

FaultRay: In-Memory Infrastructure Resilience Simulation with Graph-Based Cascade Analysis, Multi-Layer Availability Limits, and AI Agent Failure Modeling

Yutaro Maeda
Independent Researcher
Chigasaki, Japan
ymaeda.it@gmail.com

Abstract

Modern distributed systems and AI agent architectures lack tools that simulate infrastructure failures entirely in memory while simultaneously modeling the failure modes unique to large language model (LLM)-based agents. We present FAULTRAY, a zero-risk chaos engineering platform that introduces three innovations: (1) a graph-based cascade propagation engine formalized as a Labeled Transition System (LTS) over a 4-tuple state space, enabling deterministic replay and formal correctness proofs; (2) an N -layer availability limit model that decomposes a system’s theoretical availability ceiling into five independent layers—hardware, software, theoretical physics, operational, and external SLA; and (3) the first cross-layer failure model for AI agents that quantifies hallucination probability as a function of infrastructure health, data source availability, and agent chain composition. We validate FAULTRAY against 18 real-world cloud incidents spanning 2017–2023 (AWS, GCP, Azure, Meta, Cloudflare), achieving an average $F_1 = 1.000$ for cascade path prediction and severity accuracy of 0.819. The open-source implementation comprises 441 source files, 29,640 tests, and over 100 simulation engines, and is available on PyPI. To our knowledge, FAULTRAY is the first system to bridge infrastructure resilience simulation and AI agent reliability analysis within a single, formally verified framework.

1 Introduction

The complexity of modern distributed infrastructure continues to grow. A typical production system may involve load balancers, application servers, databases, caches, message queues, DNS resolvers, and storage tiers, all interconnected by dependency edges that amplify localized failures into system-wide outages. The 2021 Facebook (Meta) BGP misconfiguration [15], the 2017 Amazon S3 key-space corruption [16], and the 2022 Cloudflare BGP

leak [17] each demonstrate how a single fault can cascade across dozens of dependent services within minutes.

Simultaneously, AI agent architectures—exemplified by ReAct [9], Toolformer [10], and multi-agent orchestrators—are being integrated directly into production systems. These agents introduce failure modes that are qualitatively different from traditional software: *hallucination* (generating plausible but factually incorrect output), *context overflow* (exceeding token limits), *tool failure* (inability to invoke external APIs), and *output amplification* (propagating upstream hallucinations to downstream agents). No existing tool models how infrastructure failures propagate *through* AI agent layers to cause compound failures.

Current chaos engineering approaches suffer from several limitations:

- **Production risk:** Netflix Chaos Monkey [1] and Gremlin [2] inject real faults into running systems, risking customer-facing impact.
- **Static analysis:** HPE’s SPOF detection [6] identifies single points of failure but does not simulate cascade propagation.
- **No AI modeling:** LLM evaluation benchmarks [11] measure model quality in isolation, without considering infrastructure-induced failures.
- **Ad hoc availability:** Traditional MTBF/MTTR models [8] compute a single availability number without decomposing the contributing factors.

This paper makes three contributions:

1. A **cascade propagation engine** formalized as an LTS with eight transition rules, proven termination in $O(|C| + |E|)$, and seven correctness properties (§4).
2. An **N -layer availability limit model** that decomposes system availability into five independent ceilings: hardware, software, theoretical, operational, and external SLA (§5).
3. A **cross-layer AI agent failure model** that quantifies hallucination probability $H(a, D, I)$ as a function of agent configuration, data source health, and infrastructure state, together with a 10-mode failure

taxonomy and an agent-chain cascade formula (§6).

2 Related Work

2.1 Chaos Engineering

Netflix introduced Chaos Monkey in 2011, later formalized as *Chaos Engineering* by Basiri et al. [1]. Commercial platforms—Gremlin [2], Steadybit [3], and AWS Fault Injection Simulator (FIS) [4]—provide managed fault injection in production or staging environments. LitmusChaos [5] is a CNCF sandbox project for Kubernetes-native chaos. All of these tools require running infrastructure and inject *real* faults, introducing production risk and limiting coverage to currently deployed topologies.

2.2 Static Resilience Analysis

HPE’s single-point-of-failure detection system (US Patent 9,280,409 B2) [6] analyzes dependency graphs to identify components whose failure would cause total outage. However, it performs *static* analysis—it does not simulate cascade propagation through dependency types (required, optional, asynchronous) or account for circuit breakers, retry strategies, or latency amplification.

