

Formal Verification of Circuit-Breaker Fleets Under Two Downstream Overload Semantics

Marian Ileana¹, Vassil Milev², Pavel Petrov²

¹Interdisciplinary Doctoral School, National University of Science and Technology POLITEHNICA Bucharest, Pitesti University Center, Pitesti, Romania

²Department of Computer Systems and Technologies, Faculty of Mathematics and Informatics, “St Cyril and St Methodius” University of Veliko Tarnovo, Bulgaria

Abstract

Circuit breakers protect a downstream dependency from a fleet of independent clients that retry after a shared failure. Engineering practice recommends adding jitter to the breaker's open-timeout so the fleet does not retry in lockstep, but the strength of this recommendation is rarely stated precisely: is jitter merely an efficiency improvement, or is it required for correctness? We show the answer depends on an assumption about the downstream dependency that is easy to leave implicit. Under fail-together overload semantics, where a batch of concurrent probes exceeding the dependency's capacity fails in its entirety, anunjittered fleet never recovers: explicit-state model checking finds a permanent livelock in every one of 14 tested fleet configurations. Under admission-control semantics, where only the first C arrivals of an overloaded batch are served, the same fleet always recovers, and jitter is merely beneficial: the exhaustive model predicts a 1.7x to 3.8x recovery speedup depending on fleet size, and an independent asyncio prototype confirms the effect directly, measuring a 2.2x speedup for the two fleet sizes it tests. We give a deterministic, modulo-based jitter assignment, prove its round-one safety bound by a pigeonhole argument cross-checked independently in Z3, and report a residual transient-overload limitation left open for future work.

Keywords: *circuit breaker, jitter, retry storm, explicit-state model checking, SMT verification, distributed web systems, resilience patterns*

1 Introduction

A circuit breaker protects a client from a downstream dependency that has started failing: after enough failures the breaker opens and short-circuits further calls, waits for an open-timeout, then allows a limited number of probe calls through in a half-open state to test whether the dependency has recovered [1, 2]. The pattern is one of the most widely deployed resilience mechanisms in distributed web systems, implemented in libraries such as Hystrix [3], resilience4j [4], and Polly [5], with a related host-ejection mechanism in service-mesh proxies such as Envoy [6], and documented as a standard cloud design pattern [7]. The same restart-storm risk is explicitly flagged in Kubernetes' own documentation for container health probes, which warns that an incorrectly tuned liveness probe can itself cause cascading failures under load [8].

A real deployment rarely runs a single breaker. A fleet of N stateless service instances typically each run their own local breaker instance guarding the same shared downstream dependency, with no coordination between instances. When the dependency fails, all N breakers observe the failure at roughly the same time and open together. If every instance uses the same open-timeout, they also attempt recovery at roughly the same time, sending a synchronized burst of probe requests. Practitioner folklore, most influentially an AWS Architecture Blog post on exponential backoff [9] and a companion Amazon Builders' Library article that also recommends circuit breakers as a complementary technique [10], recommends adding randomized jitter to the timeout so that instances spread their retries instead of retrying in lockstep. This advice is normally justified empirically or by simulation of a specific scenario (optimistic concurrency control on a shared object) rather than

proved for the circuit-breaker setting specifically, and it is usually framed as an efficiency recommendation: jitter reduces wasted work. The same recommendation, that every RPC retry implement backoff with jitter to avoid amplifying overload into a cascading failure, appears as a general operational rule in Google's site reliability engineering practice [11], again as a rule of thumb rather than a proof for any specific mechanism.

This paper asks a sharper question: for a fleet of circuit breakers, is jitter merely beneficial, or can its absence be a correctness bug, in the sense that the fleet provably never recovers no matter how long it waits? We show the answer is genuinely both, depending on one modeling choice that is easy to leave implicit when describing a downstream dependency: what happens when concurrent probes exceed its capacity. We call the two natural choices fail-together semantics (the whole batch fails, as when an overloaded connection pool or shared cache starts timing out for every caller) and admission-control semantics (only the first C arrivals are served, as when a semaphore or load balancer gates admission). Our contributions are:

