



Optimization is also too confident.

Autonomy Platform · Public Blog

P E R S P E C T I V E

© 2026 Azirella Ltd. All rights reserved worldwide.

[Read more at azirella.com/blog](https://azirella.com/blog)

TL;DR

Tim Varelmann argues for AI on the outside and mathematics at the core, because a mathematical model cannot hallucinate. I agree with most of it. But optimization shares System 1's worst habit: it treats what it was given as the whole world, and it returns one answer with no likelihood of realization and no range of outcomes. The likelihood of realization is the measure of fragility, and an optimum, having no slack by construction, is often fragile: it scores best on expected value and is often the plan least likely to deliver it. Making the solver probabilistic by running it across scenarios makes it slower, and in a supply chain a late answer is an answer to a smaller question. There is a faster route to the distribution: build it from the record, and on day one from a digital twin calibrated on the company's own history and checked forward against every outcome. What wins is a fast decision that carries a calibrated range and likelihood of realization, and that adjusts when the world moves.

KEY POINTS

- Optimization shares two of System 1's traits: it treats what it was given as the whole world, and it states one answer with no likelihood and no range.
- An optimum has no slack by construction, so it is often fragile: the plan with the best expected value is often the least likely to deliver it, and when it misses, the cost grows faster than the error (Taleb).

- A neural network is a solver too, and gives neither number on its own. Both come from checking answers against outcomes, and on day one from a digital twin calibrated on the company's own history.
- Options expire. A better answer that arrives late is an answer to a smaller question, and the real delay is the batch cycle and the approval queue, not the compute.
- Every planning system freezes its parameters at the numbers someone typed in. Exogenous signals should move them. Speed is safe only with calibration: latency compounds cost, velocity compounds value.

Tim Varelmann wrote [a thoughtful reply](#) to [my post on Kahneman's System 1 and System 2](#). His argument: every large language model is System 1, fast and never short of an answer, so put mathematics at the core and let AI wrap it. The model makes the decision; AI reads the context, calls it with the right parameters and explains the result.

I agree with more of it than I disagree with, so start there.

He is right that “wrong and inventive are different failure modes, and only one of them is debuggable.” A mathematical model that is wrong is wrong in a place you can find. He is right that reproducibility matters: store the inputs, and you can rerun exactly what ran six months ago. And he is right that the logic in a structural model extrapolates. If a truck needs a driver, fifteen trucks need fifteen drivers, whatever the history contained.

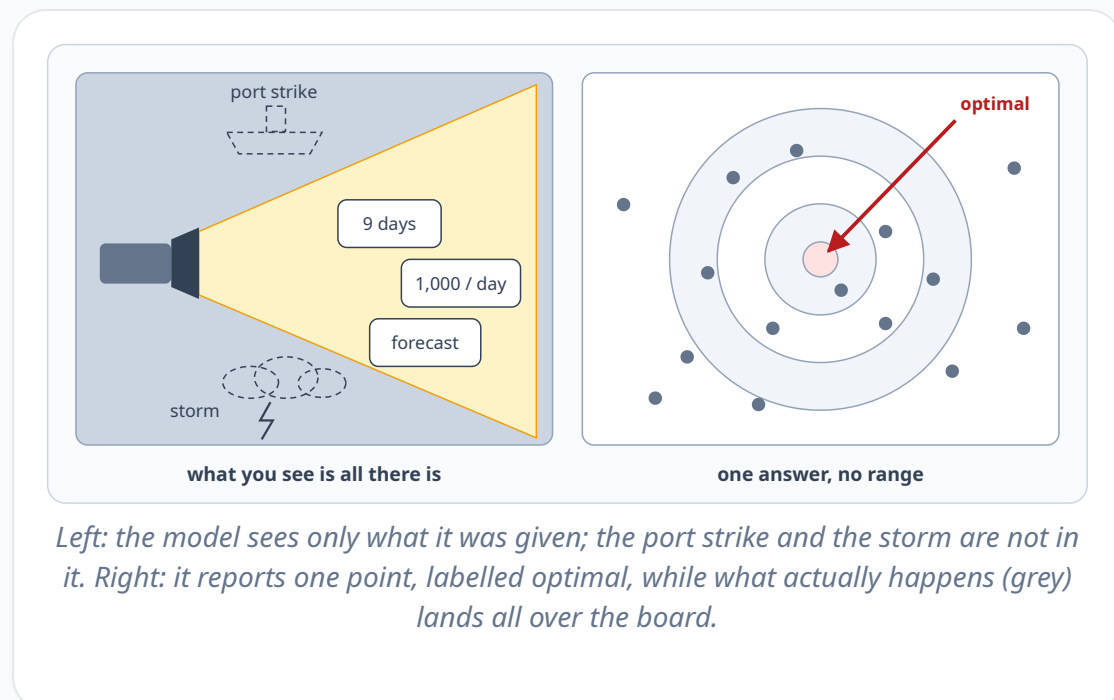
Where we differ is on what sits at the core. I think optimization has System 1's worst habit too.