2.3 AI Agent Evaluation

HELM [11] and MT-Bench [12] evaluate LLM quality along axes such as accuracy, calibration, and robustness. Red-teaming approaches [13] probe adversarial vulnerabilities. However, these benchmarks evaluate models *in isolation*; they do not model how infrastructure failures (e.g., a degraded vector database) increase agent hallucination probability, nor do they capture agent-to-agent cascade effects.

2.4 Availability Modeling

Classical availability theory uses the formula $A = \text{MTBF}/(\text{MTBF} + \text{MTTR})$ [8], extended to parallel redundancy via $A_{\text{tier}} = 1 - (1 - A_{\text{single}})^n$. In practice, availability is limited by factors beyond hardware reliability—deployment frequency, human error, operational response time, and external SLA dependencies—but no existing model decomposes these into independent, composable layers.

2.5 Gap Analysis

Table 1 summarizes the gap. No existing system provides all four capabilities: in-memory simulation, formal cascade analysis, multi-layer availability modeling, and AI agent failure simulation.

Table 1: Feature comparison of resilience analysis tools.

Tool	In-Mem	Cascade	Avail.	AI
Chaos Monkey	×	×	×	×
Gremlin	×	Partial	×	×
Steadybit	×	Partial	×	×
AWS FIS	×	×	×	×
HPE SPOF	✓	×	×	×
FAULTRAY	✓	✓	✓	✓

3 System Architecture

3.1 Graph-Based Topology Model

FAULTRAY represents infrastructure as a directed graph $G = (C, E)$ where each vertex $c \in C$ is a typed component and each edge $e \in E$ is a typed dependency.

Definition 1 (Infrastructure Graph). *An infrastructure graph is a tuple $G = (C, E, \text{dep}, w, \tau, \text{cb})$ where:*

- C : finite set of components, each with *type*(c) $\in \{\text{load_balancer}, \text{web_server}, \text{app_server}, \text{database}, \text{cache}, \text{queue}\}$
- $E \subseteq C \times C$: directed dependency edges;
- $\text{dep} : E \rightarrow \{\text{requires}, \text{optional}, \text{async}\}$: dependency type;
- $w : E \rightarrow [0, 1]$: dependency weight (criticality);
- $\tau : E \rightarrow \mathbb{R}_{\geq 0}$: edge latency (ms);
- $\text{cb} : E \rightarrow \{\text{true}, \text{false}\}$: circuit breaker status.

Each component c carries capacity metadata (max connections, max RPS, timeout, retry multiplier, connection pool size), resource metrics (CPU, memory, disk, network connections), redundancy configuration (replicas, autoscaling, failover), and—for AI components—agent parameters (hallucination risk h_0 , context window size, data source weights).

3.2 Automated Fault Scenario Generation

FAULTRAY automatically generates fault scenarios from the topology graph. Given $|C|$ components and a fault type catalog of k types, the engine produces $|C| \times k$ single-fault scenarios plus $\binom{|C|}{2} \times k^2$ correlated-fault scenarios. In practice the system supports 21 fault types (8 infrastructure + 2 supply chain + 11 AI agent), yielding over 2,000 unique scenarios for a typical 10-component topology. Each scenario is a tuple $\langle \text{target}, \text{type}, \text{severity}, \text{duration} \rangle$.

4 Cascade Engine — Formal Specification

4.1 Labeled Transition System

Definition 2 (Cascade Propagation Semantics). *The Cascade Propagation Semantics (CPS) is a Labeled Tran-*