- 1) a discrete-event formal model of an N -breaker fleet with exponential backoff, capped at a maximum timeout, guarding a dependency with a real recovery time and a concurrency capacity C , under both overload semantics;
- 2) an explicit-state verification, exhaustive for each concrete configuration because the model is deterministic once the jitter assignment is fixed, showing that theunjittered fleet livelocks under fail-together semantics in all 14 configurations tested (N in $\{3,5,7,9,11\}$, C in $\{1,2,3\}$) but always recovers under admission-control semantics;
- 3) a deterministic modulo jitter assignment that eliminates the livelock in all 14 fail-together configurations, together with a round-one safety proposition proved by pigeonhole and cross-checked independently in Z3 across 23 (N,C) pairs up to $N=50$;
- 4) a characterization of a residual transient-overload limitation of the modulo scheme on later retry rounds, together with a tested but unsuccessful attempted fix, left open as future work; and
- 5) an independent asyncio prototype that empirically confirms the admission-control predictions, matching the abstract model's round structure and admission pattern exactly and measuring a 2.2x recovery speedup from jitter for the two fleet sizes it tests, with a discretization-gap analysis explaining the small, well-understood numerical difference between predicted and measured tick counts.

The rest of the paper is organized as follows. Section 2 relates this work to the circuit-breaker literature and to prior model-checking work by the authors. Section 3 gives the formal model. Section 4 describes the verification method. Section 5 reports the model-checking and SMT results. Section 6 reports the prototype validation. Section 7 discusses the residual limitation and threats to validity. Section 8 concludes.

2 Background and Related Work

The circuit breaker pattern was popularized by Nygard [1] as part of a catalog of stability patterns for production software, and independently described and named by Fowler [2]; both works, together with the standard cloud-pattern write-up [7], agree on the canonical state machine of closed, open, and half-open states. Practitioner documentation typically discusses the pattern per-instance and is largely silent on fleet-wide interaction, treating each breaker as if it guarded a dependency of unbounded capacity.

The need for randomized jitter in retry logic is best known from an AWS Architecture Blog post [9], which simulates optimistic concurrency control on a single shared object: N clients contend to update the same object, one succeeds per round under plain backoff, and the number of rounds to drain the population grows linearly with N while total client work grows quadratically; adding jitter to the backoff interval breaks the synchronization and substantially reduces both quantities. That analysis

is scenario-specific and stops short of a general correctness claim, as is the cascading-failures guidance discussed above [11]. The broader problem of taming tail latency in large-scale distributed systems, including techniques such as hedged requests that are conceptually related to a bounded number of retries, is surveyed by Dean and Barroso [12]. The broader phenomenon of a system that degrades under a transient stressor and then fails to recover even after the stressor is removed is studied under the name of metastable failure [13], and the related notion of gray failure [14] highlights why the genuine-outage and self-inflicted-overload conditions in our model are hard to tell apart operationally: a fleet's own failure detectors may not distinguish a dependency that is still down from one that has recovered but is being overwhelmed by the fleet itself. To our knowledge, no prior work states or verifies conditions under which the absence of jitter causes a fleet of breakers to never recover, as opposed to merely recovering slowly.

This gap sits alongside a broader question we have studied before in a different guise: how a fleet of independent instances shares a constrained downstream resource without central coordination. Our own dynamic load balancing algorithm for distributed web systems [15] routes normal-condition traffic across replica servers using real-time load signals, the same many-clients-one-shared-resource structure as the circuit-breaker fleet studied here, though for a different goal, spreading steady-state load rather than staggering post-failure retries, and without a formal correctness argument. The present paper supplies that formal argument for the retry and recovery setting specifically, and the same admission-style reasoning reappears below in the definition of admission-control semantics (Section 3.2). More broadly, resilience under partial failure is a further dimension of a wider research program on distributed web systems that has also examined IoT intrusion detection [16], distributed cloud architectures for CRM platforms [17], and distributed web system performance within an educational assessment framework [18].

3 System Model

3.1 Breaker dynamics

We model N independent breaker instances guarding one shared downstream dependency in discrete time. All N breakers trip open simultaneously at tick 0, the realistic worst case for a fleet that calls the same failing dependency. Breaker i has a phase in $\{\text{OPEN}, \text{RECOVERED}\}$ (RECOVERED is absorbing), a backoff level starting at 0, and a timer, the same timer-and-guard structure used in timed automata [19], as implemented in tools such as Uppaal [20], to constrain discrete transitions with real-valued clocks.

