LOGISTICS CASE STUDY / PREDICTION SYSTEMS

A Prediction Is Worth Nothing After the Plan Locks

Engineering a cloud prediction platform for delivery planning: value of information, latency budgets, queueing, capacity and decision-impact measurement

EXECUTIVE CONCLUSION

A prediction creates value only if it arrives in time to change a decision. Latency, throughput and capacity requirements therefore come from the planning window and the value of the decision, not from what is convenient to compute.

A major global logistics company wanted predictions to drive delivery planning and operational recommendations across its network, at a volume that made it one of the largest prediction workloads on its cloud platform. This note sets out the operations research behind a platform of that kind, built on Microsoft Azure: valuing a prediction as information with a deadline, deriving latency budgets, the queueing that governs delay, sizing pooled capacity, degrading gracefully under overload, and measuring whether the predictions actually improve plans.

Network scale prediction workload

Low latency and high throughput, at once

Azure cloud platform

Embedded in delivery planning workflows

Work led by Adam DeJans Jr. in an earlier industry role, not a Bit Bros client engagement. The company is not named and no performance or financial figures are published. The architecture and methods below are a reference description, not the company's implementation. Exhibits 1 to 3 and Tables 1 and 2 are synthetic models built for this note; the scripts that produce them are published with it.

The planning problem

Delivery planning turns the day's volume into routes, loads and staffing before vehicles leave. Predictions can improve almost every one of those choices: how long a stop will take, how much volume will arrive at a facility, which recommendations an operator should act on. But planning runs on a clock. Inputs are gathered, a plan is computed and the plan locks; after that, a better number changes nothing.

The platform therefore had to do two hard things at once: deliver a very large number of predictions per planning cycle, which is a throughput requirement, and deliver each one before its decision was made, which is a latency requirement. Its outputs also had to land where planners and planning systems already read them.

THE DECISION

How should a prediction service be engineered and sized so that every prediction a plan needs arrives before that plan locks, at network scale and acceptable cost, and is consumed by the planning process rather than admired in a dashboard?

Symbol	Meaning
X, F	Latency of one prediction call and its distribution function
b	Latency budget: time from request to plan lock
k	Fan-out: predictions a plan waits on in parallel
v, v_0	Value of a plan with the prediction and with its fallback
λ, μ	Arrival rate of requests; service rate of one replica
c, a, ρ	Replicas; offered load $a = \lambda/\mu$; utilization $\rho = a/c$
A, C, T	Per-batch overhead, per-item cost, latency budget of a batching replica
κ, ε	Cost of a replica per period; tolerated probability of a late call

1. Value a prediction as information with a deadline

A prediction is worth the decision it changes

Decision analysis gives the right unit of account. Let $a^*(x)$ be the plan chosen with prediction x and a_0 the plan chosen on the fallback. The expected value of the prediction is

$$\mathrm{VOI} = \mathbb{E}\big[u(a^*(x), \omega)\big] - \mathbb{E}\big[u(a_0, \omega)\big],$$

where u is the plan's realized value in state ω. Two consequences follow. A prediction that would not change the plan has zero value however accurate it is, and the predictions worth engineering hardest for are those on decisions that are both sensitive and expensive. Accuracy is an input to VOI, not a substitute for it.

The deadline turns value into a step function

Let a plan lock at t_ℓ. A prediction requested at t_r with latency X is used only if $t_r + X \le t_\ell$; otherwise the plan proceeds on its fallback. Writing $v - v_0$ for the VOI, the expected value of a request is

$$\mathbb{E}[V] = v_0 + (v - v_0)\ \Pr\{X \le b\}, \qquad b = t_\ell - t_r.$$

If a plan needs all k of its inputs, the value becomes $v_0 + (v - v_0)F(b)^k$; if each input improves the plan separately, values add and each term carries its own $F(b)$. The budget b is set by the planning process and then allocated across network transit, queueing, model computation and integration. Engineering's job is to make $\Pr\{X \le b\}$ high at the required volume and an acceptable cost.