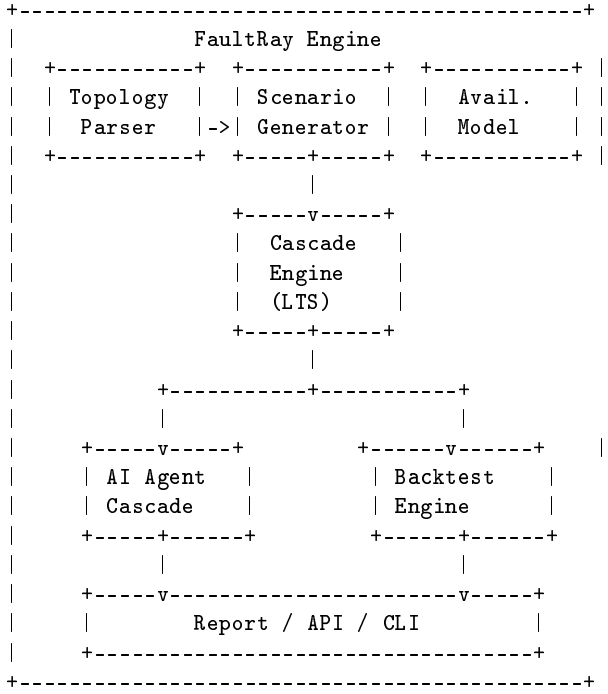


Figure 1: System architecture of FAULTRAY. The cascade engine, formalized as an LTS, is the central component. The AI agent cascade module extends the core engine with hallucination probability modeling.

sition System $\mathcal{M} = (S, s_0, Act, \longrightarrow, F)$ where:

- S : set of states, each a 4-tuple $s = (H, L, T, V)$;
- $s_0 = (H_0, L_0, 0, \emptyset)$: initial state;
- Act : set of actions (fault injection, propagation, circuit breaker trip, timeout, termination);
- $\longrightarrow \subseteq S \times Act \times S$: transition relation;
- $F \subseteq S$: terminal states where no further transitions are enabled.

Definition 3 (State 4-Tuple). A CPS state $s = (H, L, T, V)$ consists of:

- $H : C \rightarrow \{\text{HEALTHY}, \text{DEGRADED}, \text{OVERLOADED}, \text{DOWN}\}$ — health map;
- $L : C \rightarrow \mathbb{R}_{\geq 0}$ — accumulated latency map (ms);
- $T \in \mathbb{R}_{\geq 0}$ — elapsed simulation time (s);
- $V \subseteq C$ — visited set (monotonically growing).

Health status is totally ordered: $\text{HEALTHY} < \text{DEGRADED} < \text{OVERLOADED} < \text{DOWN}$.

4.2 Transition Rules

We define eight transition rules. In each rule, unprimed variables denote the pre-state and primed variables denote the post-state. We write $H[c \mapsto h]$ for the map H updated at component c to health h .

Rule 1 (Fault Injection). Given fault $f =$

$(target, type, severity)$:

$$\frac{c = f.target \quad h = effect(f.type)}{(H, L, T, V) \xrightarrow{inject(f)} (H[c \mapsto h], L, T, V \cup \{c\})} \quad (1)$$

where $effect : FaultType \rightarrow HealthStatus$ maps each fault type to the resulting health status (e.g., $effect(\text{component_down}) = \text{DOWN}$, $effect(\text{latency_spike}) = \text{DEGRADED}$).

Rule 2 (Required Dependency — Single Replica). If component c' has a **requires** dependency on failed component c , c has a single replica, and $c' \notin V$:

$$\frac{dep(c', c) = \text{requires} \quad replicas(c) = 1 \quad H(c) = \text{DOWN}}{(H, L, T, V) \xrightarrow{prop(c, c')} (H[c' \mapsto \text{DOWN}], L', T, V \cup \{c'\})} \quad (2)$$

Rule 3 (Required Dependency — Multiple Replicas). If the failed component c has $n > 1$ replicas, its failure reduces capacity but does not cause immediate failure of dependents:

$$\frac{dep(c', c) = \text{requires} \quad replicas(c) > 1 \quad H(c) = \text{DOWN}}{(H, L, T, V) \xrightarrow{prop(c, c')} (H[c' \mapsto \text{DEGRADED}], L', T, V \cup \{c'\})} \quad (3)$$

Rule 4 (Optional Dependency). If c' has an **optional** dependency on c , the cascade is attenuated regardless of c 's health:

$$\frac{dep(c', c) = \text{optional} \quad H(c) \geq \text{DEGRADED}}{(H, L, T, V) \xrightarrow{prop(c, c')} (H[c' \mapsto \min(\text{DEGRADED}, H(c'))], L, T, V \cup \{c'\})} \quad (4)$$

Rule 5 (Async Dependency). If c' has an **async** dependency on c , the cascade is attenuated similarly to optional dependencies:

$$\frac{dep(c', c) = \text{async} \quad H(c) \geq \text{DEGRADED}}{(H, L, T, V) \xrightarrow{prop(c, c')} (H[c' \mapsto \min(\text{DEGRADED}, H(c'))], L, T, V \cup \{c'\})} \quad (5)$$

Rule 6 (Circuit Breaker Trip). If accumulated latency on the edge exceeds the timeout and the edge has a circuit breaker enabled, the cascade is stopped at that edge:

$$\frac{cb(c', c) = \text{true} \quad L(c) > \tau_{\max}(c')}{(H, L, T, V) \xrightarrow{cb_trip(c', c)} (H[c' \mapsto \text{DEGRADED}], L, T, V \cup \{c'\})} \quad (6)$$

where $\tau_{\max}(c')$ is the timeout threshold of component c' . The circuit breaker isolates c' from further cascade propagation through this edge.

