The Case for Automated Probabilistic Reasoning for Network Performance Analysis

Jamie Kai
The University of British Columbia
Vancouver, Canada

Rachee Singh
Cornell University
Ithaca, USA

Mina Tahmasbi Arashloo
University of Waterloo
Waterloo, Canada

Abstract

The dominant approach in formal network analysis is all-or-nothing: a property must hold in every scenario, and is refuted with a single counterexample. That is the right standard for correctness. Performance, however, can almost always degrade somewhere. In this paper, we advocate for *conditional probabilistic reasoning* for performance analysis, i.e., reporting the *conditions*, as restrictions on the scenarios a system may face, under which degradation is likely. We envision such analysis as a loop between an *explorer* proposing conditions and a *checker* formally evaluating them, and we identify the combinations of modeling, exploration, and checking techniques for making this loop automated, scalable, and accessible to network practitioners. In three preliminary case studies, incast, LLM inference scheduling, and packet scheduling, our probabilistic checks find significant conditions that cannot be found by all-or-nothing checks.

CCS Concepts

• **Networks** → **Network performance analysis**; *Network performance modeling*; • **Theory of computation** → **Automated reasoning**; *Markov decision processes.*

Keywords

Network performance analysis, Probabilistic reasoning

ACM Reference Format:
Jamie Kai, Rachee Singh, and Mina Tahmasbi Arashloo. 2026. The Case for Automated Probabilistic Reasoning for Network Performance Analysis. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16-17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859965

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859965

1 Introduction

Understanding how a network performs is one of the oldest problems in our field, and yet we still approach it the way we always have. We simulate or emulate the network topology and protocols, or measure a deployed network to understand its performance under specific traffic patterns, bursts of requests, or failures. Each experiment exercises a single scenario, whereas a real system faces numerous scenarios that can lead to degraded network performance. We generalize from the few scenarios we have tried and hope the network behaves similarly in the rest.

Formal analysis promises more than the hope that all scenarios behave like the few we have tried. Given a symbolic model of the network and a property we care about, a verifier reasons about every scenario admitted by the model, and either proves the property or returns a *counterexample* that breaks it. We call this style of reasoning *all-or-nothing*, because the property either should hold across every scenario or is considered refuted by a single counterexample. All-or-nothing reasoning makes sense if the properties we care for are *correctness* checks, e.g., whether traffic can reach a forbidden host, a forwarding loop exists, or a configuration resolves DNS queries as intended. A correctness property should hold always, so even a single counterexample is a bug that operators should fix. So the verifier either establishes the property over all scenarios or gives us a concrete scenario that exposes a bug [35, 38, 42, 56, 57].

Performance analysis needs conditional reasoning. Consider three concrete performance degradation scenarios: (1) an output buffer overflows under incast at a top-of-rack switch [17], (2) an ML inference request misses its latency target while contending for GPU batch slots [3], and (3) a flow loses its fair share under a queuing scheduler. Just confirming the fact that these can happen conveys very little useful information to the operator, because every system degrades under some workload. So the all-or-nothing question of whether it degrades is close to a tautology.

What we want is a formal analysis framework that characterizes the *conditions* under which performance degradation occurs, as restrictions on the quantities that vary across

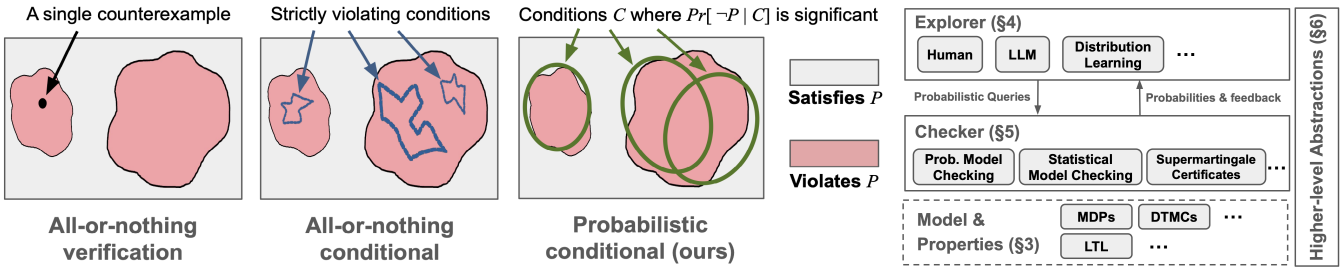


Figure 1: (left) The space of all scenarios, with regions violating the property P in red. All-or-nothing verification returns a single violating scenario. All-or-nothing conditional analysis accepts a condition only if it is entirely inside the violating region, forcing small, contorted conditions. Our conditional probabilistic framework finds conditions under which violation probability is significant enough to matter without forcing all scenarios within to violate the property. (right) Overview of our proposed framework.

scenarios, e.g., arrival rates, burst sizes, or synchronization degree. An operator can then look for those conditions in a deployment, engineer the design away from them, watch for them at runtime, or judge if the risk they pose is acceptable. In a nutshell, performance analysis requires more nuanced reasoning than an all-or-nothing verdict provides.