Two System 1 traits, in a solver

Kahneman's System 1 has two traits that matter here.

The first he called “what you see is all there is.” System 1 builds the most coherent story it can from what is in front of it and never asks what is missing. An optimization model is the purest form of that. A constraint you did not write does not exist. A lead time you entered as nine days is exactly nine days, every time, in every scenario.

The second is overconfidence. System 1 always has an answer and never says how sure it is. A solver returns one plan, labelled optimal, with no probability attached. It says it with the same confidence whether its inputs were measured last week or guessed three years ago.



Neither trait is a flaw in the mathematics. Every model is an approximation; George Box's line that all models are wrong, but some are useful, is nearly fifty years old. The problem is what we do with the answer: we act on it as though the model were exact. Nothing in a plan says how approximate the model behind it is, so the approximation is invisible at exactly the moment it matters, and a plan from a model that is roughly right is executed as if it were precisely right. That is true of every model, including the digital twin we use ourselves, which is why the rest of this piece is about measuring the gap rather than assuming it away.

Put concretely, a plan leaves two questions unanswered, and I want to be precise about the words:

- **Likelihood of realization: how likely is it that the plan's expected result actually comes true?** The plan says a fill rate of 97%. Is that a 90% outcome or a 40% outcome? This is my measure of **fragility**. A fragile plan is one that looks excellent and rarely delivers it.
- **Probability: what is the range of outcomes?** The full spread of what could happen: if it misses, by how much; if it goes well, how well. This is the distribution the likelihood of realization is read from.

When I have objected to optimization in the past, I talked about likelihood of realization as if it were a question of confidence. It is really a question of fragility, and that is the half that hurts.

Tim writes that a model “will never return a schedule that violates its own constraints.” True. The supply chain will. A feasible plan built on a capacity figure that is 15% too high is still an invented answer. It is just an invented answer in a tidier format, and it is harder to catch precisely because it is internally consistent.

The overconfidence is built in

This is not a matter of sloppy inputs. It has a name: the optimizer’s curse (Smith and Winkler, 2006).

Picture twenty suppliers who all genuinely deliver in ten days. Your estimates of their lead times are honest, off by a day or two either way at random. The optimizer picks the one that looks fastest: eight days. Its true lead time is still ten. Choosing the best-looking estimate systematically chooses the most flattering error, so the plan you pick will on average disappoint, even though no single input was biased.

The more options an optimizer searches, the stronger the effect. That is the uncomfortable part: the better the search, the more confidently it finds the flattering error.