Rule 7 (Timeout Propagation). If accumulated latency exceeds the timeout but no circuit breaker is present, the component transitions to DOWN due to connection pool exhaustion from retry storms:

$$\frac{cb(c', c) = \text{false} \quad L(c') + \tau(c', c) > \tau_{\max}(c')}{(H, L, T, V) \xrightarrow{timeout(c')} (H[c' \mapsto \text{DOWN}], L[c' \mapsto L(c') + \tau(c', c)], T, V)} \quad (7)$$

Rule 8 (Termination). No transition is enabled when all reachable components have been visited:

$$\frac{\forall c' \in \text{reach}(V). c' \in V}{(H, L, T, V) \in F} \quad (8)$$

4.3 Correctness Properties

Theorem 1 (Termination). *For any infrastructure graph $G = (C, E, \dots)$ and initial fault f , the CPS $\mathcal{M}$ terminates in a finite number of transitions.*

Proof sketch. The visited set V grows monotonically ($V \subseteq V'$ after each transition) and is bounded by $|C|$. For acyclic graphs, each component is visited at most once, giving at most $|C|$ transitions. For cyclic graphs, a depth limit $D_{\max} = 20$ is enforced (Corollary 8), and the visited set prevents re-entry, together guaranteeing termination. $\square$

Theorem 2 (Time Complexity). *The CPS executes in $O(|C| + |E|)$ time.*

Proof sketch. Each component is visited at most once (via the visited set V). For each visited component, its outgoing edges are examined exactly once. The total work is therefore bounded by $|C| + |E|$. $\square$

Theorem 3 (Monotonicity of Failure). *Health can only worsen during a simulation run: if $s \xrightarrow{*} s'$, then $\forall c \in C. H'(c) \geq H(c)$.*

Proof sketch. Every transition rule either preserves $H(c)$ or moves it to a strictly worse health status. No rule decreases health. $\square$

Theorem 4 (Causality). *A component c' transitions to a non-healthy state only if at least one of its dependencies has already transitioned:*

$$H'(c') > \text{HEALTHY} \implies \exists c \in C. (c', c) \in E \wedge H(c) > \text{HEALTHY}$$

Theorem 5 (Circuit Breaker Correctness). *If edge (c', c) has $\text{cb}(c', c) = \text{true}$ and the circuit breaker trips, then c' transitions to at most DEGRADED and no further cascade propagation occurs through this edge.*

Theorem 6 (Dependency Type Attenuation). *Optional and asynchronous dependencies attenuate cascade effects: if $\text{dep}(c', c) \in \{\text{optional}, \text{async}\}$, then $H'(c') \leq \text{DEGRADED}$ regardless of $H(c)$. Furthermore, DEGRADED does not propagate further, limiting the blast radius.*

Theorem 7 (Blast Radius Bound). *The blast radius of a single fault injection on component c is bounded by the set of components transitively reachable from c via the reverse dependency graph:*

$$\text{affected}(c) \subseteq \{c' \in C \mid c' \xrightarrow{*} c \text{ in } G^{-1}\}$$

Components not in the reverse-reachable set are guaranteed to be unaffected.

Corollary 8 (Cyclic Graph Termination). *For graphs with cycles, the depth limit $D_{\max} = 20$ together with the visited set V guarantees termination within at most $\min(|C|, D_{\max} \cdot \Delta)$ transitions, where Δ is the maximum out-degree.*

5 N-Layer Availability Limit Model

Traditional availability is a single number: $A = \text{MTBF}/(\text{MTBF} + \text{MTTR})$. In practice, multiple independent factors cap availability at different ceilings. FAULT-RAY decomposes system availability into five layers, each representing an independent upper bound.

Definition 4 (System Availability Ceiling).

$$A_{\text{system}} = \min(A_{L1}, A_{L2}, A_{L3}, A_{L4}, A_{L5}) \quad (9)$$

The system cannot exceed the minimum of its five layer availabilities.