The slowest call governs the plan

A plan rarely needs one prediction. If it waits on k predictions requested in parallel and each independently meets its budget with probability p, the plan is on time with probability p^k: at $k = 100$ and $p = 0.99$, it misses 63% of the time. Independence is optimistic in one respect and pessimistic in another. Latencies are often correlated through a shared dependency, so a single slowdown can delay every call in a plan at once; the plan's miss rate then approaches the probability of that event, which no per-call target controls. Exhibit 1 shows how quickly the requirement tightens. For a plan with 100 inputs to be on time 99% of the time, each call must meet its budget 99.99% of the time; with 1,000 inputs, 99.999%. Four responses are standard: reduce fan-out by batching requests per plan; send a hedged backup request after a delay chosen against the budget, which trims the tail for a priced amount of extra load (Table 1); propagate the plan's deadline with every call; and define a fallback so that no plan ever waits indefinitely.

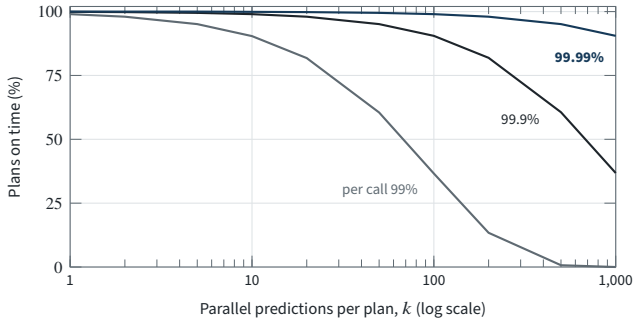

Exhibit 1. Share of plans on time, p^k, when a plan waits on k independent parallel predictions each within budget with probability p. Correlated slowdowns make the curves an optimistic bound.

The plan's latency is an order statistic

Formally, a plan that waits on k independent calls has latency $M_k = \max_i X_i$, with distribution $\Pr\{M_k \le x\} = F(x)^k$. Its q-quantile is therefore the single-call quantile at a much higher level,

$$F^{-1}_{M_k}(q) = F^{-1}\big(q^{1/k}\big),$$

so the plan's 99th percentile at $k = 100$ is the single call's 99.99th percentile. Hedging works on the other end of the order statistics. Two simultaneous requests return at $\min(X_1, X_2)$, with $\Pr\{\min > x\} = (1 - F(x))^2$, which squares the tail at the price of double load. A hedge sent only after a delay d returns at $\min(X_1,\ d + X_2)$ and costs extra load $1 - F(d)$.

Strategy	Call p99 (ms)	$F(60)$	Plans on time, k=100	Extra load
Single request	139.5	0.970	4.8%	0%
Hedge at p95 (41 ms)	64.2	0.983	18.5%	5%
Hedge at p90 (33 ms)	57.2	0.993	50.1%	10%
Hedge at p75 (26 ms)	50.6	0.997	75.4%	25%
Duplicate always	33.3	0.999	91.2%	100%

Table 1. Hedging a straggler-prone call (97% lognormal, median 20 ms; 3% six times slower) against a 60 ms budget. Synthetic, two million draws; script `hedging.py`.

Two lessons sit in Table 1. A hedge must leave room for the backup to finish: here d plus a typical call must fit inside b, which a hedge at the 95th percentile barely does. And hedging is a priced trade, not a free lunch; the extra load itself raises utilization and, through the queueing in Section 2, latency for everyone. Choosing d is a small optimization against the VOI of the plan.

2. Queueing governs delay

Latency explodes near capacity, and the tail first

Little's law, $L = \lambda W$, links the average number of requests in a service to their average time in it, for any stable system. In the single-server Markovian queue (M/M/1), time in system is exponential with rate $\mu - \lambda$, so $\Pr\{X > t\} = e^{-(\mu-\lambda)t}$ and its p-quantile solves $e^{-(\mu-\lambda)x_p} = 1 - p$:

$$x_p = \frac{\ln\big(1/(1-p)\big)}{\mu - \lambda}, \qquad x_{0.99} = \frac{4.61}{\mu - \lambda} = 4.61\,\mathbb{E}[X].$$

Both grow without bound as utilization $\rho = \lambda/\mu$ approaches one, and a fixed budget b caps utilization at $\rho \le 1 - 4.61/(\mu b)$ for a 99th-percentile target.

