Digital Twins, or:
A Way to Separate the What from the How

We spend a lot of our effort describing *how* something works, and much less describing *what* it's supposed to do. Then we're surprised when it doesn't do what we wanted.

The code is the how. The tests are usually the how, too — they poke at the code and check that the code agrees with itself, but at least a subset of them should represent the whats. If you ask where the *what* lives, people might wave at a requirements document nobody has opened in six months, or at a Jira ticket, or at the shared mental model of two engineers who have since moved to different teams. Unless the what is formalized, *it is a rumor*.

A digital twin, properly understood, is a way to pin the what down. That's an odd claim, because when people talk about digital twins, they usually mean something industrial — a virtual copy of a jet engine or a factory floor, updated in real time with telemetry from sensors. The term was coined by John Vickers at NASA around 2010, and the underlying idea was formalized by Michael Grieves at the University of Michigan in 2002, though NASA had been doing versions of it since the Apollo program, when they built ground-based simulators to figure out how to keep Apollo 13 alive.[1] The canonical digital twin has a physical cousin. The point of the twin is to mirror that cousin closely enough that you can ask it questions the real thing can't safely answer.

But the deeper idea, the one that's almost hidden under the industrial marketing, has nothing to do with the physical world. The deeper idea is that a digital twin is a place to put the *what* — the invariants, the expected behaviors, the things that must be true — separate from the *how*, and *an outside-in view* of what a software system or service should do.

Why separation matters

Bugs come from conflating *the what* and *the how*.

When you write a test that calls your code and checks the result, your test knows too much about the implementation. It knows which function to call, what arguments it takes, what shape the return value has. It can't catch a whole class of errors, because it's made out of the same assumptions the code is. If the requirement was wrong — if you built the wrong thing correctly — the tests pass. If the requirement changed and nobody told the tests, the tests pass. A hundred percent coverage of the how tells you nothing about the what. *In fact*, a hundred percent coverage of the how *only* tells you that your code hasn't failed within the

boundaries of what you've tested. It says *nothing* about whether your test suite is complete.

What you actually want is something independent. Something that watches the system from outside and says: *these things must remain true.* Pre-conditions, post-conditions, invariants. The temperature never exceeds a certain value. The ledger always balances. The switch always forwards a packet within so many milliseconds. This is the vocabulary of the what.

A digital twin is the natural home for this vocabulary. You build a model that describes the thing you're making — not how you're making it, but what it is — and then you run the real thing and the model side by side. When they disagree, one of them is wrong. Often it's the real thing. Sometimes it's the model, and that's valuable too, because now you've found a gap in your understanding that would have eventually bitten you in production.

I once worked with a team building management software for a network switch. The hardware wasn't ready yet, which would normally mean the software team sits around drinking coffee. Instead, a really smart guy in the team built a switch simulator — a digital twin of the device we couldn't touch. The smart engineer went further, too: he wrote a tool that parsed the requirements documents and generated the simulator's state machine automatically. So the what lived in prose, the simulator was derived from the prose, and the production software had something to argue with. When production disagreed with the simulator, we didn't first ask which code was buggy. We asked which *requirement* was wrong.

That second question is the one almost nobody asks. It's the most valuable question in software.

The twin that isn't physical

Once you see the pattern, it shows up everywhere, and most of the places it shows up have nothing to do with factories or jet engines.

A type system is a kind of digital twin. It's a lightweight model of the program that says what must remain true about values as they flow through the code. The compiler is the thing that checks whether the how agrees with the what.

A database schema with constraints is a kind of digital twin. The constraints are the invariants; the rows are the runtime behavior; the database refuses to let them diverge.

A well-written specification — the sort people used to write at NASA and don't usually write anymore — is a digital twin. TLA+ models are digital twins.[2] Property-based tests are digital

twins, more than unit tests ever were, because they describe the shape of correctness rather than enumerating examples of it.