Optimal is often fragile

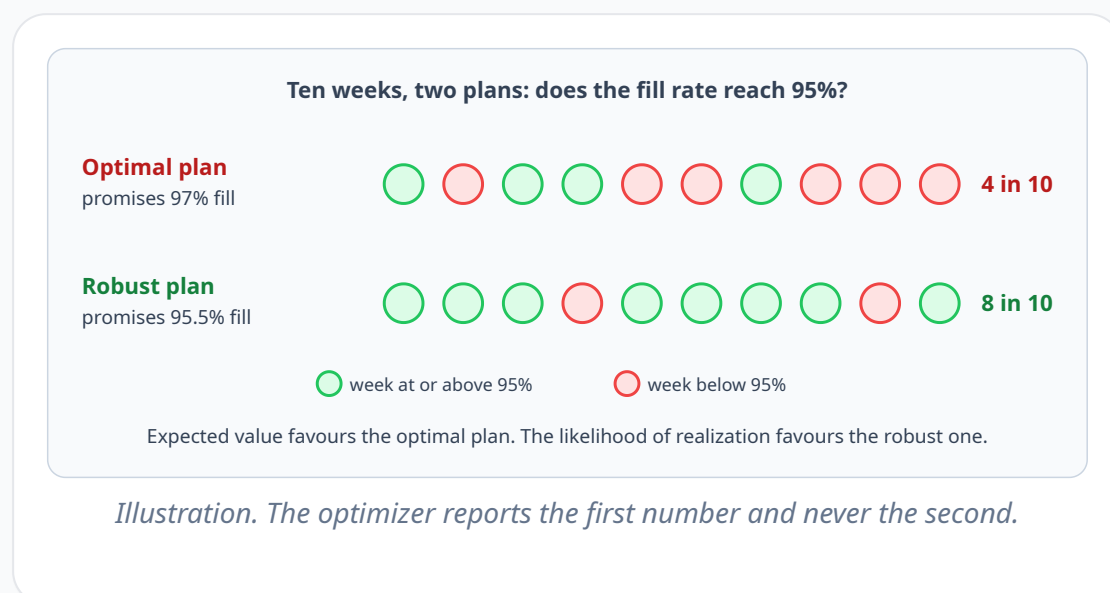
An optimal plan is, by construction, a plan with no slack. The optimum of a linear program sits on a corner where the constraints bind: it uses exactly the capacity available, holds exactly the inventory the model says is enough, and ships on exactly the lanes that are cheapest. That is what makes it optimal. It is also what often makes it fragile. Take ten percent off one binding constraint and the plan does not degrade gracefully; it becomes infeasible, and something downstream has to absorb the difference.

Small input changes can also flip the plan. Planners have a name for it: nervousness. A slightly different forecast, and the solver jumps to a different corner with a very different set of orders, though the two plans may be almost equally good on paper.

In [the comments under Tim's post](#), Tuyen Trung Truong asked the sharper question. Most optimization beyond linear programming is iterative and starts from somewhere; start somewhere else and you can get a different plan. Tim's answer is right as far as it goes: store the starting point and you can reproduce the run exactly. But reproducing a choice does not make it any less arbitrary. If a different start gives a different "optimal" plan, the spread between them is a measure of how uncertain the answer is, and the solver throws it away.

And the starting point is rarely neutral. In industry it is usually a heuristic or yesterday's plan, and for the hard problems (whole-number decisions, a time limit, the best plan found so far) the answer tends to stay near where it started. Kahneman called that anchoring. It is a third System 1 trait, sitting inside the most deliberate tool we have.

This is why I measure fragility as the likelihood of realization. Picture two plans for the same network, as an illustration. The optimal plan promises a 97% fill rate and, with every unit of slack spent, reaches 95% in only four weeks out of ten. A robust plan promises 95.5% and reaches 95% in more than eight weeks out of ten. On expected value the optimal plan wins. On likelihood of realization it loses badly, and the solver reports only the first number. The optimizer's curse and the missing slack are the same failure seen twice: the plan that scores best is the plan least likely to deliver its score.



How badly it breaks: Taleb

The likelihood of realization says how often a plan breaks. It does not say how badly, and in my experience that is where the money goes. When an optimal plan goes astray, the cost is not a little worse than planned. It is enormous.

Nassim Taleb gave this a precise meaning in *Antifragile* (2012). Something is **fragile** when the damage grows faster than the error: a small mistake costs a little, a mistake twice as large costs far more than twice as much. A glass survives a hundred small knocks and not one large one.

A supply chain running an optimal plan is often fragile in exactly this sense, and the mechanism is the missing slack. While the error is smaller than the buffer, it costs almost nothing; the buffer absorbs it. Once the error is larger than the buffer, every extra unit of shortfall is paid for at the worst price available: expedited freight, overtime, a spot buy, a lost order, a retailer's chargeback. And a shortfall does not stay where it started. One missing component stops every assembly that needs it; a late inbound empties the shelves of every customer that lane serves; the replan that follows disturbs orders that were fine. An optimal plan has spent its buffers, so it pays that price from the first unit of error.