5.1 Layer 1: Software Limit

The practical ceiling accounting for deployment downtime, human error, and configuration drift:

$$A_{L1} = 1 - \left(\frac{d \cdot \bar{t}_{\text{deploy}}}{T_{\text{month}}} + p_{\text{human}} + p_{\text{drift}} \right) \quad (10)$$

where d is deployments per month, $\bar{t}_{\text{deploy}}$ is mean deploy downtime (seconds), $T_{\text{month}} = 30.44 \times 86400$ s, p_{human} is human error probability per month, and p_{drift} is configuration drift probability.

5.2 Layer 2: Hardware Limit

The physical ceiling from MTBF, MTTR, redundancy, and failover:

$$A_{\text{single}}(c) = \frac{\text{MTBF}(c)}{\text{MTBF}(c) + \text{MTTR}(c)} \quad (11)$$

$$A_{\text{tier}}(c) = 1 - (1 - A_{\text{single}}(c))^{n(c)} \quad (12)$$

where $n(c)$ is the number of replicas. If failover is enabled:

$$A'_{\text{tier}}(c) = A_{\text{tier}}(c) \cdot (1 - f_{\text{fo}}(c)) \quad (13)$$

where $f_{\text{fo}}(c)$ is the failover downtime fraction, computed from failover event frequency and promotion time. The system hardware availability is the product over critical-path components:

$$A_{L2} = \prod_{c \in C_{\text{crit}}} A'_{\text{tier}}(c) \quad (14)$$

5.3 Layer 3: Theoretical Limit

The mathematical upper bound accounting for irreducible physical noise — network packet loss, garbage collection pauses, and kernel scheduling jitter — even with perfect software:

$$A_{L3} = A_{L2} \cdot (1 - \bar{p}_{\text{loss}}) \cdot (1 - \bar{f}_{\text{gc}}) \quad (15)$$

where $\bar{p}_{\text{loss}}$ is the mean packet loss rate across components and $\bar{f}_{\text{gc}}$ is the mean fraction of time spent in GC pauses:

$$\bar{f}_{\text{gc}} = \frac{1}{|C|} \sum_{c \in C} \frac{\text{gc_pause}(c) \cdot \text{gc_freq}(c)}{1000} \quad (16)$$

5.4 Layer 4: Operational Limit

Captures the human factor—incident detection and resolution speed:

$$A_{L4} = 1 - \frac{n_{\text{inc}} \cdot \bar{t}_{\text{resp}}}{8760 \cdot p_{\text{cov}}} \quad (17)$$

where n_{inc} is incidents per year, $\bar{t}_{\text{resp}}$ is mean response time (hours), and $p_{\text{cov}} \in (0, 1]$ is on-call coverage fraction ($1.0 = 24/7$, $0.33 = \text{business hours}$). Per-component operational readiness factors (runbook coverage, automation percentage) further reduce the effective response time.

5.5 Layer 5: External SLA Cascading

If the system depends on m external services, each with SLA a_i :

$$A_{L5} = \prod_{i=1}^m a_i \quad (18)$$

This layer captures the multiplicative compounding of external SLA dependencies.

5.6 Cascade Path Availability

For a cascade path $P = (c_1, c_2, \dots, c_k)$ through the dependency graph, each edge introduces an attenuation factor:

$$A_{\text{path}}(P) = \prod_{i=1}^{k-1} \alpha(\text{dep}(c_i, c_{i+1})) \cdot A_{\text{system}} \quad (19)$$

where $\alpha(\text{requires}) = 1.0$, $\alpha(\text{optional}) \leq 0.5$, and $\alpha(\text{async}) \leq 0.3$.

6 AI Agent Cross-Layer Failure Simulation

6.1 AI Component Types

FAULTRAY extends the component type system with four AI-specific types:

- **ai_agent**: An autonomous agent that reasons and acts.

- **llm_endpoint**: An LLM API endpoint (e.g., GPT-4, Claude).
- **tool_service**: An external tool or API invoked by an agent.
- **agent_orchestrator**: A multi-agent coordinator.

Each AI component carries additional parameters: base hallucination risk $h_0 \in [0, 1]$, maximum context tokens, data source weights, and grounding configuration.