Variability costs as much as load

With Poisson arrivals and general service (M/G/1), the Pollaczek–Khinchine formula is exact for the mean wait,

$$\mathbb{E}[W_q] = \frac{\lambda\,\mathbb{E}[S^2]}{2(1-\rho)} = \frac{\rho}{1-\rho} \cdot \frac{1 + c_s^2}{2} \cdot \mathbb{E}[S],$$

so service-time variability enters through its second moment. For general arrivals as well, Kingman's approximation makes both sources explicit:

$$\mathbb{E}[W_q] \approx \frac{\rho}{1-\rho} \cdot \frac{c_a^2 + c_s^2}{2} \cdot \mathbb{E}[S],$$

where c_a and c_s are the coefficients of variation of inter-arrival and service times. Planning workloads are bursty by construction, since many plans open their windows at the same moment, which drives c_a^2 well above one. Staggering plan submissions, or smoothing them through a queue the platform controls, can cut waiting time as effectively as adding servers.

Batching buys throughput, and batch size regulates itself

Scoring requests in batches spreads fixed per-batch overhead across many predictions. Exhibit 2 models one serving replica with 20 ms overhead per batch, 0.5 ms per prediction, and a 10 ms limit on waiting for a batch to fill.

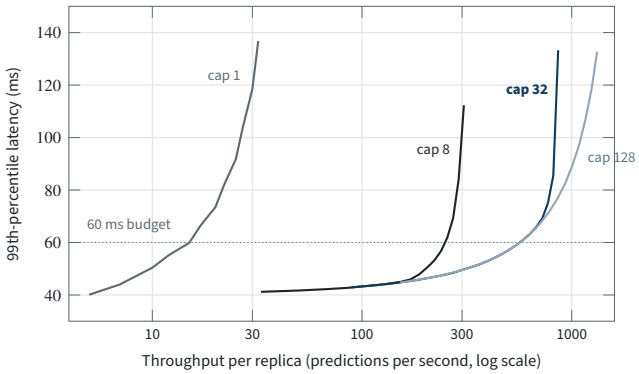

Exhibit 2. 99th-percentile latency against throughput for one serving replica at four batch-size caps, with a 60 ms budget (dotted). Synthetic queueing model, not platform measurements.

Within a 60 ms budget, an unbatched replica sustains about 15 predictions per second, a cap of 8 about 250, and caps of 32 and 128 the same 567. The tie is the instructive part, and a short derivation explains it. Under load, the next batch accumulates while the current one is computed, so the batch size n and cycle time τ satisfy $\tau = A + Cn$ and $n = \lambda\tau$. Solving,

$$\tau = \frac{A}{1 - \lambda C}, \qquad n = \frac{\lambda A}{1 - \lambda C},$$

about 16 predictions at 567 per second, whatever the cap. A cap above that natural size never binds; a cap below it, such as 8, throttles capacity. A request waits for the batch in progress and then for its own, so latency is roughly 2τ, and the budget $2\tau \le T$ gives

$$\lambda_{\max} \approx \frac{1 - 2A/T}{C},$$

667 per second here, which Poisson variability lowers to 567 at the 99th percentile. The levers are per-batch overhead A and per-item cost C: in the same model, halving A to 10 ms doubles usable throughput to about 1,130 per second, while changing the fill limit from 5 to 20 ms changes almost nothing.

Precompute what does not need to be fresh

Not every prediction must be computed inside the window. Quantities that change slowly can be scored in bulk ahead of time and cached; only those that depend on information arriving inside the window need online scoring. Moving work off the critical path is often the cheapest latency improvement available.

3. Size pooled capacity, then add headroom

Erlang C and the square-root rule

With c replicas sharing one queue, Poisson arrivals at rate λ and service rate μ per replica, the probability that a request waits longer than t is

$$\Pr\{W > t\} = C(c, a)\, e^{-(c\mu-\lambda)t}, \qquad a = \lambda/\mu,$$

where $C(c, a)$ is the Erlang C probability of waiting,

$$C(c, a) = \frac{\dfrac{a^c}{c!}\dfrac{c}{c-a}}{\displaystyle\sum_{n=0}^{c-1}\frac{a^n}{n!} + \frac{a^c}{c!}\frac{c}{c-a}}, \qquad a < c,$$

computed stably through the Erlang B recursion. Sizing is then a one-dimensional integer program,

$$c^* = \min\big\{c \in \mathbb{Z}_+ : C(c, a)\, e^{-(c\mu-\lambda)t} \le \varepsilon\big\},$$

solved by increasing c from $\lceil a \rceil$. Exhibit 3 finds the fewest replicas meeting a 20 ms queueing allowance at the 99th percentile and compares them with a fixed 70% utilization rule.