All-or-nothing conditional analysis is insufficient. Recent work on performance analysis already attempts to discover conditions that cause a property to hold or fail. FPerf [6] automatically searches for *workloads*, conditions on the input traffic that cause a contention point to perform poorly. CCAC [7], through manual iterations, accumulates restrictions on how an Internet path can behave until the property holds, i.e., the conditions under which the target congestion control algorithm performs well. Both, however, analyze conditions the all-or-nothing way. That is, the property must fail (or hold) on every scenario that satisfies the condition. A single exception disqualifies the condition.

To survive this test, conditions must trace the exact boundary of the violating region for FPerf and the satisfying region for CCAC. So, they tend to be either worst case scenarios and corner cases operators already expect or contorted predicates that are difficult to interpret. Both tools then add heuristics to clean up and broaden their own output. In a nutshell, they patch around the underlying mismatch: the all-or-nothing question is an awkward fit for performance analysis.

Our position. We argue that performance analysis should ask probabilistic questions from the start. Instead of requiring a condition to admit *only* failing scenarios, we instead ask two sets of probabilistic questions. First, among the scenarios admitted by the condition, does a violation occur with high probability? This limits how much benign behavior is included in a condition. Second, among the scenarios where the property fails, what fraction is captured by the condition? This limits how much of the failing region is missed by the condition. This is a conceptually simple, yet powerful relaxation. By adjusting the probability threshold of these two questions, an analyst can trade extreme worst-case conditions for simpler, broader ones that occur often, i.e., the mild but common regimes operators most want to know about.

We envision an effective conditional reasoning framework for network performance analysis as a loop between two roles: an *explorer* who proposes a condition, and a *checker* who analyzes the condition against a model and property, returning feedback that the explorer uses to propose the next condition (Figure 1). The explorer may be a person or an algorithm (§4), while the checker is a formal verifier (§5). Realizing this vision takes more than swapping the verifier for a probabilistic one. Probabilistic reasoning is a rich space: decades of work in formal methods offer many kinds of probabilistic models, specification logics, exact and approximate checking algorithms, and learning techniques, each with its own trade-offs. The technical challenge, and what makes this problem space unique and interesting, is finding the combination that fits network performance analysis: expressive enough to model contention points and their uncertainty, automated enough to drive the explorer–checker loop, scalable to the state spaces induced by such models, and accessible to network practitioners who are not formal-methods experts.

The bulk of this paper takes on that question. §2 defines the probabilistic checks and discusses their benefits over all-or-nothing analysis. §3 proposes MDPs as the models and LTL as the property language, a combination that opens up an array of automated and scalable analysis techniques. §4 automates exploration, through distribution learning and LLMs as explorers. §5 lays out techniques for scaling the checks, and §6 discusses how to make all of the above accessible to practitioners through domain-specific probabilistic programming languages, LLMs, and automated frameworks to automatically choose the right checks and techniques under the hood on the user's behalf.

Preliminary case studies. We ground the argument for probabilistic reasoning in three different case studies: incast at a data-center top-of-rack switch, batch scheduling for

LLM inference, and packet scheduling at an endpoint, each modeled as a DTMC and checked with the PRISM model checker with an LLM as an explorer. For each one, the all-or-nothing question recovers only the textbook worst case that operators already know, while the probabilistic question uncovers regimes that matter in practice and that all-or-nothing checks cannot find. In the incast study, for example, our probabilistic exploration finds a configuration that drops a non-trivial amount of traffic with probability 11% to 56% under far fewer and far more loosely synchronized senders than the worst-case scenario found by all-or-nothing checks.

2 Probabilistic conditional reasoning

We use $\vec{x}=(x_1,\ldots,x_n)$ for the variables representing the uncertainty at a contention point, e.g., timing and number of packet arrivals or link failures. A *scenario* is a single assignment to the variables, i.e., one way the uncertainty can resolve at run time. Consider incast, one of our case studies (§7), as the running example: N workers respond to a fanned-out request, each sending a burst of L packets, all starting within W time steps of each other (the "synchronization window"), and the bursts converge on the output buffer of the receiver's top-of-rack switch. The variables are N, W, L, and the workers' start times. The *model* M is a set of logical constraints over these variables that capture how the system behaves under that uncertainty, e.g., how the uplink buffers forward packets to the output buffer, and the enqueue, dequeue, and drop logic. The *property* $P(\vec{x})$ is the performance property of interest, e.g., the output buffer dropping less than $\frac{1}{4}$ its capacity, and a *condition* $C(\vec{x})$ is a predicate over the same variables that restricts the scenarios under consideration, e.g., a certain arrival burst pattern.

What probabilistic questions should we ask? All-or-nothing analysis asks if the property is violated in *every* scenario satisfying the condition: $\forall\vec{x}.\, C(\vec{x}) \wedge M(\vec{x}) \Rightarrow \neg P(\vec{x})$. The two questions we pose instead are about conditional probabilities under the model: **(1)** $\mathbf{Pr}_M[\neg P(\vec{x}) \mid C(\vec{x})] = ?$ asks, among the scenarios the condition admits, how often the property is violated, and **(2)** $\mathbf{Pr}_M[C(\vec{x}) \mid \neg P(\vec{x})] = ?$ asks, among the scenarios that violate the property, how often the condition captures them. Each check can also be posed against a threshold, e.g., $\Pr_M[\neg P(\vec{x}) \mid C(\vec{x})] \geq 1-\alpha$, where α bounds how much of the condition may be non-problematic. A similar threshold on (2) can bound how much of the problematic space it may miss.