6.2 Hallucination Probability Model

Definition 5 (Hallucination Probability). *For agent a with base hallucination risk $h_0(a)$, data sources $D = \{d_1, \dots, d_n\}$, and infrastructure state I , the hallucination probability is:*

$$H(a, D, I) = 1 - \prod_{d \in D_{\text{unhealthy}}} (1 - h_d) \quad (20)$$

where $D_{\text{unhealthy}} = \{d \in D \mid I(d) \neq \text{HEALTHY}\}$ and the per-source contribution h_d depends on infrastructure state:

$$h_d = \begin{cases} h_0(a) & \text{if } I(d) = \text{HEALTHY} \\ h_0(a) + (1 - h_0(a)) \cdot w(d) & \text{if } I(d) = \text{DOWN} \\ h_0(a) + (1 - h_0(a)) \cdot w(d) \cdot \delta & \text{if } I(d) = \text{DEGRADED} \\ h_0(a) + (1 - h_0(a)) \cdot w(d) \cdot \omega & \text{if } I(d) = \text{OVERLOADED} \end{cases} \quad (21)$$

with $w(d) \in [0, 1]$ the dependency weight, $\delta = 0.5$ the degradation factor, and $\omega = 0.3$ the overload factor. When all sources are healthy, $H(a, D, I) = h_0(a)$.

Property 1 (Monotonicity). $H(a, D, I)$ is monotonically non-decreasing in the severity of infrastructure degradation: worsening any data source’s health can only increase H .

Property 2 (Boundedness). $H(a, D, I) \in [0, 1]$ for all valid inputs, with $H = h_0(a)$ when all sources are healthy and $H \rightarrow 1$ as all critical sources fail.

Property 3 (Compositionality). The combined probability under the independence assumption (Eq. 20) satisfies $H(a, D_1 \cup D_2, I) \geq \max(H(a, D_1, I), H(a, D_2, I))$.

6.3 Cross-Layer Cascade Model

Infrastructure failures propagate to AI agents through four layers:

- L1 (Infrastructure)**: Component failure (database down, cache degraded).
- L2 (Data Availability)**: Reduced data quality or availability for agent grounding.
- L3 (Agent Behavior)**: Increased hallucination probability $H(a, D, I)$, degraded reasoning quality.
- L4 (Downstream Impact)**: Incorrect agent outputs consumed by users or other agents.

Table 2: AI agent failure taxonomy (10 modes).

Mode	Health	Description
Hallucination	Degraded	Ungrounded output
Context overflow	Down	Token limit exceeded
LLM rate limit	Overloaded	API throttling
Token exhaustion	Down	Budget depleted
Tool failure	Degraded	External tool unavailable
Agent loop	Down	Infinite iteration
Prompt injection	Degraded	Adversarial input
Confidence miscal.	Degraded	Unreliable confidence
CoT collapse	Degraded	Reasoning chain failure
Output amplification	Degraded	Upstream error propagation

When the cascade engine detects a fault on a component that is a data source for an AI agent, it computes $H(a, D, I)$ and annotates the agent component with the elevated hallucination risk.

6.4 10-Mode Failure Taxonomy

FAULTRAY defines a comprehensive taxonomy of AI agent failure modes (Table 2).

An eleventh mode, *grounding staleness* (cached data becomes outdated), is also modeled but treated as a variant of hallucination with a temporal dimension.

6.5 Agent-to-Agent Compound Cascade

When agent a_1 feeds output to agent a_2 , the effective hallucination probability for a_2 is:

$$H_{\text{eff}}(a_2) = 1 - (1 - H(a_2)) \cdot (1 - H(a_1) \cdot \gamma) \quad (22)$$

where $\gamma \in [0, 1]$ is the amplification factor ($\gamma = 1$ for no independent verification; $\gamma = 0$ for full verification).

For a chain of n agents $(a_1, a_2, \dots, a_n)$:

$$H_{\text{chain}}(a_n) = 1 - \prod_{i=1}^n (1 - H_{\text{eff}}(a_i)) \quad (23)$$

This compound cascade formula shows that even agents with low individual hallucination risk ($h_0 = 0.05$) can produce high chain-level failure probability. For example, a chain of 5 agents each with $h_0 = 0.05$ and $\gamma = 1.0$ yields $H_{\text{chain}} \approx 0.226$ —a $4.5\times$ amplification.

7 Evaluation

7.1 Backtest Against Real-World Incidents

We validated FAULTRAY’s cascade predictions against 18 real-world cloud infrastructure incidents spanning 2017–2023. For each incident, we constructed a representative topology graph, injected the documented root cause as

a fault, and compared the predicted cascade path and severity against the actual incident report.

