scenario[15].

So the fact that Davidson has wavered, if even slightly, is notable. But he argues it's a one-time slowdown, not a wall, and in general, those expecting a fast takeoff still think that it will be able to recursively self-improve on its own with relatively little data, and that this process could spiral out of control in isolation in a data center and lead to the birth of a super-intelligence.

The way they expect to be able to do this is via what they call a Software Intelligence Explosion[16] (SIE). Specifically, a subtle but load-bearing moment in the argument comes from Davidson's post on data bottlenecks:

> Rather than building RL environments that teach the construction of real-world software artefacts, AI must construct RL environments that teach AI to make better software than anything that exists in the real world.

> To be clear, I think this will be possible! It is easier to construct difficult challenges and verify their answers than it is to solve them. Already today, human experts build RL environments that produce data above the human-quality ceiling. Data quality will not hit a wall at the human ceiling.

Let's break this down.

What Davidson's saying here is that AI must be able to make better software than anything that exists in the real world, largely in isolation. To do this, it will need to be able to make a Reinforcement Learning environment[17] that can produce that software. The way this works is essentially:

1. Define a goal
2. Define a set of things you can do
3. Use an algorithm to try out a lot of different methods to accomplish the goal
4. Measure how close you get to accomplishing it, and how good your method is
5. Find the best method(s)

For an SIE to occur, the AI would need to be able to do this entire thing on its own to discover better and better *algorithms* for learning, and that's the important part. It's not about producing better software per se, it's about producing better algorithms for *learning and improving*.

The Goodhart Singularity effectively argued that an AI, in isolation in a data center, would not be able to do this for most real-world tasks of consequence due to the measurement step. If you want to figure out how to build a better car or rocket or computer chip, you might be able to build a simulation and try things out, but you would still need to test against the real world to validate your findings, or you would eventually drift further and further away from reality as you pushed the designs to more extreme places. This introduces a rate limit which would slow things down significantly.

But this isn't true for some domains, which is what Davidson is saying in the second part. For games like Go or chess[18], the rules of the game are the entire world and there are clear victory conditions, so it's possible to run in isolation and find better and better strategies, and also to find better algorithms for *finding* better strategies. The same is true for more complex games like StarCraft[19] or Dota[20].

But these are *games*, and we can do this because the victory conditions are clear and simple and the rules or environments are very limited. In order to get our explosion, we need to be able to apply these methods to *something that matters*, and we need to be able to *verify* that **in the bubble** of a data center, without checking against the outside world.

That brings us to software. I think that SIE proponents have really latched onto software because they see it as a way to bridge this gap. Software can run in isolation, and you can test it in isolation very easily and quickly. That's perfect for creating an RL environment, and they think that eventually, we will be able to build one that doesn't require syncing with the real world.

Because after all, it's *software*. It can do just about anything, right?

You can see how this would feel so natural to those in Silicon Valley. We pride ourselves on being very intelligent and being able to solve big problems. A significant portion of computer science education is all about logically proving properties of software and algorithms and the like. Will this program complete? Will this sorting algorithm be fast? How fast? Will this system be safe and not crash for all cases that could ever possibly exist?

We're used to thinking about problems in this way, and used to the idea that software has no limits. Tech took over the world, so it's only a natural extension of thinking that if you can teach a machine to write code, eventually, it will be able to teach itself to be better at writing code.

And then, eventually, the machine will take over the world too.

The problem with this viewpoint is that it fully misses what actually makes software *good*, which is coherence.

Allow me to elaborate. Software is a tool to accomplish a goal in the real world. That goal may be many steps removed, through many abstractions, but whether or not it succeeds is ultimately determined by how well it solves that goal.

You might think, ok, but once you have the software and it solves the goal, can't you use that as the output to target? And then the AI could just iterate on better ways to achieve those goals?

If we define "better" as something simple and easily measurable, like lines of code, speed, memory, or efficiency, then yes. These are all important, but if we treat them as the target, we'll end up with software that is small and fast and *completely decoherent*.

Code gets fast by skipping work, and it can only skip work by assuming things about the rest of the system: this list is already sorted, that value is never zero. Every one of those assumptions is a hidden connection that nobody wrote down. Optimized native code sits under abstractions for a reason: we need it for performance, so we seal it in a box where its assumptions can't leak. If we wrote all code like that, it would be a tangled mess. (A practical example in JavaScript is combining multiple logical classes into a single class to keep hot paths monomorphic[21]. It's always a huge pain, but it pays off as long as you never have to fiddle with it.)