Even an organization can have a twin. This is where it starts to get interesting. If you can write down what a team is *supposed* to produce, how it's *supposed* to communicate, what decisions it's *supposed* to own, you have a model you can compare against the actual behavior of the team. Most organizations don't do this. They have org charts, which describe the how — who reports to whom — but no twin that describes the what. So when the team drifts, nobody notices. The drift becomes the new normal. A year later, a consultant is hired to explain why nothing works.

Markov blankets, or: why the twin must have edges

There's a concept from statistics called a Markov blanket. The Markov blanket of a variable is the smallest set of other variables you'd need to know in order to predict it — everything else is redundant. Once you have the blanket, the rest of the world can be ignored.[3]

This turns out to be exactly what a digital twin needs. A twin that tries to model everything is useless; it's just a second, slower copy of reality. A twin that models too little misses the signals that matter. The art is drawing the blanket: identifying the minimum set of variables that makes the thing predictable, and declaring that everything outside the blanket is, for the purposes of this model, noise.

This is another way of saying that digital twins are a discipline of selective ignorance. You're declaring, on the record, what you're willing to care about. That declaration is the whole point. It's the thing that makes the *what* inspectable — because now you can point at it and argue about it, rather than squinting at code.

And once you can argue about it, you can evolve it. Which is where the twin starts doing something more interesting than testing.

The twin as a place to think

Most of what people call "design" in software is actually coding using whiteboards. The artifact that survives is the code. The design, if it existed, lives in someone's head and decays on a half-life of weeks.

A digital twin gives design a durable substrate. You can sketch the behavior before you build the implementation, and run the sketch, and watch it do things that surprise you. You can change one assumption and see what breaks. You can hand the twin to a colleague and ask, *does this feel right?* — a question you can't meaningfully ask about a 30,000-line codebase.

This matters more as systems get bigger, and as hyperspatial statistical marvels contribute to them. In a small system, you can hold the whole thing in your head, and the twin feels redundant. In a large system, no one holds the whole thing in their head, and without a twin there is no place where the whole thing exists at all. It's scattered across services and teams and Slack channels. The twin is sometimes the only artifact that represents the system as a whole, something that makes it exceedingly valuable when evaluating future possibilities via simulations using the twin.

There's a related idea in organizational practice called the *collective tacit*:[4] the knowledge that lives not in any one person's head but in the spaces between practitioners. A good digital twin makes some of that collective tacit legible, maybe *formal and explicit*. It pulls understanding out of the spaces between people and puts it somewhere you can point at. Not all of it — tacit knowledge is tacit for a reason — but enough to argue about.

What twins are not

Twins aren't prophecy. A model is always a simplification, and a simplification is always wrong in some direction. The question isn't whether the twin lies, but whether its lies are useful. A good twin lies about the things that don't matter and tells the truth about the things that do.

Twins also aren't replacements for the real thing. You still have to build the real thing, and run it, and watch it break. The twin just gives you a second opinion. Its value comes from being independent — from being made of different assumptions than the production system. If you generate the twin from the same spec the production code is written against, you've built a mirror, not a witness. The point is to have two witnesses whose stories you can compare.

This, I think, is the deepest reason digital twins matter, and the one least appreciated. They're a technique for having more than one version of the truth on hand. Most engineering cultures have exactly one version of the truth: the running code. When the code is wrong, there's nothing to check it against. A twin is a cheap second truth.

There's a beautiful hardware precedent for this that probably many software engineers have never heard of. In 1980, Bill Foster, Gardner Hendrie, and Robert Freiburghouse founded Stratus Computer to build machines that didn't fall over. Their competitors were trying to solve fault tolerance in software. Stratus bet that hardware had gotten cheap enough to solve it a different way: by running the same computation twice, in parallel, on identical boards. Each board was itself internally duplicated — a pair — and the pair was paired with another pair. The architecture came to be called *pair and spare*.[5] Two CPUs on a board ran in lockstep, executing the same instruction on the same cycle, and compared

results continuously over a dedicated bus. When they disagreed, the board took itself offline and its twin kept running, with no interruption to the program on top. As it turns out, I was the beneficiary of this when I worked for a defense contractor. One day, I came into work and a hardware component had arrived to replace a faulty part detected in the Stratus we were using to design software for a next-generation manufacturing system. None of our team was aware of the fault, but the Stratus phoned home and scheduled part delivery and replacement.

