

Formal Cascade Propagation and Multi-Layer Availability Limits for In-Memory Infrastructure Resilience Simulation

Anonymous Author(s)

Abstract—Production chaos engineering tools inject real faults into running systems, creating risk for regulated operators. Classical reliability analysis (FTA, RBD) assumes component independence, which does not hold for cloud infrastructure where correlated failures span multiple layers simultaneously. We present two core technical contributions implemented in an open-source research prototype: (1) a cascade propagation engine formalized as a Labeled Transition System (LTS) over a dependency graph, with proven termination, $O(|V|+|E|)$ complexity, monotonicity, and bounded blast radius; and (2) an N -layer availability limit model composing categorical ceilings via min rather than the classical series-product. A post-hoc backtest on 54 real-world cloud incidents (2017–2024, 21 providers) achieves $F_1 = 0.87$, precision = 1.00, recall = 0.81 (severity accuracy = 0.58). We do not claim prospective predictive accuracy; the backtest is illustrative. The prototype is released under Apache 2.0 on PyPI.

I. INTRODUCTION

Cascading failures in cloud infrastructure—the 2021 Meta BGP outage [7], the 2020 AWS Kinesis cascade [8], the 2024 CrowdStrike incident—demonstrate how a single fault propagates across dozens of dependent services within minutes. Two gaps remain in current approaches:

Gap 1: Production risk. Chaos engineering tools [1], [2], [3], [4] inject *real* faults into running systems. For regulated industries under the EU Digital Operational Resilience Act (DORA, EU 2022/2554), touching production to discover weaknesses introduces unacceptable risk.

Gap 2: Independence assumption. Classical Reliability Block Diagrams [5] compose component availabilities via the series-product $A = \prod_i A_i$, assuming statistical independence. Cloud infrastructure violates this: a region outage simultaneously degrades compute, network, and storage.

Concurrent work by Krasnovsky [6] (ICSE-NIER 2026) proposes in-memory Monte-Carlo simulation over service-dependency graphs auto-discovered from distributed traces. Our work addresses the same high-level positioning with a different technical approach: formal LTS semantics with proven correctness properties and multi-layer availability decomposition. The two approaches are complementary, not competing.

Contributions.

- 1) A **cascade propagation engine** formalized as an LTS with proven termination, $O(|V|+|E|)$ complexity, monotonicity, causality, and bounded blast radius (§II).
- 2) An **N -layer availability limit model** using min-composition of categorical ceilings (§III).

- 3) A post-hoc **backtest on 54 real-world cloud incidents** spanning 21 providers and 7 years (§IV).

II. CASCADE PROPAGATION ENGINE

Definition 1 (Infrastructure Graph). $G = (C, E, dep, w, \tau, cb)$ where C is a finite set of typed components, $E \subseteq C \times C$ directed dependency edges, $dep : E \rightarrow \{., \}$, $w : E \rightarrow [0, 1]$ weights, $\tau : E \rightarrow \mathbb{R}_{\geq 0}$ latencies, and $cb : E \rightarrow \{T, F\}$ circuit breaker status.

Definition 2 (Cascade Propagation Semantics (CPS)). An LTS $\mathcal{M} = (S, s_0, Act, \longrightarrow, F)$ where each state $s = (H, L, T, V)$ consists of: health map $H : C \rightarrow \{\text{HEALTHY} < \text{DEGRADED} < \text{OVERLOADED} < \text{DOWN}\}$, latency map $L : C \rightarrow \mathbb{R}_{\geq 0}$, elapsed time T , and visited set $V \subseteq C$ (monotonically growing).

Eight transition rules govern: (R1) fault injection, (R2–R3) required dependency propagation (single vs. multi-replica), (R4–R5) optional/async attenuation, (R6) circuit breaker trip, (R7) timeout cascade, and (R8) termination. The key correctness properties are:

Theorem 1 (Termination). *For any finite G and fault f , the CPS terminates. V grows monotonically and is bounded by $|C|$; for cyclic graphs, a depth limit $D_{\max}$ and the visited set together guarantee termination within $\min(|C|, D_{\max} \cdot \Delta)$ transitions ($\Delta = \max \text{ out-degree}$).*

Theorem 2 (Complexity). *The CPS executes in $O(|C| + |E|)$ time: each component visited at most once, each outgoing edge examined once.*

Theorem 3 (Monotonicity). *Health only worsens: if $s \xrightarrow{*} s'$, then $\forall c. H'(c) \geq H(c)$.*

Theorem 4 (Blast Radius Bound). *Affected components are bounded by the reverse-reachable set: $affected(c) \subseteq \{c' \mid c' \xrightarrow{*} c \text{ in } G^{-1}\}$.*

These properties distinguish the CPS from Monte-Carlo sampling (which provides statistical but not formal guarantees) and from static SPOF detection (which does not model propagation semantics).