That would be fine if we only had to write software once and never touch it again. But that's never true in practice. Because software *is* the never-ending *Ship of Theseus*. Applications and libraries and frameworks must evolve over time to meet changing conditions on the ground: real-world constraints that shift and evolve as we change how society works, as we discover more about the natural world, and as we push the limits of what we can achieve.

So good software is adaptable over time. You can take the existing technological capital (all of the software we have written before), and you can quickly recombine it to solve new problems on the fly.

And what makes it adaptable is *coherence*. Most of the big advances in computer programming and software quality have been about making software more coherent, and the only way to test if it is more coherent is with real-world feedback.

Let's break it down with a really concrete example: futures[22].

Futures make asynchronous programming easier in some really fundamental ways. Before futures, there were lots of ways you could handle async, and it was a predictably chaotic mess. Specifically, a decoherent one: many locally sensible ways of doing the same thing, none of which fit together.

Developers would have to live within that mess, and they would have to carefully and thoughtfully navigate between all of the different types of async that any one app might have. If you missed a step or forgot something, you would have difficult-to-track-down bugs and issues. And the code itself read like a horror show, as callbacks multiplied and as apps got more complex and had more and more asynchrony to handle.

Futures were first introduced as a concept in the 1970s, with the core idea being that they were a *placeholder* for a future value. The first concrete implementation was in MultiLisp, in 1985. But they didn't make it into the mainstream until the 2000s, when Java added them in 2004, and they didn't get popular until the 2010s when chainable promises were first introduced into JavaScript and C# shipped Task. [22]

There were two main reasons for this:

1. The first futures APIs worked, but were clunky. You could pass around a future, but when you tried to read the value it would block, which meant you had to be careful about when you did it. They also weren't independently chainable, so you couldn't really *fork* the future easily.
2. Futures introduce a decent amount of overhead. You have to create a wrapper object to track the state each time, rather than just storing a function pointer, for instance.

However, over time we found the APIs that worked well, worked out the kinks, and we realized that the overhead was worth it because it made it easier to reason about and combine larger systems (compute and memory getting cheaper didn't hurt either).

Bringing this back to AI.

In order to build better AI software, you need to be able to test if it is actually better. And importantly, as we mentioned before, we have to be able to measure if it's better **in the bubble**. We can't check outside of the bubble because that brings us back to being rate-limited by the world.

So, what can we check inside the bubble?

Well, we've narrowed it down to:

1. Games
2. Software
3. Mathematics

Games don't really get us that far, because they're not grounded in the real-world tasks we care about. So let's move on to software.

As we've established, good software is coherent. So, how do we measure coherence?

Well, you take an abstraction and see how well it holds up in different contexts, and how much the code has to bend to fit each one. Since we're not allowed to leave the data center, there are exactly two options for how we can do this:

1. **Apply it to all existing software.** This will only get us so far, because even if it works well for all of them, there's no way to know if it will work well for all *future* software. Callbacks and event loops weren't even common until GUIs became standard, and early web servers still used threads and traditional parallelism, so it wasn't clear how valuable futures would be until the entire set of programs that make up the web was discovered and explored. We would be overfitting to the programs we already know. The result can look perfectly coherent today, and then the first new problem that doesn't fit has to be patched around.

2. **Create new software.** This is something we can attempt to do in theory, but in practice we can't just generate plausibly useful programs to expand the training set. Once again, software is a tool to accomplish a goal in the real world, and if we don't validate its usefulness from time to time we could drift substantially from reality.

   Beyond this core issue, the other important test of coherence is how well an abstraction or concept translates over time, across a variety of different contexts and evolving requirements. A standard can look good on paper and even work well in one environment, like desktop computers, but then be completely inadequate for mobile devices or cloud-based systems at scale.

   You would not only need to generate plausibly useful software, you would also need to generate plausible evolution scenarios to see how it adapts, what happens when not everyone upgrades, when people (or agents) lag behind or decide to go in a different direction, when you have an *ecosystem* of systems that are a constant moving target.

You can get arbitrarily good at making the software we already have more coherent. Getting better at the software we *don't* yet have requires either (a) waiting for the real world to produce new software or (b) making new software, testing it in the real world, and waiting for it to evolve over time. Both of these options require round trips outside the bubble.