Figure 1 depicts the pattern in its familiar three-state form (CLOSED, OPEN, HALF-OPEN), matching the standard presentation in the literature [1, 2], so that the correspondence to the formal model is explicit. The formal model collapses HALF-OPEN into an instantaneous decision taken the moment a probe is sent, so no persistent HALF-OPEN state is needed; and CLOSED is reached only once, as the absorbing RECOVERED state, because this model analyzes a single failure-and-recovery episode rather than the fleet's entire operational lifetime, during which a real breaker could of course trip open again after a later, independent failure. The base duration at level L is

$$\text{duration}(L) = \min(T_{\max}, T_0 \times 2^L) \quad (1)$$

with T_0 the base open-timeout and $T_{\max}$ a cap (we use $T_0=2$ and $T_{\max}=8$ ticks throughout, reached at $L=2$). Breaker i 's timer is initialized at tick 0 to $\text{duration}(0) + \text{jitter}_i$. After each failed probe, the breaker's level is incremented first (capped at $L=2$), and its timer is reset to $\text{duration}(L) + \text{jitter}_i$ using the updated level L . At each tick, a breaker whose timer is already 0 fires a probe; otherwise, a positive timer is decremented by one. This check-then-decrement convention is used throughout.

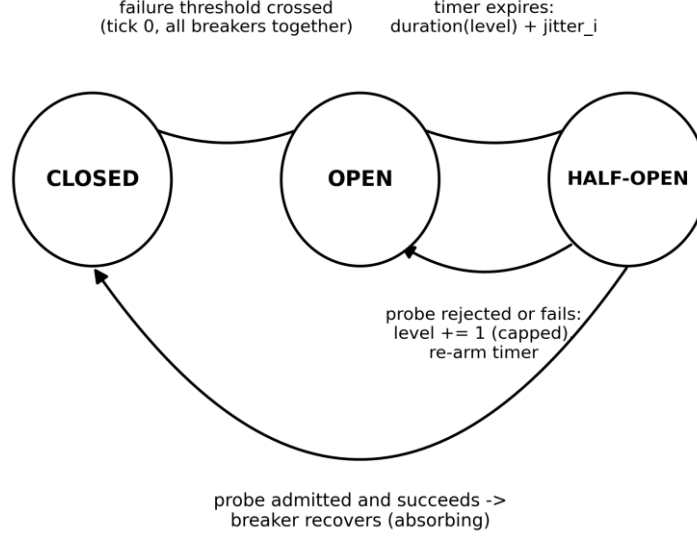


Figure 1. Circuit-breaker state machine. At HALF-OPEN, k = number of breakers probing on the same tick; under fail-together semantics all k probes fail if $k > C$, under admission-control semantics the first C succeed and the remaining $k-C$ fail (Section 3.2).

3.2 Downstream dependency and two overload semantics

The dependency has a genuine recovery tick D (before D , every probe fails regardless of concurrency) and a concurrency capacity C ; we use $D=7$ ticks throughout. Let k be the number of breakers that fire on the same tick. We model two semantics for what happens when $k > C$ at or after D :

Fail-together: all k probes fail. This models a shared resource that degrades wholesale under load, such as a connection pool or cache that begins timing out for every caller once saturated.

Admission-control: the first C firing breakers (in index order, matching the FIFO order of a lock-protected shared resource) succeed and the remaining $k-C$ fail. This models a semaphore or load-balancer-style gate that admits only its first C concurrent callers, the same admission principle as our own load balancing algorithm cited above [15] and as the thread-and-semaphore isolation used by Hystrix [3].

Under both semantics, a failed probe increments the breaker's level (capped at $L=2$, where duration saturates) and resets its timer to $\text{duration}(\text{new level}) + \text{jitter}_i$.

3.3 Jitter assignment

We compare two schemes. **None:** $\text{jitter}_i = 0$ for all i , the common naive default. **Modulo:** $\text{jitter}_i = i \bmod m$ for a chosen modulus m , a deterministic, designed assignment rather than a random one, chosen so that the analysis in Section 5.1 is exact rather than probabilistic. Section 7 also reports a third, rotating scheme tested as an attempted refinement.

4 Verification Method

4.1 Explicit-state simulation with cycle detection

Because breaker dynamics, the initial condition, and both overload semantics are fully deterministic once $(N, C, \text{jitter assignment}, T_0, T_{\text{max}}, D)$ are fixed, a single forward simulation from the initial state is exhaustive for that configuration: it either reaches the all-RECOVERED state, establishing liveness and the tick at which it is reached, or it revisits a previously-seen global state before doing so, which is only possible in a deterministic system by re-entering an exact repeating cycle, i.e. a livelock. We detect this by memoizing every visited global state $(t_{\text{eff}}, \text{breaker states})$ in a hash table

during the forward simulation; a repeated state yields the cycle directly (its prefix length and period), without needing general graph search, because determinism guarantees exactly one successor per state. The tick component t_{eff} is capped at D , since nothing in the transition rule depends on t beyond the boolean $t_{\text{eff}} \geq D$, which keeps the per-configuration state space finite.