Key rules illustrated. Rule 2 (required dependency, single replica) captures the most common cascade pattern: if compo-

nent c' has a dependency on failed component c with a single replica, c' transitions to DOWN:

$$\frac{dep(c', c) = replicas(c) = 1 \quad H(c) = \text{DOWN}}{(H, L, T, V) \xrightarrow{prop(c, c')} (H[c' \mapsto \text{DOWN}], L', T, V \cup \{c'\})} \quad (1)$$

Rule 3 (multiple replicas) degrades rather than downs: $replicas(c) > 1$ yields $H[c' \mapsto \text{DEGRADED}]$. Rule 6 (circuit breaker) halts propagation: if $cb(c', c) = \text{true}$ and latency exceeds the timeout, c' transitions to at most DEGRADED and no further cascade passes through this edge. This formalization enables reasoning about circuit breaker placement as a graph-theoretic cut problem.

Comparison with Krasnovsky’s approach. Krasnovsky [6] uses a “connectivity-only topological model” with Monte-Carlo fail-stop sampling (Algorithm 1 in their paper: each replica removed with probability p_{fail} , BFS reachability check, 900,000 samples per failure fraction). This provides *statistical* resilience estimates ($\text{MAE} \leq 0.0004$ on DeathStarBench). Our CPS provides *deterministic* cascade paths with *formal* guarantees on termination and blast radius, but does not estimate resilience distributions. The approaches address complementary questions: “what is the expected resilience?” vs. “what is the exact cascade path and its formal properties?”

III. N-LAYER AVAILABILITY MODEL

We decompose system availability into five categorical layers:

$$A_{\text{sys}} = \min(A_{L1}, A_{L2}, A_{L3}, A_{L4}, A_{L5}) \quad (2)$$

where L1 captures software availability (deploy downtime, human error, config drift), L2 hardware (MTBF/MTTR with parallel redundancy), L3 theoretical physics (packet loss, GC pauses), L4 operational (incident response time, on-call coverage), and L5 external SLA dependencies ($\prod_i a_i$).

Why min rather than $\prod$? (1) Cloud layers exhibit *correlated* failure modes: a region outage degrades L2, L3, and L5 simultaneously, violating the independence assumption required by $\prod$. (2) The min operator captures a weakest-link interpretation: observed availability over a window is dominated by whichever category failed. (3) min is conservative (tighter bound) and interpretable (the dominating layer is immediately visible).

This is a *modeling choice*, not a universal theorem. Practitioners with genuinely independent layers should use the series-product. Within each layer, classical parallel-redundancy formulas are retained: $A_{\text{tier}} = 1 - (1 - A_{\text{single}})^n$.

Layer formulas. L1 (software): $A_{L1} = 1 - (d \cdot \bar{t}_{\text{deploy}} / T_{\text{month}} + p_{\text{human}} + p_{\text{drift}})$ where d = deploys/month, $\bar{t}_{\text{deploy}}$ = mean deploy downtime. L2 (hardware): per-component $A_{\text{single}} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$, composed as $A_{\text{tier}} = 1 - (1 - A_{\text{single}})^n$ with failover penalty. L4 (operational): $A_{L4} = 1 - n_{\text{inc}} \cdot \bar{t}_{\text{resp}} / (8760 \cdot p_{\text{cov}})$ where p_{cov} is on-call coverage fraction. L5 (external SLA): $A_{L5} = \prod_i a_i$.

The practical insight is immediate: if $A_{L5} = 0.999$ (three nines from external SLAs) but $A_{L2} = 0.9999$ (four nines from

TABLE I
PER-PROVIDER BACKTEST SUMMARY (54 INCIDENTS).

