BIT BROS LLC

WHITE PAPER

A Practical Architecture for Simulation-Based Supply Chain Policy Optimization

Adam DeJans Jr.

June 29, 2026

Originally published on LinkedIn

White-paper edition: original prose, reformatted for reading and print.

The Core Challenge

Many supply chain problems do not fail because we lack a model. They fail because the model is embedded inside a larger operational system that is difficult to search through directly. The replenishment logic may be known. The allocation rules may be known. The simulator may already exist. The business objectives may be mostly understood.

The hard part is that the decision policy has too many parameters, those parameters interact with each other, and every serious evaluation requires running an expensive simulation.

This is a common situation in real supply chain work. A team may have a simulator that can replay demand, lead times, capacity constraints, purchasing behavior, inventory movement, service failures, substitution, vendor performance, transportation costs, and downstream operational impacts. The simulator can answer the question, "What would happen if we ran the business with this policy?" But that does not automatically answer the more important question, which is, "What policy should we run?"

Why Parameter Tuning Breaks Down

At first, the problem appears to be simple parameter tuning. We may have safety stock multipliers, reorder thresholds, buying cadence rules, expedite triggers, service-level targets, inventory caps, allocation penalties, smoothing weights, lead time buffers, and exception logic. Each of these parameters exists for a reason. They are not arbitrary mathematical decorations. They encode business preferences, operational constraints, risk tolerance, and planner judgment.

A safety stock multiplier expresses how much uncertainty we are willing to protect against. An expedite trigger expresses when we are willing to pay for speed. An allocation penalty expresses how painful it is to short one node, channel, product family, or customer segment relative to another.

The obvious solution is to try a lot of parameter combinations and pick the best one. That instinct is natural, but it breaks down almost immediately. If we have ten parameters and each parameter has only five possible values, we already have nearly ten million combinations. If each simulation takes several minutes, the brute-force approach is no longer an analytical method. It is an uncontrolled compute problem.

In a realistic system, the situation is even worse because many parameters are continuous, some are categorical, some are integer-valued, and some only matter when another parameter is active.

Interactions and Policy Complexity

The deeper issue is that the parameters are not independent. A reorder threshold may only make sense under a certain buying cadence. An expedite trigger may only matter when lead time uncertainty is high. A smoothing weight may help when forecasts are noisy but hurt when demand shifts quickly. A service-level target may be economically reasonable for high-volume products and completely irrational for slow movers.

A capacity penalty may appear irrelevant until the policy creates inbound spikes that overload a warehouse. In other words, we are not tuning isolated knobs. We are searching through complete operating policies.

This is where one-factor-at-a-time experimentation becomes weak. If we change one parameter while holding all others fixed, we are implicitly assuming that the system is mostly separable. In many supply chain systems, that assumption is false. The interactions are often the most important part of the problem. A policy can look good only because another parameter is compensating for it.

A parameter can appear unimportant only because the tested region of the policy space never exposed the condition where it matters. Real operational behavior emerges from combinations of choices, not from individual settings viewed in isolation.

Separating Evaluation from Search

A better way to think about the problem is to separate evaluation from search. The simulator is not the optimizer. The simulator is the evaluator. It tells us how a proposed policy behaves under a specific set of assumptions, demand paths, lead times, capacities, and operational rules. The optimizer is the learning loop wrapped around the simulator. Its job is to decide which policy should be evaluated next, based on everything that has already been learned.

This distinction is important because it prevents us from treating simulation as a passive reporting tool. A simulator can tell us what happened under a policy, but by itself it does not know how to spend the next experiment wisely. If every simulation run is expensive, then every run should have a purpose. Some runs should explore unknown regions. Some should refine promising regions. Some should stress-test the current best policy. Some should eliminate obviously bad areas of the search space.

Some should investigate parameter interactions. Some should validate whether a strong result was real or merely lucky.