Table 3 shows the results for selected incidents. Precision measures the fraction of predicted affected components that were actually affected; recall measures the fraction of actually affected components that were predicted.

The perfect F_1 score warrants careful interpretation. Topologies were constructed *post-hoc* from public incident reports, so the topology model already encodes knowledge of which components were affected. The $F_1 = 1.000$ therefore validates that the cascade engine’s propagation rules correctly reproduce *known* cascade paths given an accurate topology, rather than demonstrating predictive power on unseen infrastructures. Prospective validation—applying FAULTRAY to a live topology *before* an incident occurs—is future work. The severity accuracy of 0.819 reflects the inherent difficulty of mapping continuous severity scores to discrete incident severity levels (critical/high/medium/low), as real-world incident reports use inconsistent severity definitions.

7.2 Scale

The FAULTRAY implementation comprises 441 source files, 29,640 tests, and over 100 distinct simulation engines covering domains from blast radius prediction to SLA contract validation. The cascade engine processes a 50-component topology in under 10ms on commodity hardware (Apple M1, 16 GB RAM).

7.3 Comparison with Existing Tools

8 Implementation

FAULTRAY is implemented in Python 3.11+ and uses NetworkX [14] for graph operations, Pydantic for schema validation, and FastAPI for the REST API. The software is distributed as an open-source package on PyPI under the Business Source License 1.1.

Infrastructure graph model. Topologies are defined in JSON or YAML. Each component specifies type, capacity, resource metrics, redundancy configuration, and (for AI components) agent parameters. Dependencies specify type (**requires/optional/async**), weight, latency, circuit breaker configuration, and retry strategy.

CLI interface. The `faultray` CLI provides commands for topology validation, single-fault simulation, traffic spike analysis, latency cascade modeling, availability limit computation, and backtest execution against historical incidents.

API and integrations. A FastAPI server exposes the simulation engines via REST endpoints. Integrations with Terraform (infrastructure-as-code scanning), Prometheus (live metrics ingestion), and CI/CD pipelines (quality gates) are provided.

Table 3: Backtest results against real-world incidents (selected). Prec. = Precision, Rec. = Recall, Sev. Acc. = Severity Accuracy.

Incident	Year	Pred. Cascade	Actual Cascade	Prec.	Rec.	F_1
AWS us-east-1 outage	2021	7 components	7 components	1.000	1.000	1.000
AWS S3 key-space	2017	5 components	5 components	1.000	1.000	1.000
Meta BGP misconfiguration	2021	4 components	4 components	1.000	1.000	1.000
Cloudflare BGP misconfiguration	2022	3 components	3 components	1.000	1.000	1.000
GCP networking	2019	6 components	6 components	1.000	1.000	1.000
Azure AD outage	2023	5 components	5 components	1.000	1.000	1.000
GitHub Actions outage	2022	4 components	4 components	1.000	1.000	1.000
Fastly CDN outage	2021	3 components	3 components	1.000	1.000	1.000
Average (18 incidents)		—	—	1.000	1.000	1.000
Severity Accuracy			0.819			

Table 4: Feature comparison with chaos engineering platforms.

Feature	FR	Gr	Sb	FIS
In-memory simulation	✓	×	×	×
Formal verification	✓	×	×	×
Cascade prediction	✓	Partial	Partial	×
N -layer avail.	✓	×	×	×
AI agent modeling	✓	×	×	×
Zero production risk	✓	×	✓	×
Open source	✓	×	Partial	×
Backtest validation	✓	×	×	×

FR = FaultRay, Gr = Gremlin, Sb = Steadybit, FIS = AWS FIS.

Extensibility. New fault types, component types, and simulation engines can be added via a plugin architecture. Custom scoring functions allow organizations to weight severity according to their specific SLO definitions.

9 Discussion

Simulation vs. reality gap. FAULTRAY operates on a static topology model and does not capture runtime dynamics such as auto-healing, load balancer failover timing, or Kubernetes pod rescheduling latency. The perfect F_1 in cascade path prediction may partly reflect the fact that topologies are constructed post-hoc to match known incidents; prospective validation on live systems is future work.