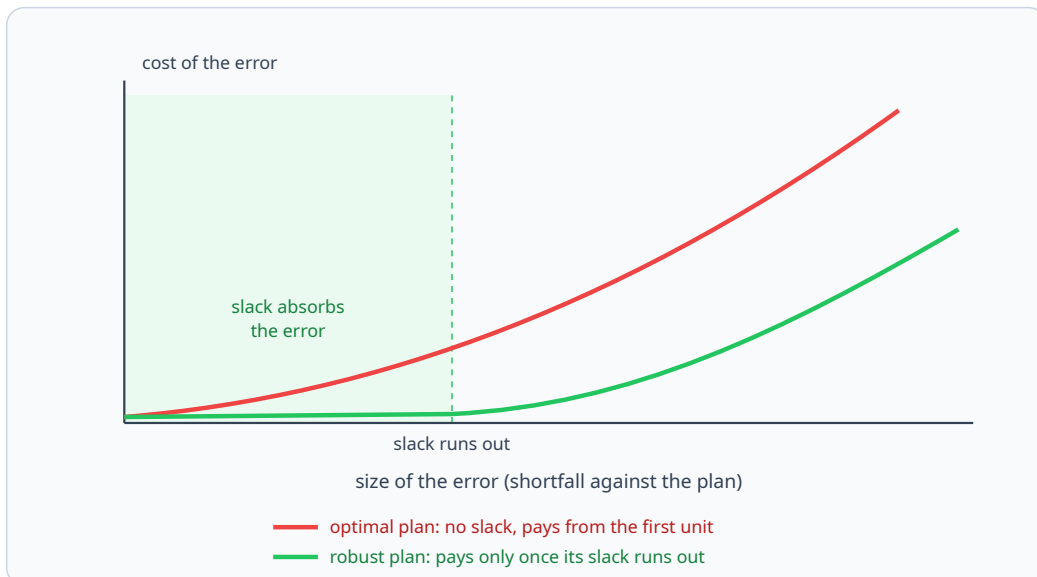


Illustration. Past the slack, every extra unit is paid for at the worst price available (expedites, spot buys, lost orders), so the cost grows faster than the error: Taleb's fragility.

Taleb's argument has a mathematical consequence that matters for every deterministic plan. When costs rise faster than errors, **a plan built on average inputs does worse, on average, than it says it will.** That is Jensen's inequality, and Sam Savage called its everyday form the flaw of averages (2009): plans based on average conditions are wrong on average. A deterministic optimization plans at the average lead time, the average demand and the average yield. Its reported cost is not the expected cost. It is a floor the real cost will almost never reach.

So fragility needs two numbers:

- **the likelihood of realization:** how often the plan breaks;
- **the tail of the range:** how badly it breaks when it does.

And here I have to be careful, because the argument cuts both ways. A mathematical optimizer gives neither number. But the neural networks we use are solvers too: they take a state and return a decision, one answer, with no likelihood and no range attached. Worse, neural networks are known to be overconfident out of the box (Guo et al., 2017): their internal “confidence” reads high whether or not the answer turns out right. We found exactly that in our own agents.

So the difference is not the solver. An optimizer and a neural network are both System 1 in this respect. The two numbers have to come from something outside the solver that compares its answers with what actually happened, and that layer can sit around either one. What a learned policy does change is the price of robustness: trained across stressed situations and penalised for shortfalls, it can learn to keep slack where the damage would be steep. An optimizer can be built to do that too, with a robust formulation. Neither does it by default.

A plan can be unlikely to hit its target and still miss by a little, or likely to hit it and occasionally miss by a mile. The second is the one that ends careers. An honest plan states both, which is why we aim for robust over optimal. Robust, in Taleb’s vocabulary, means a plan that is not harmed much by the error it did not foresee, and the price of that is slack an optimizer would call waste.

“Then use stochastic optimization”

That is the natural reply, and it is a fair one. Operations research has had answers to uncertainty for a long time. Chance-constrained programming (Charnes and Cooper, 1959) states the probability a constraint holds. Stochastic programming produces a distribution of outcomes. Robust optimization produces bounds.

Two problems remain.

The first is **calibration**. A stochastic program’s range is only as honest as the distribution you fed it, and that distribution is another input someone chose. Nothing in the method compares the stated probability with what actually happened. A range that is never checked against outcomes is still a confident claim, just a wider one.