Peak rate (per s)	Load a	Erlang C replicas	Utilization	70% rule replicas
100	2.0	5	41%	3
1,000	20.5	25	82%	30
10,000	205.0	210	98%	293

Exhibit 3. Replicas needed so that $\Pr\{W > 20\text{ ms}\} \le 1\%$ in an M/M/c queue with 20.5 ms mean service, against a 70% utilization rule. Synthetic; before failure and scaling headroom.

A fixed utilization target is wrong at both ends. It under-provisions a small service, which needs 5 replicas where the rule gives 3, and over-provisions a large pooled one by about 40%. Pooling is the reason. In the Halfin–Whitt regime, staffing $c = a + \beta\sqrt{a}$ gives a delay probability that converges, as load grows, to

$$\Pr\{W > 0\} \to \Big[1 + \beta\,\Phi(\beta)/\phi(\beta)\Big]^{-1},$$

with Φ and ϕ the standard normal distribution and density, so spare capacity need grow only with $\sqrt{a}$. Exhibit 3 sits even more favorably: because its delay allowance (20 ms) is close to the service time, the exponential factor does most of the work and the required spare $c - a$ stays between 3 and 5 replicas from 2 to 615 units of load, with the implied β falling from 2.1 to 0.2. The practical corollary is to pool replicas behind one shared queue rather than giving each consumer its own silo, which forfeits the economy of scale.

Headroom the model does not see

Erlang C assumes Poisson arrivals, one shared queue and instant capacity. Production needs explicit additions: spare replicas for failures and deployments, and headroom for autoscaling lag, since a reactive scaler that waits for utilization to rise adds capacity minutes after a planning peak begins. Planning peaks are known in advance, so capacity should be scaled on the planning schedule rather than on observed load.

Price the service level

The right latency target is itself an economic choice. With replica cost κ per period and λ requests per period, choose c to minimize

$$\kappa\, c + \lambda\,(v - v_0)\,\Pr\{X > b \mid c\},$$

Because c is an integer, the first-order condition becomes a marginal one: add replicas while the expected value they save exceeds their cost,

$$\lambda\,(v - v_0)\big[\Pr\{X > b \mid c\} - \Pr\{X > b \mid c+1\}\big] \;\ge\; \kappa.$$

Marginal analysis is optimal when the miss probability is convex and decreasing in c, the property Dyer and Proll established for M/M/c delay; it sets the miss rate where one more replica saves exactly what it costs. High-VOI predictions justify tighter tails than low-VOI ones on the same platform.

ALGORITHM 1. PRICE THE REPLICA COUNT

Require: peak rate λ, service rate μ, budget b, value gap $v - v_0$, replica cost κ, headroom h

$a \leftarrow \lambda/\mu$; $c \leftarrow \lfloor a \rfloor + 1$ ▷ smallest stable pool
function Miss(c)
  return $C(c, a)\, e^{-(c\mu-\lambda)b}$ ▷ Erlang C tail
end function
while $\lambda(v - v_0)\,[\text{Miss}(c) - \text{Miss}(c+1)] \geq \kappa$ **do**
  $c \leftarrow c + 1$ ▷ marginal replica pays for itself
end while
return $\lceil c\,(1+h) \rceil$ ▷ failure and scaling headroom

Table 2 runs Algorithm 1 for the 1,000-per-second pool of Exhibit 3 at a replica cost of $0.50 an hour, across value gaps spanning three orders of magnitude.

Value gap per prediction	Replicas c^*	Utilization	Late share	Late cost per hour
$0.00001	24	85%	1.2%	$0.42
$0.0001	26	79%	0.083%	$0.30
$0.001	28	73%	0.0054%	$0.19
$0.01	29	71%	0.0013%	$0.47

Table 2. Algorithm 1 on an M/M/c pool: 1,000 requests per second, 20.5 ms mean service, 20 ms queueing allowance, $0.50 per replica-hour. Synthetic; script `price_replicas.py`.