Putting it all together now:

1. We *can* make an AI that produces faster, smaller, and more efficient software beyond the human ceiling while staying inside the bubble.

2. We *cannot* make an AI that improves software *coherence* beyond the human ceiling without leaving the bubble, as noted above.

3. Because AI cannot check if the software is universally better without leaving the bubble, the AI cannot tell if it is actually improving its learning algorithms or if it's drifting.

4. We're right back at the Goodhart Singularity.

This puts the nail in the coffin for a fast, isolated SIE right off the bat. But we can also approach this from *another* angle, and this one also applies to *other* forms of intelligence explosions.

1. Any form of intelligence explosion depends on the AI researching and improving its own software or hardware, with each generation building on the improvements of the previous one.

2. Each generation will have some percentage chance of introducing decoherence into its systems, since the AI will be building or expanding systems beyond the frontier of its current understanding.

3. If the AI moves too quickly, it will compound decoherence faster than it can correct, leading to a stall or regression in its overall capabilities.

4. If the AI validates and verifies all of its findings, it will be rate-limited by its capacity to gather data from the world around it.

5. This cannot be addressed by better learning algorithms developed in pure isolation, because the AI has nothing to measure these against (see the first proof above).

We can see what the failure mode in point 3 looks like in the following chart, adapted from Eth and Davidson's model in their SIE paper[16]. Their projection has the growth curve going parabolic before month 5. The chart below keeps their model and adds one assumption: that the cost of each change doubles as accumulated work grows, which is what decoherence looks like in human codebases.

If this is true and AI cannot bypass decoherence, then ultimately AI is rate-limited by reality. It's not that it won't ever surpass us, but that it is a long way off from doing so in any meaningful sense, and even when it does it will still have to do all of the work that reality demands, constrained by the same physical and informational limits that govern us.

Davidson claims we could gather sufficient data easily by just recording real-world tasks:

> If the whole world worked together to make this happen quickly, it would not take long. For each job, 1000 people could record their screens and wear video cameras on their heads, and within 4 months AI would have 300 years' worth of experience.

This, fundamentally, underestimates just how complex the world is. If we expect a machine to be able to improve itself indefinitely, it can't just be good at moving a cursor on a screen or identifying objects in videos. It needs to develop deep insights into understanding the world itself. It doesn't need to experience it like a human, but it needs to be able to capture the same kind of information, the same perspective, to measure the intangibles and subtleties that cannot be captured by a screen recording and video feed alone.

It needs to be able to make connections between things that, at first, appear to have no relationship at all. Which brings us to our last topic.

MATHEMATICS

Mathematics is the other domain which could, in theory, be checked in the bubble by using verified proofs. If an AI can get better at math in a bubble, then it can develop a better learning algorithm overall, and then maybe the

SIE argument still stands.

The thing is, math is sort of *arbitrary*. We make a lot of decisions about how it works, and what is useful. For instance, the number zero was not always accepted as a valid number. Imaginary numbers took centuries to be accepted, and their four-dimensional cousins, quaternions, struck Lewis Carroll as so absurd that he is thought to have satirized them in the Mad Hatter's tea party[23].

These are examples of things that we take for granted, but which really *aren't* as obvious or as natural as we might think. They only gained wide acceptance when they were used in real-life practices, and when the outputs lined up with reality in meaningful ways.

In a similar way, Boolean algebra, the fundamental mathematics that underlies computing, was also considered an esoteric branch of mathematics for nearly a century. It was first published[24] by George Boole in 1847, but it had no practical applications, and it was studied as a curiosity of formal logic rather than a tool for serious engineering.

The importance of this math wasn't discovered until the 1930s, when Claude Shannon was working at MIT on the relay controls for Vannevar Bush's differential analyzer. He had previously encountered Boolean algebra in a philosophy class he took during his undergrad,[25] and realized as he was working on circuits that they matched up perfectly. His master's thesis, "A Symbolic Analysis of Relay and Switching Circuits,"[26] made the connection in 1937, and the rest is history.

Boole wasn't just inventing mathematics for the sake of it. He had an intuition that the properties he discovered, the system he made, would be important someday. He thought he was discovering the laws of human thought, and while that might have been somewhat misguided, it was interesting enough that logicians and philosophers studied it and passed it down, until Shannon found it and built on it.