Defining the Policy Space

The first step in building this kind of architecture is to define the policy space carefully. This means identifying the parameters the system is allowed to change, the valid ranges for those parameters, and the operational rules that govern feasible combinations. A continuous parameter might represent a safety stock multiplier. An integer parameter might represent a review period. A categorical parameter might represent a buying cadence such as daily, weekly, or biweekly.

A conditional parameter might only be relevant when expediting is enabled or when a product belongs to a certain class.

This step deserves more attention than it usually receives. The search space is part of the model. If we define it poorly, the learning loop will search through nonsense efficiently. For example, we should not allow a vendor-specific buying cadence that violates the vendor's actual shipping calendar. We should not allow a slow-moving product to chase a service target that would require absurd inventory. We should not include an expedite threshold if expediting is disabled for that product family.

We should not treat a parameter as globally adjustable if the business would only ever change it by region, node, item class, or vendor group.

Designing the Scoring Function

Once the policy space is defined, the next step is to define the scoring function. This is where supply chain optimization becomes much more serious than generic prediction work. A policy is not good because it minimizes forecast error. A policy is good because it produces better decisions under uncertainty. The simulator therefore needs to return an economic evaluation of the policy, not just a collection of technical metrics.

The score might include holding cost, lost sales, markdowns, expediting cost, transportation cost, labor impact, capacity violations, late orders, vendor constraints, customer service penalties, cancellation risk, and operational instability. The exact terms depend on the business, but the principle is consistent. We need a score that reflects the tradeoffs the business actually cares about. A policy that lowers inventory but increases lost sales may or may not be good.

A policy that protects service but floods the network with excess inventory may or may not be good. A policy that reduces transportation cost but creates late orders may or may not be good. The answer depends on the economics.

This does not mean that all complexity should be hidden inside a single number. The learning loop may need a scalar objective to compare policies, but the business should still see the underlying KPI breakdown. We should know whether a policy won because it reduced stockouts, lowered inventory, reduced expediting, smoothed inbound volume, or simply exploited an unrealistic assumption. The scalar score guides the search. The KPI decomposition explains the decision.

Starting with Exploration

After defining the policy space and the scoring function, the next question is where to begin. The wrong instinct is to immediately hunt for the best policy. At the start of the process, the system knows very little. It does not know which parameters matter. It does not know where the unstable regions are. It does not know whether the objective surface is smooth, jagged, flat, or full of cliffs. It does not know whether good policies are common or rare.

The first experiments should therefore be designed to learn the shape of the problem, not merely to find a winner.

A good initial experiment set should be diverse. It should include conservative policies, aggressive policies, low-inventory policies, high-service policies, policies that buy frequently, policies that buy less often, policies that expedite early, policies that rarely expedite, policies that smooth aggressively, and policies that react quickly. The goal is to cover meaningful regions of the policy space so that the system can begin to understand how the decision levers affect the outcome.

This is especially important in supply chain because bad policies often do not fail in a smooth or obvious way. Many systems behave reasonably until a threshold is crossed, and then the behavior changes dramatically. A warehouse may operate normally until inbound volume exceeds capacity. A replenishment policy may look fine until lead times stretch. An allocation rule may look fair until constrained supply forces painful tradeoffs.

A buying policy may reduce inventory until it suddenly creates service failures that cascade into expediting, substitutions, and future demand distortion. The first experiments should expose these cliffs.

Building an Approximate Model

Once we have evaluated an initial set of policies, we have more than isolated simulation results. We have data. Each policy evaluation tells us which parameter values were used, what the simulator observed, how the policy performed against the objective, and how the underlying KPIs behaved. At this point, it becomes wasteful to continue sampling blindly. The next step is to build a cheap approximation of the policy landscape.

This approximation does not replace the simulator. That point is essential. The simulator remains the source of truth. The approximation is only a guide for deciding where to spend the next expensive simulation run. It can be wrong and still be useful. Its purpose is not to perfectly predict the business system.

Its purpose is to provide directional intelligence about which regions of the policy space appear promising, which appear risky, which are underexplored, and which have already been shown to perform poorly.