Informative exploration signals. Probabilistic checks give explorers (human or not) more helpful feedback than a yes/no verdict and a single counterexample. Under an all-or-nothing check, a condition that just misses the boundary of the violating region is rejected exactly like one far from it, and the explorer learns nothing about which way to move. The outcomes of questions (1) and (2), in contrast, tell the explorer how much of the proposal is actually problematic, and how much of the problem it leaves out. Comparing outcomes across checks helps further, e.g., re-checking after adding or dropping an individual constraint says how much that constraint contributes to the violation, as we show in §7.

Revealing significant common-case performance problems. All-or-nothing analysis catches extreme worst-case problems (e.g., many tightly synchronized workers in the incast study). Probabilistic checks can catch those too, but their real value is finding conditions under which the property is violated frequently but not always. For incast, we find such a condition, a moderate number of loosely synchronized workers, under which the property is violated in 11% to 56% of scenarios (§7). These conditions always evade all-or-nothing analysis, yet are arguably the most valuable ones to report, as operators are more likely to be surprised by them than by already-well-known worst cases.

Where do underlying distributions come from? Even the least "informed" choice, a uniform distribution that weighs all scenarios equally, improves on all-or-nothing checks. When there is existing domain knowledge, e.g., typical burst sizes, failure rates, or, for incast, a worker likely to start late in the window, it can be slotted into the distribution. A mismatched distribution shifts the reported probabilities and can deflate the probability of a condition the explorer would otherwise find. That said, the distribution is a knob: distribution learning iteratively shifts it toward the scenarios that matter for the query, counteracting this and more (§4).

Won't probabilistic reasoning miss some cases? Probabilistic *verification* rightly raises this concern: a property that holds ≥99% of the time is hard to act on without knowing what the remaining 1% contains. However, our checks are not for safety certification but a *principled way* to find actionable characterizations of scenarios where violations are likely, while bounding the benign scenarios admitted by a condition and the problematic ones missed by it.

3 Defining models and properties

Markov models for contention points. Probabilistic checks require a precise model of the contention point that allows us to define and compute probabilities. We use *Markov decision processes* (MDPs) for this purpose. MDPs are state machines whose transitions may be either *probabilistic*, randomly determined according to a specified probability distribution, or *nondeterministic*, a control decision that is left unspecified [49]. To model a contention point as an MDP, one identifies the uncertainties in its environment (§2) that may influence the performance properties of interest, e.g., flow traffic patterns and sizes, characteristics of cross-traffic, or failures. Each uncertainty becomes a choice in the MDP:

probabilistic when we give it a distribution, whether from domain knowledge or as an uninformed default (§2), and nondeterministic when we leave it unspecified. In the incast study, for example, if each worker starts its burst at a time drawn uniformly from the synchronization window, the start times are probabilistic transitions. If we instead leave them unspecified, they are nondeterministic choices.

MDP vs. DTMC. An MDP can be paired with a *policy*, which assigns each nondeterministic choice a probability distribution. The resulting state machine has only probabilistic transitions and is called a *discrete-time Markov chain* (DTMC). A DTMC induces a probability distribution over *system state traces*, i.e., paths of states through the model. That is, every trace has a well-defined probability, so every check of §2 is available for a DTMC model M, a property P, and a condition C. If M is an MDP, some transitions have no probabilities attached. So, we cannot ask for exact probabilities, only their minimum and maximum, i.e., the least and greatest probability of $\neg P$ across all the ways the non-determinism could be resolved. The policy attaining an extreme is itself part of the answer, as it describes how the non-determinism has to be resolved for the property to be that likely (§4).

Properties as LTL formulas. To ask how likely a performance property is under the induced trace distribution, we must specify which traces we are asking about, i.e., those where the property holds. This makes *linear temporal logic* (LTL), which describes predicates over traces, a natural fit. LTL can state, e.g., that a predicate over system state should *always* hold on a trace, *eventually* hold, or hold *until* another does. The property in the incast case study, for example, asks that cumulative drops at the output buffer should *always* remain below a quarter of its capacity.

Why Markov models + LTL formulas? Two observations, which we expand on throughout the paper, ground this choice. First, MDPs (and their induced DTMCs) with LTL properties come with an array of techniques and tools from multiple disciplines for automated and scalable probabilistic reasoning (§4, §5). Second, established and emerging work around them, from probabilistic programming languages to translation between LTL and natural language, can make analysis accessible to non-specialists in formal methods (§6).

4 Automated condition exploration

In §2, we discussed how probabilistic checks are more helpful in providing feedback to explorers (human or not). Here, we discuss how to automate this exploration.

Distribution learning. One could view an MDP's policies (§3) as another way to characterize scenarios: a policy concentrates probability on some of the model's traces, picking them out the way a condition picks out the scenarios satisfying it. Policies are harder to interpret (§6 discusses remedies), but they do not commit in advance to which variables characterize the scenarios. Learning a distribution over the scenarios that lead to, say, property violations is thus the same problem as learning a policy for the MDP.