The second is **time**. A full network plan in weekly buckets over two years is millions of variables. As a linear program that is minutes. Add lot sizes, set-ups and minimum runs and it is often hours, usually stopped at a time limit with the best plan found so far rather than a proven optimum. Now make it probabilistic by solving across many scenarios, and you multiply it. That is why network plans run overnight in a batch.

Getting the distribution without the scenarios

So a probabilistic answer from a solver is slow and unchecked. That is not an argument against probabilistic answers. It is an argument for getting the distribution a different way.

The principle is to build the range from the record rather than assume it. Conformal prediction (Vovk, Gammerman and Shafer, 2005) is the cleanest form of that idea. It starts from the question a stochastic program never asks: how wrong have our predictions actually been? Compare each past prediction with what happened, keep the misses, and size the band on the next prediction from them. If eight in ten of the past misses were smaller than a given size, a band that wide will hold about eight times in ten, as long as the future resembles the recent past. No assumption that errors are normal, and no assumption that the model is right: a worse model simply earns a wider band. And at decision time it costs a lookup, not a thousand scenario solves.

We use that today on demand, where the misses of past forecasts size the band on the next one. For lead times, transit times and yields we build the range from what those quantities have actually been. Either way the agents buffer against the uncertainty that is really there, not a constant someone chose.

That still leaves the hard part: a likelihood of realization for each **decision**, not just a range on each input.

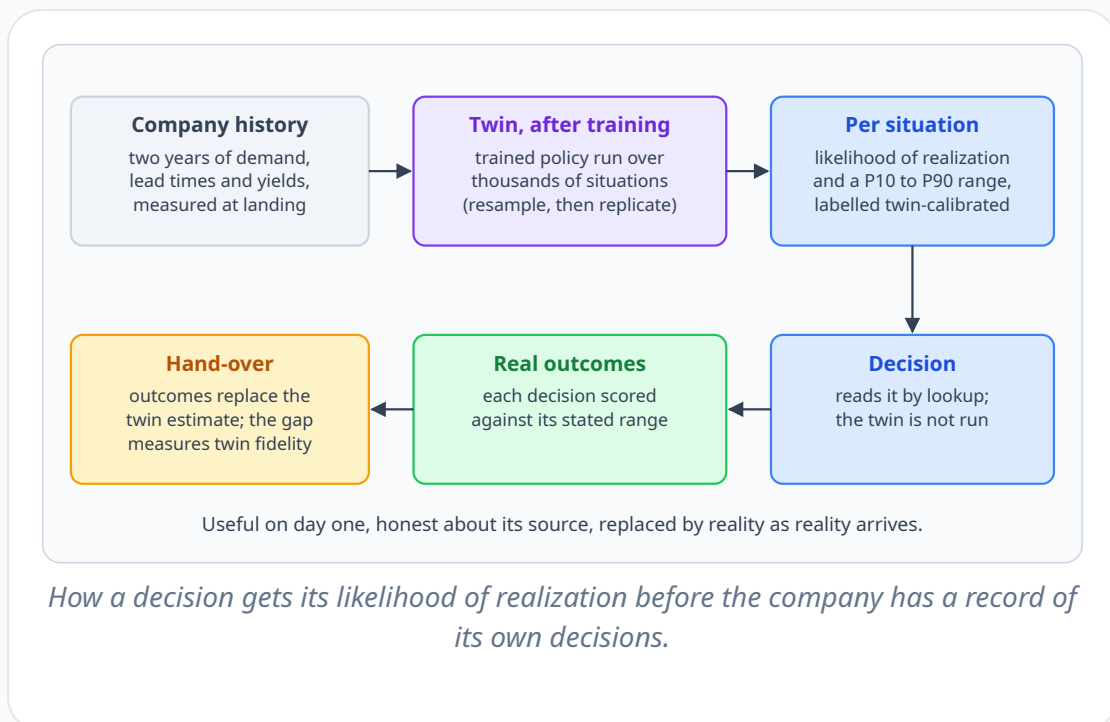
Day one, without years of history

The objection to anything built from the record is that you need a record. I do not want a system that is only useful after two years of its own decisions, and it does not have to be.