This creates a practical learning loop. We evaluate a set of policies with the simulator. We use those results to estimate how policy parameters relate to outcomes. We then use that estimate to propose the next set of policies worth evaluating. After those policies are simulated, the approximation is updated, and the loop continues. The architecture becomes sequential rather than static. Each simulation run makes the next simulation run more informed.

Balancing Exploration and Exploitation

The decision rule for choosing the next experiment has to balance two competing needs. On one hand, we want to exploit what already looks good. If a region of the policy space contains strong policies, we should test nearby variations and see whether we can improve the result. On the other hand, we need to explore regions we do not yet understand. If we only search near the best policy found so far, we may become trapped in a mediocre region and never discover a better one elsewhere.

This exploration-versus-exploitation tension is not just a technical detail. It is how good analysts think. A good analyst does not only ask, "What is the best answer so far?" A good analyst asks, "What experiment would teach us the most right now?" Sometimes the most useful experiment is a local refinement near the current best policy. Sometimes it is a test in an uncertain region where the system has little evidence. Sometimes it is a direct challenge to a policy that looks good but may be fragile.

Sometimes it is a deliberate attempt to rule out a region that appears unlikely to be useful.

This is where simulation-based optimization becomes more disciplined than random experimentation. Random search may eventually find good policies, but it does not fully learn from its own history. It can keep wasting budget in regions that have already shown poor performance. It can fail to intensify around promising regions. It can treat all experiments as equally valuable even when some are clearly more informative than others.

A sequential learning loop, by contrast, uses the evidence accumulated so far to decide where the next unit of compute is most likely to create value.

Intensification and Robust Evaluation

Another important part of the architecture is intensification. Not every policy deserves the same amount of evaluation. If a policy performs terribly after a small number of scenarios, there is usually little reason to spend much more budget on it. If it stocks out constantly, overloads capacity, or relies on excessive expediting, we can often eliminate it early. But if a policy looks promising, it deserves more scrutiny.

We should run it on more demand paths, more lead time samples, more historical periods, more product segments, and more stress conditions.

This is especially important when simulation outcomes are noisy. A policy can look good because it happened to receive favorable demand paths. It can look stable because the tested scenarios did not include a supplier disruption, port delay, promotion spike, or capacity bottleneck. It can win on average while still performing poorly in the tail. A promising policy should therefore be forced to prove itself. The more promising it looks, the more evidence we should demand.

A fair comparison also requires careful experimental design. When two policies are compared under different random demand paths, different lead time samples, or different historical windows, the comparison can be polluted by noise. Whenever possible, policies should be evaluated using common scenarios so that the difference in performance is more attributable to the policy itself rather than to random variation in the simulation environment. This is a simple idea, but it matters.

Without it, the learning loop can chase noise.

Maintaining Multi-Objective Visibility

The architecture should also preserve multi-objective visibility even when a scalar score guides the search. Supply chain leaders rarely care about one number in isolation. They care about tradeoffs. A policy that is cheapest may be too fragile. A policy that is robust may carry too much inventory. A policy that protects service may create operational volatility. A policy that looks excellent at the network level may create unacceptable pain in a specific region, node, vendor, or product class.

For that reason, the output should include the tradeoff frontier, not just the winning policy. It should show how the best policies compare on cost, service, inventory, expediting, capacity usage, volatility, and risk. It should make clear which policy is cheap but fragile, which is stable but expensive, which performs well in normal conditions, and which survives stress conditions. In practice, the best recommendation is not always the policy with the absolute lowest simulated cost.

It may be the policy that is nearly as good economically but much easier to operate, explain, and trust.

Communicating Results Effectively

This also changes how we communicate results. Telling an operator or executive that "the model says this parameter setting is best" is not very convincing.