We implemented this in Python and ran it as an exhaustive check (not a statistical sample) over 14 configurations, N in $\{3,5,7,9,11\}$ times C in $\{1,2,3\}$ excluding $C \geq N$ (trivial), for both jitter schemes and both overload semantics, 56 runs in total. This deterministic-simulation style of explicit-state analysis is the same general approach, applied to a smaller and fully deterministic state space, as the reachability and liveness checking built into general-purpose specification languages and model checkers such as TLA+ [21].

4.2 Independent SMT cross-check

Proposition 1 (round-one spread). Let $\text{jitter}_i = i \bmod m$ for $i = 0, \dots, N-1$. If $m \geq \lceil N/C \rceil$, then no residue class $\{i : i \bmod m = r\}$ contains more than C indices, so on their first retry round (before any breaker's level has changed) no more than C breakers ever fire on the same tick.

Proof. Assigning N items to m residue classes by $i \bmod m$ distributes them as evenly as possible: every class has either $\lfloor N/m \rfloor$ or $\lceil N/m \rceil$ members. With $m = \lceil N/C \rceil$, $\lceil N/m \rceil \leq C$, so the largest class has at most C members.

This is elementary and holds for every N , but because it is the crux of the paper's safety claim we cross-checked it independently with the Z3 SMT solver [22] rather than trusting only the Python arithmetic that also drives the simulator: for each of 23 (N, C) pairs with N from 3 to 50, we built the concrete residue-count formula and asked Z3 to find a residue class exceeding C . All 23 queries returned unsat, i.e. Z3, an independently implemented solver, agrees no violation exists.

5 Results

5.1 Fail-together semantics: livelock and its elimination

Table 1 summarizes the exhaustive explicit-state results under fail-together semantics: the naive scheme livelocks in 14 of 14 configurations, permanently, at an identical period of 9 ticks in every case, with all breakers stuck at the capped backoff level and identical timers. The modulo scheme livelocks in 0 of 14; 10 of 14 never exceed C at any tick, while 4 of 14 exhibit transient overload with peak concurrency exactly $C+1$ before ultimately converging (Section 7).

Table 1. Fail-together semantics, exhaustive explicit-state results ($m = \lceil N/C \rceil$).

N	C	scheme	outcome	recovery tick	max concurrent	exceeds C?
3	1	none	livelock	-	3	yes, permanent (+2)
3	1	modulo	recovered	12	1	no
3	2	none	livelock	-	3	yes, permanent (+1)
3	2	modulo	recovered	10	2	no
5	1	none	livelock	-	5	yes, permanent (+4)
5	1	modulo	recovered	14	1	no
5	2	none	livelock	-	5	yes, permanent (+3)
5	2	modulo	recovered	12	2	no
5	3	none	livelock	-	5	yes, permanent (+2)
5	3	modulo	recovered	10	3	no
7	1	none	livelock	-	7	yes, permanent (+6)
7	1	modulo	recovered	18	2	yes, transient (+1)
7	2	none	livelock	-	7	yes, permanent (+5)
7	2	modulo	recovered	14	2	no
7	3	none	livelock	-	7	yes, permanent (+4)
7	3	modulo	recovered	12	3	no
9	1	none	livelock	-	9	yes, permanent (+8)

N	C	scheme	outcome	recovery tick	max concurrent	exceeds C?
9	1	modulo	recovered	22	2	yes, transient (+1)
9	2	none	livelock	-	9	yes, permanent (+7)
9	2	modulo	recovered	14	2	no
9	3	none	livelock	-	9	yes, permanent (+6)
9	3	modulo	recovered	12	3	no
11	1	none	livelock	-	11	yes, permanent (+10)
11	1	modulo	recovered	26	2	yes, transient (+1)
11	2	none	livelock	-	11	yes, permanent (+9)
11	2	modulo	recovered	18	3	yes, transient (+1)
11	3	none	livelock	-	11	yes, permanent (+8)
11	3	modulo	recovered	14	3	no

Example witness ($N=5$, $C=2$, no jitter): all five breakers open at tick 0; the genuine outage masks the first probe at tick 3; backoff brings all five back, now synchronized, to fire together at ticks 8, 17, 26, ... forever at period 9, because with zero jitter every breaker's level and timer evolve identically. The system never reaches a state where fewer than 5 breakers are competing, so it never recovers. With modulo jitter ($m = \text{ceil}(N/C) = 3$), the same configuration recovers by tick 12, with concurrent probes never exceeding $C=2$ at any tick.