Would Shannon have been able to find that connection in the age of AI? If he hadn't been standing in a room full of relays with a real problem in front of him, telling him where to look, and instead had been sorting through endless reams of proofs that were all locally valid and globally meaningless?

There are an infinite number of logical and mathematical systems you can define in a vacuum, and you can get extremely good at building and exploring these hypotheticals. Figuring out whether or not they matter is a different skill, and that is one that cannot be improved in the bubble.

Mathematicians are already worried about this dichotomy. In September, twenty-five Fields medalists, Terence Tao among them, signed a declaration[27] arguing that solving problems is only a stand-in for the real goal of mathematics, which is understanding, and that mass-producing true statements can erode the ground that new ideas grow from.

What happens when all of that ground is worn away?

COLLAPSE

The work of building coherence is a long and unrelenting process. It's the work of exploring every crevice and looking under every rock. Of getting down into the mud of reality and letting it seep into your boots as you grope around blindly, looking for all of the different ways that the world can configure and reconfigure itself around you, endlessly, without letting it swallow you whole.

It's not just about solving problems. It's about figuring out what problems we're trying to solve in the first place.

Of figuring out all of the problems we're trying to solve, and finding a single solution that solves each one without exploding into a labyrinth of complexity.

Which brings me to my last point.

There's been much said about Omarchy[28] recently. There are many who love it, and many who are, shall we say, skeptical of an agent-forward Linux distro. They have charged forward rather quickly into the space, and appear to be morphing from the original goal of being a clean set of defaults toward a full distro with a custom kernel[29].

Now, I want to be clear that experimentation at the edge of the ecosystem is important. Forking, exploring ideas, figuring out what works without constraints, and bringing your findings back to the community is a huge part of what makes Open Source Software work so well.

But I'm worried here.

I'm worried that somewhere along the line, we may have forgotten that we are all climbing the mountain together. That what has made us so successful and so resilient as a community, what has advanced computing so far so quickly, is the process of building coherence.

Back in the 80s, 90s, and all the way through the 00s, every company managed their own stack. Each server was its own little bundle of dependencies and scripts and OS versions and so on. You needed a team just to keep things running smoothly at your average company.

During the OS wars[30], that fragmentation was worse. Unix did not have a clear successor, and Windows was gaining ground with an entirely different paradigm. If Linux hadn't been built in that time, and captured the aging *nix ecosystem, where would we be today? (Yes, I'm aware that FreeBSD was a thing and could have taken its place. What I really mean is: what if there had been no free software movement at all?)

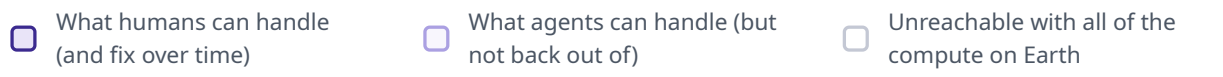
Would we be paying license fees every time we boot up a container in the cloud?

How many startups wouldn't have been able to get started, just because of the costs of running their stack?

That fragmentation was also the reason every system started to feel old and clunky over time. The entire ecosystem was decoherent, and much of the work over the last two decades has been about adding coherence (and capturing some of that value in SaaS products, but I'm getting ahead of myself).

I fear we may be practicing a sort of slash-and-burn agriculture here, where we are throwing away the coherence we have built in favor of rebuilding every dependency, customizing every piece of software, extending in every direction to solve short-term problems.

And not realizing that all the while, we're running deep into the trap.



If we push too long and too far without maintaining or rebuilding coherence, with current trends, we'll end up in a place where we *must* use agents to maintain our systems, because the systems will be so large and so

overwhelming that we cannot do so ourselves. But if we get caught there, and it turns out that agents cannot *back us out* by doing the repair work, we may find ourselves in a very uncomfortable position:

> Locked into a development model that slowly degrades our systems no matter how much effort we put into fixing them.

It's not that we won't be able to eventually fix the systems. But it might take years or even decades to do so, and in the meantime, we won't be able to ship features or make progress on the many other pressing problems we have in our society.

This is what I call **Decoherence Collapse**. And I am far more afraid of it than runaway super-intelligence, because it's actually happening right now, in front of our very eyes.