A thousandfold increase in what a late prediction costs buys a tail almost a thousand times thinner for five more replicas, about 20% more capacity. In a pooled system, the tail is cheap; what is expensive is not knowing what a late prediction is worth.

4. Degrade gracefully under overload

When demand exceeds capacity despite planning, a late prediction is worth no more than its fallback and consumes capacity that on-time requests needed. Three mechanisms turn overload into graceful degradation. Admission control rejects a request immediately, with its fallback, when its predicted completion already exceeds its deadline. Deadline-aware scheduling serves requests by earliest deadline, or by VOI, instead of arrival order. And load shedding drops low-VOI work first. Each converts a system-wide miss into a controlled, priced loss.

The scheduling rule has theory behind it. On a single server, ordering work by earliest deadline minimizes the maximum lateness (Jackson's rule), so no other order can bring the worst request closer to its deadline; with VOI weights, the analogous rule ranks by weight per unit of work. Admission control needs only a completion estimate. A request arriving at time t with deadline $t + b$ joins a queue holding Q requests ahead of it on c replicas, and is admitted only if

$$t + \frac{Q+1}{c\,\mu} + \hat{s} \;\leq\; t + b,$$

where $\hat{s}$ is a high quantile of its own service time; otherwise its fallback is returned at once. Rejecting a request that cannot make its deadline costs nothing it would not have lost anyway, and it protects the requests behind it.

5. A reference architecture

The components below are what a platform of this kind needs. They describe the engineering pattern, not the company's specific implementation.

Component	Responsibility
Feature pipeline	One feature definition shared by training and serving, preventing training-serving skew
Model registry	Versioned models with lineage, so any plan traces to the model that informed it
Scoring service	Pooled, batched inference sized to peak load at the latency budget
Bulk scoring	Ahead-of-time predictions for slowly changing quantities, cached for the window
Admission and fallbacks	Deadline checks on entry and a defined value for every prediction that cannot arrive in time
Monitoring	Latency percentiles, input drift, and calibration of probabilistic outputs, per model and consumer
Release controls	Shadow scoring and canary rollout before a model reaches live plans
Planning interface	A contract with each planning step: schema, deadline and fallback

Monitoring is a set of hypothesis tests

Each monitor tests a specific hypothesis, which keeps alerts meaningful. Input drift compares the live distribution of a feature, binned with shares q_i, against its training distribution p_i through the population stability index,

$$\text{PSI} = \sum_i (q_i - p_i) \ln \frac{q_i}{p_i},$$

with values above roughly 0.25 conventionally read as material drift. Calibration of a probabilistic prediction is tested through its probability integral transform: if the predictive distribution $\hat{F}_j$ is correct, the values $\hat{F}_j(y_j)$ at the realized outcomes are uniform, so a skewed histogram reveals bias and a U-shape reveals overconfidence. A prediction interval at nominal level α should cover the outcome in a binomial share α of cases, which gives an exact test of coverage. And latency monitors track percentiles per consumer, because a platform-wide average hides the consumer whose plan is late.

6. Embed the prediction, then measure its effect

The final design principle was organizational as much as technical: predictions were integrated into delivery planning workflows rather than sitting in a separate analytics layer. A prediction that must be looked up, interpreted and re-keyed into a plan has already lost most of its latency budget to people. Writing it to where planning reads, with a contract for timing and fallback, is what turns a model into a decision input.

Measuring its effect calls for care. In a shared delivery network, plans interfere: routes, vehicles and facilities are common resources, so a simple A/B split of shipments violates the independence that the comparison assumes. Switchback experiments, which alternate the treatment across regions and time blocks, are the standard design for this setting and measure what matters: the change in plan cost and service, not the change in model accuracy.

Formally, the experimental unit is a region-period (r, t), assigned $W_{rt} \in \{0, 1\}$ at random, and Y_{rt} is the realized plan cost. With $\mathcal{B}$ the burn-in periods just after each switch, the estimator is

$$\hat{\tau} = \frac{\sum_{(r,t)\notin\mathcal{B}} W_{rt} Y_{rt}}{\sum_{(r,t)\notin\mathcal{B}} W_{rt}} - \frac{\sum_{(r,t)\notin\mathcal{B}} (1 - W_{rt}) Y_{rt}}{\sum_{(r,t)\notin\mathcal{B}} (1 - W_{rt})}.$$

It handles interference by construction: every shipment that shares vehicles, routes or facilities within a region and period receives the same treatment, so spillovers stay inside a unit rather than contaminating the comparison. Carryover across periods, such as vehicles positioned by yesterday's plan, is handled by choosing blocks longer than the carryover and discarding $\mathcal{B}$. Variance must be computed at the unit level, not per shipment, or the experiment will look far more precise than it is.