Provider	N	F_1	Sev. Acc.
AWS	11	0.846	0.64
GCP	7	0.952	0.57
Azure	6	0.889	0.57
Cloudflare	6	0.797	0.50
GitHub	5	0.880	0.59
Other (16 prov.)	19	0.857	0.57
All	54	0.866	0.58

hardware redundancy), then $A_{\text{sys}} = 0.999$ —the external SLA dependency is the binding constraint, not the hardware.

A. Related Work

Fault Tree Analysis (FTA; NASA NUREG-0492, IEC 61025) and Reliability Block Diagrams (RBD; Trivedi [5], IEC 61078) are the classical analytical methods. Both operate on static trees under component independence. HPE’s SPOF patent (US 9,280,409 B2) [10] identifies single points of failure in dependency graphs but does not model cascade propagation or availability decomposition. The I³ model (arXiv:2503.02890) uses graph autoencoders for cascade prediction in urban infrastructure—a different domain and method. Our contribution is the specific combination of formal LTS cascade semantics, min-composition of categorical availability layers, and open-source release for cloud infrastructure analysis.

IV. EVALUATION

Dataset. We assembled 54 real-world cloud incidents spanning 2017–2024 from 21 providers (AWS, GCP, Azure, Meta, Cloudflare, GitHub, Slack, Datadog, PagerDuty, DigitalOcean, Discord, Heroku, GitLab, Atlassian, Zoom, Okta, Fastly, CrowdStrike, Dyn, OVH, Roblox). Each incident has a public post-mortem URL, a hand-crafted topology graph, and an expected cascade path derived from the post-mortem narrative.

Methodology. For each incident, we constructed a representative topology, injected the documented root cause, and compared the predicted cascade path against the post-mortem. This is a *post-hoc illustrative* evaluation: topologies were built with knowledge of the outcome. We **do not** claim prospective prediction capability.

Results. Across 54 incidents: cascade path $F_1 = 0.866$ (precision = 1.000, recall = 0.810); severity accuracy = 0.577. Table I shows the per-provider breakdown. The engine processes a 50-component topology in <10 ms on commodity hardware.

Comparison with Krasnovsky 2026. Krasnovsky [6] evaluates on DeathStarBench with Monte-Carlo simulation ($\text{MAE} \leq 0.0004$). The two evaluations address different dimensions: Krasnovsky measures deviation from observed resilience on a benchmark; our backtest measures cascade path reproduction on real incidents. Neither constitutes prospective validation.

TABLE II
POSITIONING RELATIVE TO REPRESENTATIVE APPROACHES.

	Ours	K.'26	Chaos	RBD
In-memory	✓	✓	×	✓
Formal proofs	✓	×	×	partial
N -layer avail.	✓	×	×	×
Monte-Carlo	×	✓	×	×
Real injection	×	×	✓	×
54 incident eval.	✓	×	×	×
Open source	✓	✓	partial	×

V. FUTURE PLANS

This paper establishes the formal foundation and post-hoc validation of the cascade engine and N -layer model. The following steps are planned to develop this into a full-length paper:

- 1) **Prospective validation.** We will construct topologies for incidents that occurred *after* the model was frozen, inject the root cause without knowledge of the outcome, and measure precision/recall on unseen incidents. This is the critical missing evaluation for a predictive claim.
- 2) **DeathStarBench integration.** We will extract service-dependency graphs from DeathStarBench [9] and compare our cascade engine’s output with Krasnovsky’s Monte-Carlo results on the same topology, enabling direct quantitative comparison.
- 3) **Temporal layer correlation.** The current min model treats layers as static. We plan to extend it with a time-varying correlation model that captures how layer ceilings shift during incident windows.
- 4) **Formal mechanization.** We plan to mechanize the termination and monotonicity proofs in Coq or Lean 4 to eliminate the risk of proof sketch errors.
- 5) **User study.** We plan a controlled study with SRE practitioners to evaluate whether the N -layer decomposition produces actionable insights compared to single-number availability metrics.