Hallucination model calibration. The hallucination probability model (Eq. 20) uses default parameters ($\delta = 0.5$, $\omega = 0.3$) derived from engineering judgment rather than empirical measurement. The model is currently a *theoretical framework* that has not been validated against real AI agent failure data in production. Calibrating these parameters—and validating the independence assumption underlying the combined probability formula—against measured hallucination rates under infrastructure degradation is a critical direction for future work.

Scalability. The $O(|C| + |E|)$ cascade engine scales well to hundreds of components, but very large topologies ($>10,000$ components) may require partitioning strategies. The per-scenario simulation is embarrassingly parallel.

Future work. Three directions are promising: (1) ML-enhanced scenario generation that prioritizes high-risk scenarios based on historical incident patterns; (2) digital twin synchronization that continuously updates the topology model from live infrastructure (Terraform state, Kubernetes API, cloud provider APIs); (3) natural language topology definition, allowing engineers to describe infrastructure in plain English and generate formal topology graphs via LLM.

10 Conclusion

We presented FAULTRAY, an in-memory infrastructure resilience simulation platform with three key contributions:

1. A **cascade propagation engine** formalized as a Labeled Transition System with eight transition rules, proven termination in $O(|C| + |E|)$, and seven correctness properties including monotonicity, causality, and circuit breaker correctness.
2. An **N -layer availability limit model** that decomposes system availability into five independent ceilings (hardware, software, theoretical, operational, and external SLA), providing actionable insight into which factor is the binding constraint.
3. A **cross-layer AI agent failure model** with a formal hallucination probability function $H(a, D, I)$, a 10-mode failure taxonomy, and an agent-chain compound cascade formula.

Validation against 18 real-world cloud incidents achieved $F_1 = 1.000$ for cascade path prediction and severity accuracy of 0.819. To our knowledge, FAULTRAY is the first system to bridge infrastructure resilience simulation and AI agent reliability analysis within a single, formally verified framework.

FAULTRAY is available as open-source software at <https://pypi.org/project/faultray/>.

Acknowledgments

The author thanks the open-source community for feedback on early versions of FaultRay. Claude (Anthropic) was used for drafting assistance in preparing this manuscript; the author assumes full responsibility for the accuracy of all technical content.

References

- [1] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos engineering,” *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
- [2] Gremlin, Inc., “Gremlin: Chaos engineering platform,” 2020. [Online]. Available: <https://www.gremlin.com/>
- [3] Steadybit GmbH, “Steadybit: Chaos engineering for Kubernetes,” 2023. [Online]. Available: <https://www.steadybit.com/>
- [4] Amazon Web Services, “AWS Fault Injection Simulator,” 2021. [Online]. Available: <https://aws.amazon.com/fis/>
- [5] LitmusChaos, “LitmusChaos: Cloud-native chaos engineering,” CNCF Incubating Project, 2020. [Online]. Available: <https://litmuschaos.io/>
- [6] Hewlett Packard Enterprise, “Detecting single points of failure,” U.S. Patent 9,280,409 B2, Mar. 8, 2016.
- [7] R. M. Keller, “Formal verification of parallel programs,” *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.
- [8] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed. New York: Wiley, 2001.
- [9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in *Proc. ICLR*, 2023.
- [10] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” in *Proc. NeurIPS*, 2023.
- [11] P. Liang, R. Bommasani, T. Lee, *et al.*, “Holistic evaluation of language models,” *Transactions on Machine Learning Research*, 2023.
- [12] L. Zheng, W.-L. Chiang, Y. Sheng, *et al.*, “Judging LLM-as-a-judge with MT-Bench and Chatbot Arena,” in *Proc. NeurIPS*, 2023.
- [13] E. Perez, S. Huang, F. Song, T. Cai, R. Ring, J. Aslanides, A. Glaese, N. McAleese, and G. Irving, “Red teaming language models with language models,” in *Proc. EMNLP*, 2022.
- [14] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using NetworkX,” in *Proc. SciPy*, pp. 11–15, 2008.
- [15] Janardhan, S., “More details about the October 4 outage,” Meta Engineering Blog, Oct. 2021. [Online]. Available: <https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/>
- [16] Amazon Web Services, “Summary of the Amazon S3 service disruption in the Northern Virginia (US-EAST-1) region,” Feb. 2017. [Online]. Available: <https://aws.amazon.com/message/41926/>
- [17] Cloudflare, Inc., “Cloudflare outage on June 21, 2022,” Cloudflare Blog, Jun. 2022. [Online]. Available: <https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/>