Distribution learning proposes, checks, and revises policies toward an "optimal" one, e.g., one maximizing the likelihood of violation. A promising example for our application is *logic-constrained reinforcement learning* (LCRL), which learns such policies by constructing product automata for MDPs and LTL specifications on-the-fly, and shaping its reward from the automata's acceptance criteria [30, 31]. Our preliminary experience here is limited but encouraging: we used our MDP encodings of network models to drive both the LCRL's black box model (the explorer) and the Storm model checker [33] (the checker), with the tabular LCRL algorithm. Applied to one of our case studies, LRCL validated the conditions we had found through other means. However, tractable performance required two adaptations: *reward shaping*, increasing reward proportionally to reaching query thresholds, which alleviates learning failures due to the sparse reward of the accepting automata, and *state projections*, restricting learning to only the variables relevant to a particular query, including derived ones such as the total backlog across several queues. We believe that further scaling with LCRL's neural-fitted Q-iteration (NFQ) is a promising direction for future work.

Large language models as explorers. An LLM is a natural candidate for the explorer's role. Proposing a good condition requires reasoning about the networking concepts and the formal model at the same time, and stating the result in terms an operator would recognize. LLMs have broad exposure to the former and are good at the latter. The concern, of course, is correctness. Here, a formal verifier computes the probabilities and decides whether a condition is acceptable, so a "bad" LLM proposal costs wasted computation, not an incorrect result. In our case studies (§7), an LLM was in fact the explorer, guided by high-level objectives given by humans, and the conditions we report were found this way, an encouraging indication for this direction. We leave a systematic evaluation of how effective it is, e.g., relative to a human explorer or distribution learning, to future work.

5 Scaling probabilistic reasoning

Probabilistic model checking typically requires constructing, from the given model M, a sparse transition matrix, a (multi-terminal) binary decision diagram (MT)BDD or a combination of these. These techniques are often termed *exact methods*, as they are guaranteed to (deterministically) compute the true probability of a query at an error tolerance ε, within a number of iterations that typically scale inversely to ε. However, off-the-shelf exact methods would be intractable even for some of our case studies (§7). For instance, PRISM [43], a

Scenario	Contended resource	Uncertainty ($\vec{x}$)	Property (P)
Incast	ToR output buffer	workers N, window W, bursts L	fewer drops than 1/4 of buffer capacity
Batch scheduling in inference	GPU worker batch slots	slots S, arrivals λ, mix p_{lp}, p_{ol}	no 3 iterations without a token
Packet scheduling	outgoing link (FQ-CoDel)	when each of five flows sends	no over-service while others backlogged

Table 1: The case studies cover different contended resources, uncertainties, and performance properties.

widely used probabilistic model checker, recommends that models contain no more than 10^9 states for exact methods, whereas our case studies have anywhere between 10^{24} and 10^{68} states, with an estimated 10^5 to 10^{35} of them reachable.

We therefore turn to *approximate methods*, which give probabilistic (not deterministic) guarantees of correctly computing the true probability of a query on a model within a given error bound. This typically manifests as a statistical guarantee of computing the true probability of a query, such as returning confidence intervals or satisfying false-positive and false-negative rates. Off-the-shelf probabilistic model checkers have limited support for approximate methods (e.g., PRISM simulations), which we discuss further in §7.

Statistical model checking (SMC). Rather than constructing the state space, SMC draws sample traces from the model and stops once it reaches a statistically sound conclusion, up to a user-supplied error tolerance, constructing only the states each trace visits. It suits our checks well: a loose tolerance is typically enough to tell whether a condition is significant (§7). Moreover, neither the *number of samples* SMC needs nor the *cost per sample* scale with the number of states in a model. Instead, the cost per sample scales only with the number of transitions from each state and the length of trace required (both of which are simple to bound by design), and the number of samples primarily depends on the user-specified error tolerance (lower tolerance requiring more samples) and the likelihood of the event of interest occurring (likelier events requiring fewer samples); the conditions we seek are exactly those under which violations are likely enough to cause significant problems in practice. For very unlikely queries, standard sampling techniques encounter the event too rarely to estimate[1]; *rare event simulation* can produce conservative estimates [14, 36, 50] in such cases; future work is to investigate this approach for critical performance properties, where failure tolerances are extremely low. Also, SMC typically does not handle the nondeterminism of MDPs, although variants such as Bayesian SMC have started to address it [13, 32, 37, 59], and dedicated methods exist for conditional reasoning in MDPs [9].

Supermartingale certificates are a proof method for LTL properties of MDPs[2] that bridge exact and approximate frameworks. A certificate is a function over the joint states of the model and query that acts as a lower bound on the probability that the query is satisfied, synthesized using SMT solvers [2, 34]. Such proofs can apply to large or infinite state spaces, and, for MDPs, certificates are obtained along with a policy inducing a DTMC with the desired probability bound. A promising recent approach uses reinforcement learning to find a *value function* witness of the certificate [1]. We did not investigate these in our case studies, but believe they are promising, particularly as the certificates themselves serve as proofs, thereby increasing trust in the checker.

Handling conditional probabilities. For a DTMC, the checks in §2 can be done using two unconditional queries, since $\Pr_M[\neg P \mid C] = \Pr_M[\neg P \wedge C] \,/\, \Pr_M[C]$. This is the approach we take while using PRISM (with its SMC engine) as a checker subroutine (§7). This does not work for an MDP that can only answer max(min) queries (§3), as the policy max(min)imizing the joint probability does not necessarily max(min)imize $\Pr_M[C]$, and dedicated algorithms may be needed [9]. Alternatively, a "joint" model of the condition and the original model generates only scenarios that satisfy the condition, and using that will make the check unconditional (e.g., $\Pr_{M\wedge C}[\neg P]$). The joint model also potentially sidesteps the SMC's rare-event problem. The probabilities in the above decomposition can both be rare even when their ratio is not. With joint models, which we use in our case studies, this is not a problem since for the conditions we explore, $\Pr_{M\wedge C}[\neg P]$ is not expected to be low.