A stronger explanation is that we evaluated a broad set of policies, eliminated regions that created operational instability, found the cost-service tradeoff frontier, stress-tested the top candidates, compared them against the baseline using common scenarios, and selected the policy that produced the best economic improvement without creating unacceptable side effects. The recommendation becomes credible because the path to the recommendation is visible.

The Importance of an Experiment Ledger

For this reason, the experiment ledger is not an administrative detail. It is part of the architecture. Every policy evaluation should be recorded with the parameter values, simulator version, input data version, scenario set, random seed, runtime, objective score, KPI breakdown, constraint violations, and comparison to baseline. Without this record, the work becomes tribal knowledge. People remember that a certain setting was bad, but not why.

A policy is rerun later under different data and appears better or worse, but the team cannot explain the change. The simulator is updated, and old results become hard to interpret. The experiment ledger creates institutional memory.

A clean experiment history also allows the system to compound. The loop should know which policies have already been tested, which regions failed, which areas remain uncertain, which parameter interactions appear important, and which policies deserve more evaluation. Over time, the process becomes more intelligent because it is not starting from scratch with every run. It accumulates evidence.

Knowing When to Stop

The stopping rule is another practical concern. In theory, we can always run one more simulation. In practice, compute is not free, calendar time matters, and additional precision may not change the business decision. The process should stop when the expected value of another experiment is low relative to its cost. That might mean the best policy has not improved for a while. It might mean the top candidates are close enough that further simulation is unlikely to change the decision.

It might mean the remaining improvement is too small to justify added complexity. It might mean the recommended policy is good enough, robust enough, and operationally acceptable enough to move forward.

This last point is important because optimization can become a ritual if we are not careful. A tiny simulated improvement is not automatically valuable. If a policy improves the objective by a fraction of a percent but becomes harder to explain, harder to maintain, or more sensitive to assumptions, it may not be the better business decision. The goal is not to worship the objective function. The goal is to improve the operating policy.

Delivering a Decision Package

The final output of this methodology should be a decision package rather than a single answer. It should describe the recommended policy, compare it to the current baseline, show the KPI decomposition, explain the tradeoff frontier, identify where the policy performs well, identify where it remains fragile, summarize which parameters matter most, document which regions of the search space were ruled out, and explain what experiment would be run next if more budget were available.

That kind of output is much more useful than a black-box recommendation.

The Broader Lesson

The broader lesson is that simulation alone is not enough. A simulator can evaluate a policy, but it does not automatically create a better policy. Brute force is usually impossible. Manual tuning is slow and biased. Random search is better than nothing, but it leaves too much learning on the table. What is needed is a disciplined sequential loop that learns from each evaluation and uses that learning to decide what to evaluate next.

This is the architecture I keep returning to for messy supply chain problems. Define the policy space. Define the economic score. Run diverse initial experiments. Learn an approximate map of the policy landscape. Use that map to choose the next experiments. Balance exploration and exploitation. Eliminate bad policies early. Intensify promising policies. Preserve the underlying KPI tradeoffs. Track every experiment. Stop when the next run is no longer worth the cost.

From Simulation to Decision Engine

This is not a dashboard, a forecast, a static optimization model, or a one-time simulation study. It is a learning system around operational decisions. That distinction matters because most supply chain organizations already have more visibility than they can act on. They have dashboards, forecasts, reports, alerts, and scenario analyses. What they often lack is a disciplined way to turn all of that information into better operating policies.

At some point, the business has to decide how much inventory to hold, when to reorder, when to expedite, how to allocate constrained supply, which demand to protect, and how much volatility it is willing to absorb. These are decision questions. When the policy space is too large to brute force and the simulator is too expensive to use casually, the answer is not to simplify the problem until it becomes unrealistic. The answer is to build a smarter experimentation loop around the simulator.

The Guiding Principle

The most important principle is simple: do not waste expensive simulations. Every run should either find a better policy, disprove a bad one, reduce uncertainty, validate robustness, reveal an interaction, or clarify a tradeoff. When simulation is used this way, it stops being a reporting mechanism and becomes part of a decision engine.