On the first day there is no record of our decisions, but there is the company's history: two years of demand, lead times and yields, which every tenant brings with it. We measure how those vary. Then, after the agents are trained, we use the digital twin a second time: we run the trained policy through thousands of situations drawn from that

measured variation and record, for each kind of situation, how often its decisions delivered what they promised and how far they missed when they did not. That becomes the likelihood of realization and the range the decision carries. The twin does this ahead of time; when a decision is made, it is a lookup.

A likelihood built this way is only as good as the twin, and I would rather say so than have it said for me. It is calibrated against a model of the supply chain, not against the supply chain. So it is labelled as coming from the twin, and it hands over to the real thing: as outcomes arrive, they replace the twin's estimate, and the gap between what the twin promised and what happened is the measure of how much to trust the twin.



Today our likelihood is coarser than this. It is set for each agent, not for each decision, which means it cannot yet tell a fragile plan from a robust one. Making it per decision, from the twin on day one and from outcomes after, is the work in front of us.

So the twin's fidelity is the whole game

That makes the twin the thing to get right, and it raises the question of how to test it.

The obvious test is to replay the company's history through the twin and see whether it reproduces what happened. I have watched that go wrong at company after company. They buy a new planning system to plan better, and then judge it on whether it can

reproduce the plans of the system it replaced. History is full of expedites, manual overrides and data errors no model carries, so every mismatch is blamed on the new system, and the old system becomes the standard the new one is held to.

So we test the parts against data and the whole going forward, and never grade the twin on reproducing the past:

- **The inputs, directly.** Demand, lead times, yields and uptime vary for reasons that have nothing to do with anyone's decisions, so each can be checked against held-out data: does the twin's range for lead time hold about eight in ten real receipts?
- **The physics, by invariant.** Stock balances, capacity is never exceeded, bills of materials are consumed correctly, lead times add up along the chain.
- **Known behaviour.** Bullwhip grows upstream; service rises with safety stock along the expected curve; queues explode as utilisation approaches 100%. Ask planners whether they can tell the twin's trajectories from real ones.
- **Its own blind spots.** The twin is a discrete-event simulation, so every replication draws new values for demand, lead times and yields. That covers the randomness. It does not cover doubt about the model itself, because every replication shares the same assumptions: a lead-time spread estimated from forty receipts might really be narrower or wider, and supplier delays might move together rather than independently. So we replicate in two layers: resample the history and refit the inputs (and flip a doubtful assumption) on the outside, replicate as usual on the inside, and widen the band by how much the outside layers disagree. You cannot prove a twin right, but you can stop it being confidently wrong.
- **Forward, from week one.** Every decision states its range and likelihood; count how often reality lands inside. That is the only true test, and it takes weeks, not years.

None of the parts tests how they interact: shared capacity, a shortage spreading through a bill of materials. Only the forward test does that, and the two-layer replication is what protects you until it has run long enough.

Latency compounds cost

In a supply chain, options expire.

Say a supplier's port closes on a Tuesday. On the Tuesday you can still re-source, re-route, pull forward, or tell the customers who matter. By the next weekly plan some of those options have gone: the alternative supplier's capacity is committed to someone

else, the vessel has sailed, the customer has already heard about it from their own supplier. The perfect optimization that runs on Sunday night is the perfect answer to a much smaller question.

So the bottleneck is not really solve time. It is the whole loop: seeing the change, deciding and acting. In most companies that loop is set by the batch planning cycle and by the approval queue in front of every plan. Compute is the smallest part of it.

This is also Taleb's point seen from the other side. For him, what protects against the unforeseen is optionality: the ability to act once you know more. Options are exactly what expire while a plan waits for its batch. A business that responds within hours keeps its options open; one that responds on Sunday has let them lapse.

That is the idea behind the line we use: **latency compounds cost; velocity compounds value.**

Moving the computation off the decision path

There is a precise name for how a learned policy changes this: amortized optimization (Amos, 2022). A solver pays the full cost of the computation at decision time, every time. A learned policy pays it once, in training, and then answers in milliseconds.

The cost does not disappear; it moves. That is the whole point. It moves off the path between the port closing and the decision being made.