REFERENCE DECISION FLOW

Planning window and VOI → latency budget and fan-out → pooled, batched scoring → admission and fallbacks → prediction inside the plan → switchback measurement

7. Validate the platform before it carries plans

A prediction platform is itself a system under uncertainty, and it deserves the same evidence as the models it serves. Four tests belong before a model reaches live plans.

Test	What it establishes
Trace replay	Recorded planning-day request streams, replayed at one and two times volume, show the tail each consumer will actually see, including the burst at window open
Failure injection	Removing replicas or slowing a dependency mid-replay confirms that fallbacks fire and plans still lock on time
Shadow scoring	A new model scores live traffic without affecting plans, so its latency and its predictions are compared before release
Decision replay	Past plans recomputed with and without the predictions estimate value before any live experiment is run

The order matters. Trace replay and failure injection test the clock; shadow scoring and decision replay test the value; the switchback experiment in Section 6 then confirms the value live.

8. What the project reported

The platform delivered predictions at network scale and speed, supporting delivery planning and operational recommendations across the company's operations. No performance or financial figures are published for this work.

9. Limitations of this note

The exhibits use stylized models: Markovian queues and independent calls. Real request streams are burstier and service times heavier-tailed than these assumptions, which shifts every number in the direction of needing more capacity; the structural conclusions, that tails govern plans, that per-batch overhead governs batched throughput, and that pooled capacity scales with the square root of load, are robust to those changes.

10. Lessons that transfer

Start from the decision's clock and its value

Ask when the decision locks and what the prediction is worth to it before asking how accurate the model is. The answers set the latency budget, the tail target and the fallback.

Design for the tail, at the real fan-out

A plan waits for its slowest input. A per-call 99th percentile that looks excellent can still make most plans late.

Capacity is throughput within the budget, pooled

Raw throughput is not capacity, and a fixed utilization target is not a sizing method. Measure throughput at the latency budget, cut per-batch overhead, and pool replicas.

A USEFUL EXECUTIVE DIAGNOSTIC

For your most important prediction, when does the decision that uses it lock, what is it worth to that decision, and what happens when it is late?

Further reading

Dean, J. and Barroso, L. A. The tail at scale. *Communications of the ACM* 56(2), 2013.

Kingman, J. F. C. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57(4), 1961.

Halfin, S. and Whitt, W. Heavy-traffic limits for queues with many exponential servers. *Operations Research* 29(3), 1981.

Bojinov, I., Simchi-Levi, D. and Zhao, J. Design and analysis of switchback experiments. *Management Science* 69(7), 2023.

Dyer, M. E. and Proll, L. G. On the validity of marginal analysis for allocating servers in M/M/c queues. *Management Science* 23(9), 1977.

Gross, D., Shortle, J. F., Thompson, J. M. and Harris, C. M. *Fundamentals of Queueing Theory*, 4th ed. Wiley, 2008.

Jackson, J. R. Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, UCLA, 1955.

Gneiting, T., Balabdaoui, F. and Raftery, A. E. Probabilistic forecasts, calibration and sharpness. *Journal of the Royal Statistical Society B* 69(2), 2007.

Sculley, D. et al. Hidden technical debt in machine learning systems. *NeurIPS*, 2015.

About the case and Bit Bros

This paper draws on prediction-platform work led by **Adam DeJans Jr.** in an earlier industry role, where he led AI and automation initiatives for a major logistics company. It was not a Bit Bros client engagement. The company's identity, systems, models and data are not disclosed. Bit Bros is a boutique applied economics and optimization firm that helps operating teams price their decisions and build the systems that make them repeatable.

Adam DeJans Jr.
Technical Managing Director, Bit Bros LLC
Decision scientist specializing in optimization, prediction systems and decisions under uncertainty.

Engineer for the decision's clock.