6 Accessibility to practitioners

We now discuss techniques and future directions that can form the foundation of an end-to-end automated probabilistic performance analysis framework, accessible to practitioners without deep familiarity with formal methods.

Models as programs. Writing programs that compile to formal models for analysis has ample and successful precedent in networking [5, 10, 44]. We envision the same experience through *probabilistic programming languages* (PPLs), where program variables can be drawn from probability distributions, so a program denotes a probabilistic model. Work on compiling such programs, including networking-specific ones, to DTMCs exists [21, 22, 58], with ProbNetKAT an encouraging precedent [19, 53]. Building a (domain-specific) compiler for MDPs, i.e., for the nondeterministic choices some models may require (§3), is a promising avenue for

[1] A rule of thumb is that for an event with probability p, we should sample at least $100/p$ traces, which can become infeasibly large [36].

[2] More generally, ω-regular properties of probabilistic transition systems.

	All-or-nothing finds		Probabilistic checks find	
Scenario	**Found condition** (C_{obv})	**Prob.**	**Found condition** ($C_{\text{non_obv}}$)	**Prob.**
Incast	$N \in [20, 24]$, $W \in [0, 24]$, $L \in [8, 12]$ *many workers, tightly synchronized*	1	N=16, $W \in [184, 192]$, $L \in [10, 12]$ *fewer workers, window 15–19× the burst*	[0.11, 0.56]
Batch scheduling in LLM inference	$S \in [8, 12]$, $\lambda \in [0.5, 0.7]$, $p_{\text{ol}} \in [0.5, 0.7]$ *under-provisioned and overloaded*	1	$S \in [20, 24]$, $p_{\text{ol}} \in [0.28, 0.32]$, $p_{\text{lp}} \in [0, 1]$ *moderate batch, moderate long-output mix*	[0.12, 0.66]
Packet scheduling	*all condition clauses have equal weight*		*finds the explanatory clauses*	

Table 2: Benefits of conditional probabilistic reasoning. All-or-nothing checks find only the extreme regimes operators already expect, while probabilistic checks surface mild yet significant property violation regimes.

future work: MDP-based program semantics are well studied [28], and the domain-specific constraints of networking programs should keep such a compiler simple.

Specify and interpret in natural language. Performance specifications are often simple thresholds over metrics (§7, [6]), and translating natural-language specifications into LTL is an active research area [12, 48]. In the reverse direction, learned policies (§4) can be summarized as LTL formulas [15, 20] and translated back to natural language [18]. LLMs can also help write the models themselves. This further motivates using PPLs described above as auditing LLM output is much easier for a program than for a raw MDP.

Choose checks and techniques automatically. Even then, using this style of analysis effectively is not easy: the checks of §2 differ in cost, e.g., a threshold query is cheaper to decide than a property value one, and MDP min/max queries are more costly than DTMC, and the best-fit technique of §5 depends on the model, the query, and how rare the behavior of interest is. To relieve operators from navigating these choices, we envision a unified high-level interface where the user supplies the model, the properties, and the kinds of conditions they seek, and the framework decides internally which checks, heuristics, and techniques to apply.

7 Case studies

We use three case studies, incast, batch scheduling for LLM inference, and packet scheduling, to show the benefit of probabilistic conditional reasoning over its all-or-nothing counterpart. Each models a contention point (§2), and they differ in contended resources, mediation logic, and metrics of interest (Table 1). We write the models as DTMCs in PRISM model checker [43] with uniform default distributions (§2), and compute the checks with PRISM's sampling-based engine (§5). The studies use check (1) in §2, which we implemented as two unconditional queries in PRISM (§5). Our check (2) requires justifying a prior distribution over the condition's variables, which we leave to future work. The explorer is Claude Opus 4.8, prompted for conditions leading to violations with probability 0.1 to 0.7 (§4). PRISM is able to act as a checker subroutine in our explorer-checker loop (§5). Although PRISM supports sweeping across ranges of values using its *experiments* feature, we used an LLM explorer for incast and batch scheduling, as PRISM does not have capabilities for explorer tasks such as deciding on a useful subset of variables and ranges of values from domain knowledge.

Incast. Our running example models the receiver-side ToR switch: N workers respond to a fanned-out request, each sending a burst of L packets within a synchronization window of W steps. Responses arrive on 32-packet uplink buffers, two workers per uplink, each forwarding one packet per time step to the equally sized output buffer. Table 2 compares the conditions found for drops. $C^{\text{incast}}_{\text{obv}}$ (probability 1) captures too many workers sending too large a response within too small a window. Every scenario satisfying it violates the property but it tells operators what they already know. $C^{\text{incast}}_{\text{non_obv}}$ (probability 0.11–0.56) has fewer workers and shorter bursts spread over a window 15–19× the burst length. It cannot be found through all-or-nothing checks, yet its loss probability is operationally significant.