I do want to be clear that there is some utility in AI, I just think it's more on the margins than in replacing core workflows or running fully autonomously. I subscribe heavily to Cory Doctorow's reverse-centaur analysis[31] - we don't want the machine driving us, we want to be driving the machine. On that note, you can read my AI Usage Policy and Guide for a more detailed discussion on how I approach it in my own work and life.

And as I've said repeatedly in this piece, decoherence is not a new problem. If anything, it has existed since the beginning of human civilization, or perhaps even further back. It is a natural sort of decay or degradation, a type of entropy, that we have always dealt with and will always have to deal with. With that in mind, I want to leave you with two last thoughts.

1. Decoherence only ever changes when the systems themselves change. This means that no matter how bad it gets, we can *always* stop moving forward and start backing ourselves out. If AI has pushed us deep into the trap, it might take years or even decades to fully recover, but the hardest part is up front. Every part after will snowball, *as long as* we keep focusing on improving coherence and finish the job. If we improve it just enough to go back to our old patterns, we'll only be restarting the cycle.

2. We already have processes that are known to improve coherence over time. Humans have found ways to create more coherent systems, and those systems are in large part what has enabled the scope and scale of technological progress in the last several centuries.

In particular, the Open Source Software process that powers much of the coherence behind software development is *so successful* at producing coherence that it has outcompeted corporate software development in several truly stunning ways. But we'll discuss that more in the next essay: Open Capital.

GET NOTIFIED

If you liked this, you can subscribe for notifications when I post new essays.

SUBSCRIBE

CITATIONS

1. Ship of Theseus. Wikipedia.

2. Being picky with the plumbing work in my new house. r/Plumbing, Reddit.

3. Space-based data center. Wikipedia.

4. Fiona Chen and James Stratton. Artificial Intelligence in the Firm: Bottlenecks in Software Production. Working paper, Harvard University, August 4, 2026.

5. Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. 23rd International Conference on Mining Software Repositories (MSR '26), 2026.

6. Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo. Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild. arXiv:2603.28592, 2026.

7. Cory Doctorow. AI turns Amazon coders into Amazon warehouse workers. Pluralistic, May 27, 2025.

8. Clive Thompson. When Robots Take All of Our Jobs, Remember the Luddites. Smithsonian Magazine, January 2017.

9. Robert C. Allen. Engels' pause: Technical change, capital accumulation, and inequality in the British industrial revolution. Explorations in Economic History, 2009.

10. Technological singularity. Wikipedia.

11. Tom Reed. The Goodhart Singularity. Meagre Protestant History, May 7, 2026.

12. Goodhart's law. Wikipedia.

13. Tom Davidson. Forethought.

14. Tom Davidson. Data bottlenecks won't prevent an intelligence explosion (but they will slow it down). Forethought, September 8, 2026.

15. Daniel Kokotajlo and Eli Lifland. Takeoff Forecast. AI 2027, April 2025.

16. a b Daniel Eth and Tom Davidson. Will AI R&D Automation Cause a Software Intelligence Explosion?. Forethought, March 26, 2025.

17. Reinforcement learning. Wikipedia.

18. AlphaZero. Wikipedia.

19. AlphaStar (software). Wikipedia.

20. OpenAI Five. Wikipedia.

21. Vyacheslav Egorov. What's up with monomorphism?. mrale.ph, January 11, 2015.

22. a b Futures and promises. Wikipedia.

23. Melanie Bayley. Alice's adventures in algebra: Wonderland solved. New Scientist, December 16, 2009.

24. Boolean algebra. Wikipedia.

25. Jimmy Soni and Rob Goodman. A Mind at Play: How Claude Shannon Invented the Information Age. Simon & Schuster, 2017.

26. Claude E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. Master's thesis, Massachusetts Institute of Technology, 1937.

27. Terence Tao and 24 other Fields Medalists. A Severe Misalignment of AI in Mathematics. What's new, September 11, 2026.

28. Omarchy. Wikipedia.

29. Bobby Borisov. Arch-Based Omarchy 4.0.4 Makes Its Custom Linux Kernel the Default. Linuxiac, September 16, 2026.

30. Unix wars. Wikipedia.

31. Cory Doctorow. The Reverse-Centaur's Guide to Criticizing AI. Pluralistic, December 5, 2025.

© 2022–2026 KRISTEN MAEVYN

ESSAYS LICENSED UNDER CC BY 4.0