What's interesting about this for our purposes is that the two halves of a Stratus machine weren't computing *different* versions of the truth. They were computing the *same* truth twice, on independent hardware, specifically so that any disagreement was diagnostic. The twin wasn't a second opinion about what to do. It was a second opinion about whether the first opinion was trustworthy. The whole point was the comparison — the bus between the halves was the place where meaning lived, because meaning, in a fault-tolerant system, is whatever both halves agree on.

That's a different use of twinning than a TLA+ model or a switch simulator, but the underlying move is the same: you build two things that ought to match, and you make the match itself an artifact you can inspect. Stratus did it in silicon to catch hardware faults. We can do it in software to catch specification faults. In both cases, the twin earns its keep not by being right, but by being *another witness* whose disagreement with the primary is information.

The real lesson

If you zoom out, the question digital twins are really answering is an old one: where does meaning live? In a system that works, meaning has to live somewhere. If it only lives in the code, the code becomes both the law and the enforcement, and you have no appeal. If it lives in a separate, inspectable, runnable model — a twin — then you have two places where meaning lives, and you can compare them, and the difference between them is the thing you actually want to look at.

This is why the most interesting applications of digital twins are not in factories. They're in software, in organizations, in everywhere we've let the how swallow the what. The fix isn't more documentation. Documentation is a twin that doesn't run. The fix is a twin that does run — a model you can poke, that pokes back when you get it wrong.

The reason we don't have more of them is that building one forces you to say, out loud, what you actually want. And it turns out most of us don't know. That's uncomfortable, but it's also the whole point. A twin that only reflects what you already knew is decoration. A twin that reveals what you didn't know you didn't know is the thing worth building.

Further reading and references

These sources informed the essay but are not cited inline.

Dahmen, U. et al. (2020). “How to tell the difference between a model and a digital twin.” *Advanced Modeling and Simulation in Engineering Sciences*. link

Lehner, D. et al. (2023). “Digital-twin-based testing for cyber–physical systems: A systematic literature review.” *Information and Software Technology*. link

Winans, T. (2024). “Digital Twins and Software Testing.” *Medium*. link

1. Grieves, M. and Vickers, J. (2017). “Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems.” The term *digital twin* was coined by John Vickers of NASA around 2010; the formal concept was introduced by Michael Grieves at the University of Michigan in 2002, originally as the *mirrored spaces model*. Background: “Grieves and Vickers — the history of digital twins” and *Digital twin* (Wikipedia).
2. TLA+ (Temporal Logic of Actions) is a formal specification language created by Leslie Lamport for describing and reasoning about concurrent and distributed systems. You write a mathematical description of the system’s states, transitions, and invariants; the TLC model checker explores every reachable state to verify that your invariants hold. See Lamport’s TLA+ home page and Newcombe, C. et al. (2015), “How Amazon Web Services Uses Formal Methods,” *Communications of the ACM*.
3. Winans, T. (2024). “Markov Blankets in Digital Twins: Enhancing Interaction Contexts and Decision-Making.” Develops the link between Markov blankets, Markov decision processes, and digital twin orchestration.
4. Winans, T., Brown, J.S., and Pendleton-Jullian, A. (2025). “Organizational Jazz: New Ways to Work.” Source of the *collective tacit* — knowledge that lives between practitioners rather than within them.
5. Stratus Computer, Inc. was founded in May 1980 by William E. Foster, Gardner C. Hendrie, and Robert A. Freiburghouse; the Stratus/32 shipped in February 1982. Its architecture used duplicated self-checking CPUs running in lockstep on a single board, with a second identical board held as a hot spare — the *pair and spare* design. Disagreement between the two halves of a pair, detected over a

dedicated comparison bus, removed the suspect board from service while its twin continued executing. See “Stratus Computer, Inc.” (Encyclopedia.com), “History of Stratus Computer, Inc.” (FundingUniverse), “Stratus: Servers that won’t quit” (The CPU Shack Museum), and *Stratus Technologies* (Wikipedia).