Batch scheduling in LLM inference. This case study models the batch scheduler of a GPU worker running chunked-prefill continuous batching [3]: at each iteration, one logical time step, every request in the batch advances by one output token or one prompt chunk, and a request holds one of the S batch slots until it finishes (the KV-cache pool is generously sized, so slots are the contended resource). The model tracks one interactive request against a background load with arrival rate λ and long-prompt/-output fractions p_{lp}, p_{ol}. The property is violated when the tracked request goes 3 consecutive iterations without a new token, i.e., misses its time-between-tokens SLO, defined as a multiple of a contention-free decode step [3]. Parameters and ranges are scaled down from real batch schedulers, preserving the ratios that matter[3]. In Table 2, $C^{\text{llm}}_{\text{obv}}$ (probability 1) is a heavily under-provisioned and overloaded worker, with few slots, arrivals near saturation, and mostly long-output requests, whose SLO violation operators would already expect.

[3]Prefill chunk is 512 tokens (matching existing systems), a long prompt is 2048 tokens, a short one is 16, a long output is 256 tokens, short one is 8. Batch slots are scaled down similarly.

$C^{\text{llm}}_{\text{non_obv}}$ (probability 0.12–0.66) has a moderate batch size and moderate long-output fraction. All-or-nothing checks cannot find it, yet it flags an operationally significant stall risk, useful for provisioning or runtime monitoring.

Packet scheduling. FPerf [6] models an FQ-CoDel-style scheduler with five input queues and synthesizes a four-constraint workload that starves one queue: The offending flow stays quiet for the first six steps (C1), then sends at most one packet every two steps (C2), accumulates at least five packets by the end of the horizon (C3), while the other four queues stay backlogged throughout (C4). Such a workload must pin down everything that could break the property to pass FPerf's all-or-nothing check. Such checks cannot tell which clauses matter most. With our probabilistic checks, perturbing one clause at a time confirmed C1 as a core clause, with a quiet period of four steps giving violation probability 0.68 and five steps 0.97, matching the scheduler's four-step warm-up during which the victim flows are absorbed into its lower-priority list. Probabilistic checks can thus decompose an existing all-or-nothing finding into an explanation, quantifying each constraint's contribution to the problem.

8 Related Work

Formal network performance analysis. A recent line of work uses logic-based formal methods for analyzing network contention points, buffers, and performance-related properties [6, 7, 45, 51, 52]. These works perform all-or-nothing checks, for single counterexamples or on conditions (§2). MetaOpt and MetaEase use multi-level optimization to find inputs that maximize a heuristic's gap from optimal [40, 46], i.e., worst-case findings. We advocate for conditional probabilistic reasoning for formal performance analysis.

XPlain [41] analyzes heuristics developed for network optimization problems that can be expressed as network flow problems such as traffic engineering and vector bin packing for VM placement. It produces a set of input regions, with each region defined using overlapping cubes expanding around one input point where the heuristic performs significantly worse than the optimal. XPlain adjusts which direction to expand a region on based on the density of bad samples (points with a significant gap between heuristic and optimal) in the existing sub-cubes. Our approach and XPlain are similar in finding conditions where not all input points but a significant portion of them result in performance problems. However, XPlain's approach is customized to network optimization problems that behave similarly for contiguous regions of the input space and does not attach probabilities to the regions. Our use of Markov models and LTL formulas allows for (1) expressing a rich set of network settings complementary to network optimization problems that can capture temporal interactions between packets and requests (§7) and (2) answering probabilistic questions with the desired level of precision.

Probabilistic network verification. ProbNV verifies control-plane properties under probabilistic failure models [26], NetDice computes the probability that a configuration satisfies forwarding properties under link failures [54], [39] profile stateful data planes probabilistically for adversarial testing, and the PPLs (§6) are used to give randomized forwarding a formal semantics [19, 21, 53, 58]. These works *verify* correctness and forwarding properties, probabilistically; our goal is *exploration*, finding conditions that cause *performance problems* at contention points.

Learning-based network analysis. A complementary line of work analyzes or stresses network performance without formal models: adversarial environments and inputs for protocols [4, 27], end-to-end analysis of learning-enabled systems [47], learned simulators and environments [8, 29, 55], and ML-assisted network calculus [23–25]. These approaches complement ours and can serve as the basis for explorers whose proposed conditions our formal checks evaluate.

Network performance analysis for traffic engineering. Traffic engineering solves a network flow optimization problem, where the quality of the solution can impact network performance (e.g., congestion and packet drops). Formulating such optimization problems for worst-case or all failure scenarios can lead to sub-optimal solutions that under-utilize the network under real-world traffic demands. Instead, TeaVar [11] proposes a probabilistic formulation based on shared risk groups (SRGs) that balances availability and utilization. Lancet [16] formulates the problem deterministically, but systematically scopes the analysis to failure scenarios that the network can feasibly handle. As discussed above, our approach is complementary to such work on network optimization problems.

9 Conclusion

The performance of a networked system can almost always degrade; the useful question is under which conditions and how often. We have made the case for conditional probabilistic reasoning for network performance analysis, assembled the models, checks, and techniques that can make such reasoning automated, scalable, and accessible, shown its potential in three case studies, and identified the research directions in this space.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback, and Diptarko Roy, Agustín Martínez Suñé and Alessandro Abate for their helpful discussion and suggestions regarding supermartingale certificates. This work was supported by NSERC grant CRC-2023-00035.