Prospective validation design. The most critical next step is prospective validation. The design is as follows: (a) freeze the cascade engine code at a specific commit hash; (b) identify 20+ cloud incidents that occurred *after* the freeze date; (c) construct topology graphs from public post-mortems *without* consulting the affected-component list; (d) inject the documented root cause; (e) measure precision/recall against the post-mortem’s affected-component list. A blinded evaluator (not the engine developer) will construct the topologies to prevent knowledge leakage. This protocol follows established practice in fault localization research, where post-hoc evaluation on known bugs is a necessary but insufficient step, and prospective or cross-project evaluation is required for a predictive claim.

DeathStarBench comparison design. We will extract the Social Network and Hotel Reservation service-dependency graphs from DeathStarBench [9]’s Docker Compose configurations. For each extracted graph, we will: (a) run our

cascade engine with systematic single-fault injection on each component; (b) run Krasnovsky’s Monte-Carlo pipeline [6] on the same graph (900,000 samples per failure fraction, as specified in their Algorithm 1); (c) compare the predicted resilience rankings and cascade paths. This enables the first direct quantitative comparison between the formal LTS and Monte-Carlo approaches on identical topologies.

Formal mechanization. The proof sketches in this paper (Theorems 1–4) rely on standard graph-theoretic arguments (monotone set growth, BFS traversal bounds). Mechanizing these in Lean 4 or Coq would strengthen the contribution by eliminating the possibility of proof sketch errors. The Verified Software Toolchain community has established patterns for similar graph reachability proofs that we plan to adapt.

VI. LIMITATIONS

The cascade engine operates on a static topology and does not capture runtime dynamics (auto-healing, load balancer failover, Kubernetes rescheduling). Dependency types are simplified to three categories. The $F_1 = 0.866$ result is post-hoc and must not be interpreted as predictive accuracy on unseen topologies. The min-composition is a modeling choice, not a universal law. The cyclic graph termination relies on a depth limit ($D_{\max} = 20$), which is an engineering parameter rather than a theoretically derived bound.

To be explicit, we do *not* claim: (i) predictive accuracy on unseen topologies; (ii) superiority over production fault injection for empirical fault discovery; (iii) novelty over every prior work on cascade failure modeling; (iv) that min-composition is universally preferable to the series-product. What we *do* claim is that the specific combination of formal LTS cascade semantics, cross-layer min-composition, and open-source analytical implementation constitutes a useful operating point not previously disclosed in this combined form.

VII. DISCUSSION

The cascade engine and N -layer model serve different but complementary purposes in resilience analysis. The cascade engine answers “what breaks when component X fails?” with formal guarantees on the answer’s completeness (bounded blast radius) and computability (termination, linear time). The N -layer model answers “what is our availability ceiling, and which category of failure is the binding constraint?” with an interpretable decomposition that makes the weakest link visible.

Together, they enable a workflow that does not exist in current chaos engineering practice: an operator can (1) define their infrastructure as a dependency graph, (2) run the cascade engine to identify which components have the largest blast radius, (3) examine the N -layer decomposition to identify the binding availability constraint, and (4) prioritize resilience investments accordingly—all without touching production.

This workflow is not a replacement for production chaos engineering. Gremlin, Steadybit, and AWS FIS test real systems under real load with real monitoring signals. Our engine tests a *model* of the system. The two approaches answer different questions and should be used together where both are feasible.

REFERENCES

- [1] A. Basiri *et al.*, “Chaos engineering,” *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
- [2] Gremlin, Inc., “Gremlin: Chaos engineering platform,” 2020.
- [3] Steadybit GmbH, “Steadybit: Chaos engineering for Kubernetes,” 2023.
- [4] Amazon Web Services, “AWS Fault Injection Simulator,” 2021.
- [5] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed. Wiley, 2001.
- [6] A. A. Krasnovsky, “Model Discovery and Graph Simulation: A Lightweight Gateway to Chaos Engineering,” in *Proc. ICSE-NIER*, 2026. arXiv:2506.11176.
- [7] S. Janardhan, “More details about the October 4 outage,” Meta Engineering Blog, Oct. 2021.
- [8] Amazon Web Services, “Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region,” Nov. 2020.
- [9] Y. Gan *et al.*, “An Open-Source Benchmark Suite for Microservices,” in *Proc. ASPLOS*, 2019.
- [10] Hewlett Packard Enterprise, “Detecting single points of failure,” U.S. Patent 9,280,409 B2, 2016.