Tim's second point is that learned models only interpolate, which is true. His example is a scheduler that learns ten trucks need ten drivers and then meets a shift with fifteen. Here is how that plays out for us. Our agents do not learn from history alone. They learn on a digital twin: a structural model of the network carrying its bills of materials, capacities, lead times and flows. The twin is where the fifteen drivers live. It is Tim's deterministic logic, used to generate far more situations than history ever contained, including ones that have not happened yet.

AlphaGo Zero is the best-known version of this. It was given the rules of Go and learned only by playing itself, and it beat the version that had learned from human expert games 100 to 0 (Silver et al., 2017). The lesson is not that it learned without rules. It had the rules. The lesson is that learning inside a faithful model of the world beat imitating the humans who had played it before.

The honest limit is that a learned policy is still strongest inside the range it was trained on. Even in Go, where the rules never change, researchers found a simple strategy that beats KataGo, a superhuman program, because self-play had never visited those positions (Wang et al., 2023). A supply chain is harder than Go: the rules change. So we aim for robust over optimal, and every decision carries a calibrated range that widens when the situation looks unlike what the policy has seen. When it strays, the band says so.

Every planning system freezes the world

The second flaw is not specific to optimization. It is shared by every planning and prediction system, including learned ones: the parameters stay at the numbers someone typed in.

A lead time is not a constant. It moves with port congestion, weather, a supplier's other customers, a strike three countries away. Demand planning has adjusted for outside drivers for years: promotions, weather, the economy. Supply planning mostly has not. Lead times, transit times, yields and capacities are set once and revisited when someone notices.

What we are building is a planning system that adjusts those parameters from exogenous information automatically, and a policy that takes the signals as inputs, so it can respond to a changed world without being retrained. The signal moves the decision; the weights do not have to.

This is also where the hard problems are, and I would rather name them than have them named for me:

- **History is confounded.** When a hurricane came, planners expedited. The history shows the hurricane plus the response, so a naive fit learns that hurricanes do not hurt.
- **Rare events are rare.** One company's history holds few port strikes. Public data about the event can be shared; how a given supply chain responds to it cannot, and should not be borrowed from someone else's.
- **An outside feed is itself a coherent story.** It is exactly the kind System 1 falls for. An adjustment that is not checked is a new source of overconfidence.

So the history does not go straight into the policy. It goes into the twin: it teaches the twin how a storm stretches transit times, and the policy is then retrained inside the corrected twin. That layer is the direction we are building, not something we have

finished.

Fast and wrong also compounds

Speed on its own is not the argument. Reacting fast to noise produces nervous plans and a bullwhip effect. What makes speed safe is knowing how sure you are.

That is where the two flaws meet. The calibrated likelihood of realization decides when to act fast and when to wait: act now on a confident call, hold on a doubtful one, and say which is which. Every decision states what it expects to happen; the outcome is measured against it; the model earns its confidence or loses it. And every automatic adjustment of a parameter is held to the same standard, so a bad adjustment shows up as a drop in calibration rather than a quiet drift.

This also gives “velocity compounds value” its mechanism. Each decide-measure-learn cycle improves the policy. A business that decides weekly learns 52 times a year; one that decides daily learns 365 times.

In fairness to the argument I am making: our own middle planning tier replans daily today. The event-driven path that would let it react within the hour is designed and not built. Velocity is the direction, and we are not there yet.

Keep the solver, as a benchmark

None of this makes optimization useless. It makes it the right tool for a different job.

A fast policy is only worth having if it is close enough to the slow optimum, and there is a clean way to find out. Run the full optimization offline, on test data, with all the time it needs, and measure how far the policy falls short of it. That number is the honest price of speed. If it is small, speed wins. If it is large, the policy is not ready.

Who checks the core?

AI on the outside, mathematics at the core: I agree with the shape. Where I would push Tim is on what makes a core trustworthy. Reproducibility tells you the same inputs give the same answer. It does not tell you the answer was right, and it does not tell you how close the answer was to a very different one.

A solver with no track record is confident, whether it is a linear program or a neural network. A solver with a track record is calibrated. What belongs at the core is whatever keeps being checked against what actually happened, and gets there while the options are still open.

Latency compounds cost. Velocity compounds value.

See Autonomy in action

Walk through how Autonomy models, executes, monitors, and governs supply chain decisions with autonomous AI agents.

[See It Live](#)