References

[1] Alessandro Abate, Daniel Contro, Mirco Giacobbe, Agustín Martínez-Suñé, and Diptarko Roy. 2026. Value Functions as Supermartingale Certificates. In *AI Verification*, Guy Avni and Christian Schilling (Eds.). Springer Nature Switzerland, Cham, 99–121.

[2] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2025. Quantitative Supermartingale Certificates. In *CAV*. Springer.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *USENIX OSDI*.

[4] Shehab Sarar Ahmed, William Sentosa, Yinjie Zhang, Yoav Lebendiker, Michael Shnaiderman, Tomer Gilad, Nathan H. Jay, Brighten Godfrey, and Michael Schapira. 2026. AdvNet: Revealing Performance Issues in Network Protocols by Generating Adversarial Environments. https://arxiv.org/abs/2605.00755v2

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM POPL*.

[6] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal Methods for Network Performance Analysis. In *USENIX NSDI*.

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward Formally Verifying Congestion Control Behavior. In *ACM SIGCOMM*.

[8] Sachin Ashok, Sai Surya Duvvuri, Nagarajan Natarajan, Venkata N. Padmanabhan, Sundararajan Sellamanickam, and Johannes Gehrke. 2020. iBox: Internet in a Box. In *ACM HotNets*.

[9] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. 2014. Computing Conditional Probabilities in Markovian Models Efficiently. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, Berlin, Heidelberg, 515–530. doi:10.1007/978-3-642-54862-8_43

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*.

[11] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TEAVAR: Striking the Right Utilization-Availability Balance in WAN Traffic Engineering. In *ACM SIGCOMM*.

[12] Andrea Brunello, Angelo Montanari, and Mark Reynolds. 2019. Synthesis of LTL Formulas from Natural Language Texts: State of the Art and Research Directions. In *26th International Symposium on Temporal Representation and Reasoning (TIME 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 147)*, Johann Gamper, Sophie Pinchinat, and Guido Sciavicco (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:19. doi:10.4230/LIPIcs.TIME.2019.17

[13] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. 2014. Verification of Markov Decision Processes Using Learning Algorithms. In *Automated Technology for Verification and Analysis*, Franck Cassez and Jean-François Raskin (Eds.). Springer International Publishing, Cham, 98–114. doi:10.1007/978-3-319-11936-6_8

[14] Carlos E. Budde, Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. 2020. An efficient statistical model checker for nondeterminism and rare events. *International Journal on Software Tools for Technology Transfer* 22, 6 (Dec. 2020), 759–780. doi:10.1007/s10009-020-00563-2

[15] Alberto Camacho and Sheila A. McIlraith. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. *Proceedings of the International Conference on Automated Planning and Scheduling* 29 (July 2019), 621–630. doi:10.1609/icaps.v29i1.3529

[16] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. 2020. Lancet: Better network resilience by designing for pruned failure sets. In *ACM SIGMETRICS*.

[17] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the ACM workshop on Research on enterprise networking*.

[18] Himaja Cherukuri, Alessio Ferrari, and Paola Spoletini. 2022. Towards Explainable Formal Methods: From LTL to Natural Language with Neural Machine Translation. In *Requirements Engineering: Foundation for Software Quality*, Vincenzo Gervasi and Andreas Vogelsang (Eds.). Springer International Publishing, Cham, 79–86. doi:10.1007/978-3-030-98464-9_7

[19] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems*. Springer.

[20] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive synthesis of temporal specifications from examples and natural language. In *ACM OOPSLA*.

[21] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *ACM PLDI*.

[22] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*. Springer.

[23] Fabien Geyer and Steffen Bondorf. 2020. On the Robustness of Deep Learning-Predicted Contention Models for Network Calculus. In *IEEE ISCC*.

[24] Fabien Geyer and Steffen Bondorf. 2021. Graph-Based Deep Learning for Fast and Tight Network Calculus Analyses. *IEEE Transactions on Network Science and Engineering* 8, 1 (Jan. 2021), 75–88. doi:10.1109/TNSE.2020.3025806

[25] Fabien Geyer, Alexander Scheffler, and Steffen Bondorf. 2023. Network Calculus With Flow Prolongation – A Feedforward FIFO Analysis Enabled by ML. *IEEE Trans. Comput.* 72, 1 (Jan. 2023), 97–110. doi:10.1109/TC.2022.3204225

[26] Nick Giannarakis, Alexandra Silva, and David Walker. 2021. ProbNV: probabilistic verification of network control planes. In *ACM ICFP*.

[27] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. 2019. Robustifying Network Protocols with Adversarial Examples. In *ACM HotNets*.

[28] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation* 73 (March 2014), 110–132. doi:10.1016/j.peva.2013.11.004

[29] Momin Haider, Ming Yin, Menglei Zhang, Arpit Gupta, Jing Zhu, and Yu-Xiang Wang. 2024. NetworkGym: reinforcement learning environments for multi-access traffic management in network simulation. In *NeurIPS*.

[30] Hosein Hasanbeig, Daniel Kroening, and Alessandro Abate. 2023. Certified reinforcement learning with logic guidance. *Artificial Intelligence* 322 (Sept. 2023), 103949. doi:10.1016/j.artint.2023.103949

[31] Mohammadhosein Hasanbeig, Daniel Kroening, and Alessandro Abate. 2022. LCRL: Certified Policy Synthesis via Logically-Constrained Reinforcement Learning. In *Quantitative Evaluation of Systems: 19th International Conference, QEST 2022, Warsaw, Poland, September 12–16, 2022, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 217–231. doi:10.1007/978-3-031-16336-4_11

[32] David Henriques, João G. Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. 2012. Statistical Model Checking for Markov Decision Processes. In *2012 Ninth International Conference on Quantitative Evaluation of Systems*. 84–93. doi:10.1109/QEST.2012.19

[33] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (Aug. 2022), 589–610. doi:10.1007/s10009-021-00633-z

[34] Thomas A. Henzinger, Kaushik Mallik, Pouya Sadeghi, and Đorđe Žikelić. 2025. Supermartingale Certificates for Quantitative Omega-Regular Verification and Control. In *CAV*. Springer.

[35] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-Net: Real-Time Network Verification Using Atoms. In *USENIX NSDI*.

[36] Cyrille Jegourel, Axel Legay, and Sean Sedwards. 2013. Importance Splitting for Statistical Model Checking Rare Properties. In *CAV*. Springer.

[37] Sumit K. Jha, Edmund M. Clarke, Christopher J. Langmead, Axel Legay, André Platzer, and Paolo Zuliani. 2009. A Bayesian Approach to Model Checking Biological Systems. In *Computational Methods in Systems Biology*, Pierpaolo Degano and Roberto Gorrieri (Eds.). Springer, Berlin, Heidelberg, 218–234. doi:10.1007/978-3-642-03845-7_15

[38] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive verification of DNS configurations. In *ACM SIGCOMM*.

[39] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic profiling of stateful data planes for adversarial testing. In *ACM ASPLOS*.

[40] Pantea Karimi, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Pooria Namyar, Mohammad Alizadeh, and Behnaz Arzani. 2026. Heuristic Analysis from Source Code via Symbolic-Guided Optimization. In *USENIX NSDI*.

[41] Pantea Karimi, Solal Pirelli, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, and Behnaz Arzani. 2024. Towards Safer Heuristics with XPlain. In *ACM HotNets*.

[42] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*.

[43] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: verification of probabilistic real-time systems. In *CAV*. Springer.

[44] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *ACM SIGCOMM*.

[45] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal abstractions for packet scheduling. In *ACM OOPSLA*.

[46] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. 2024. Finding Adversarial Inputs for Heuristics Using Multi-Level Optimization. In *USENIX NSDI*.

[47] Pooria Namyar, Michael Schapira, Ramesh Govindan, Santiago Segarra, Ryan Beckett, Siva Kesava Reddy Kakarla, and Behnaz Arzani. 2024. End-to-End Performance Analysis of Learning-Enabled Systems. In *ACM HotNets*.

[48] Daniel Neider and Rajarshi Roy. 2025. What Is Formal Verification Without Specifications? A Survey on Mining LTL Specifications. In *Principles of Verification: Cycling the Probabilistic Landscape*. Springer Nature Switzerland.

[49] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1 ed.). Wiley. doi:10.1002/9780470316887

[50] Gerardo Rubino and Bruno Tuffin. 2009. Introduction to Rare Event Simulation. In *Rare Event Simulation using Monte Carlo Methods* (1 ed.), Gerardo Rubino and Bruno Tuffin (Eds.). Wiley, 1–13. doi:10.1002/9780470745403.ch1

[51] Amir Seyhani, Aarti Gupta, David Walker, and Mina Tahmasbi Arashloo. 2026. Count-Based Abstractions for Performance Verification of Contention Points. In *USENIX NSDI*.

[52] Amir Seyhani, Junyi Zhao, Aarti Gupta, David Walker, and Mina Tahmasbi Arashloo. 2024. Buffy: A Formal Language-Based Framework for Network Performance Analysis. In *ACM HotNets*.

[53] Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *ACM PLDI*.

[54] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *ACM SIGCOMM*.

[55] Weikang Su, Tong Li, Wenzhen Huang, Zihan Huang, and Yong Li. 2026. GenNet: A Generative AI-Driven Mobile Network Simulator for Multi-Objective Network Optimization With RL. *IEEE Transactions on Mobile Computing* 25, 5 (May 2026), 6472–6483. doi:10.1109/TMC.2025.3637214

[56] VMware. 2019. VMware to Advance Network Monitoring with Acquisition of Veriflow. https://web.archive.org/web/20250617004101/https://blogs.vmware.com/management/2019/08/vmware-to-advance-network-monitoring-with-acquisition-of-veriflow.html. Archived copy; accessed October 2026.

[57] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM*.

[58] Han Zhang, Chi Zhang, Arthur Azevedo de Amorim, Yuvraj Agarwal, Matt Fredrikson, and Limin Jia. 2021. Netter: Probabilistic, Stateful Network Models. In *Verification, Model Checking, and Abstract Interpretation*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer International Publishing, Cham, 486–508. doi:10.1007/978-3-030-67067-2_22

[59] Paolo Zuliani, André Platzer, and Edmund M. Clarke. 2013. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design* 43, 2 (Oct. 2013), 338–367. doi:10.1007/s10703-013-0195-3