5.2 Admission-control semantics: always recovers, jitter still helps

Under admission-control semantics, all 28 configurations tested (14×2 jitter schemes) recover; there is no livelock, because each overloaded round admits exactly $\min(k, C)$ breakers, shrinking the retrying population every round. Table 2 gives the exact recovery tick predicted by the exhaustive model and the resulting speedup from jitter, ranging from 1.70x to 3.77x, growing with the N/C ratio because the unjittered fleet needs $\text{ceil}(N/C)$ full backoff rounds while the jittered fleet converges in a small, near-constant number of rounds regardless of N .

Table 2. Admission-control semantics: recovery ticks and jitter speedup, exhaustive results.

N	C	ticks (none)	ticks (modulo)	speedup
3	1	26	12	2.17x
3	2	17	10	1.70x
5	1	44	14	3.14x
5	2	26	12	2.17x
5	3	17	10	1.70x
7	1	62	18	3.44x
7	2	35	14	2.50x
7	3	26	12	2.17x
9	1	80	22	3.64x
9	2	44	14	3.14x
9	3	26	12	2.17x
11	1	98	26	3.77x
11	2	53	17	3.12x
11	3	35	14	2.50x

6 Prototype Validation

To check that the abstract discrete-tick predictions survive a genuinely concurrent implementation, we built an independent asyncio prototype: real Python coroutines, a real `asyncio.Lock`-protected shared counter standing in for the dependency, and real wall-clock timing (1 abstract tick = 50 ms), rather than a re-use of the abstract simulator's code.

For $N=5$, $C=2$ (no jitter), the prototype's admission pattern matched the abstract admission-control model exactly in structure: five breakers fire together once the real recovery window opens; the first two (by lock-acquisition order) are admitted and recover, the remaining three retry, fire together again, two more are admitted, and the last recovers alone, four rounds total, identical to the abstract model's prediction. The measured recovery tick (about 22.5) differs from the abstract model's

predicted tick 26 by a small, well-understood constant: the abstract model's check-then-decrement stepping convention adds an implicit rounding delay of roughly one tick per backoff round relative to a continuous-time wait, so the gap grows by about one tick per round, while the round structure, the admission pattern, and the qualitative outcome match exactly. Table 3 reports measured recovery ticks, maximum observed concurrency, and probe counts for both jitter schemes at two fleet sizes; jitter reduced measured recovery time from 22.46 to 10.24 ticks ($N=5, C=2$) and from 22.48 to 10.22 ticks ($N=9, C=3$), consistent with the abstract model's predicted speedup. Re-running this same prototype reproduces these figures to within about 0.1 tick, real wall-clock variance intrinsic to genuine concurrent execution; the admission counts, concurrency, and probe totals in Table 3 are exactly reproducible every run, since only the timing, not the algorithm's outcome, is affected by this variance.

Table 3. Asyncio prototype measurements (independent implementation, real concurrency and timing).

N	C	Scheme	Measured recovery (ticks)	Max concurrency	Failed probes	Total probes
5	2	none	22.46	5	9	14
5	2	modulo	10.24	2	5	10
9	3	none	22.48	9	18	27
9	3	modulo	10.22	3	9	18

7 Discussion, Limitations, and Threats to Validity

The modulo jitter scheme's round-one guarantee (Proposition 1) does not extend unconditionally to later rounds. In 4 of the 14 fail-together configurations ($N=7, C=1$; $N=9, C=1$; $N=11, C=1$; $N=11, C=2$), a breaker already on its second attempt, having failed once during the genuine outage window, can land on the same tick as another breaker still on its staggered first attempt, because the backoff duration change does not preserve the relative spacing that the fixed offsets established for round one. This can produce one or more transient ticks at $C+1$ concurrent probes rather than a permanent livelock; the fleet still converges. We traced the exact mechanism on $N=7, C=1$ and confirmed it is not an artifact of the simulator by manual step-by-step verification of the transition sequence.

We tested one natural attempted fix: a rotating offset, $\text{jitter}_i(\text{level}) = (i + \text{level}) \bmod m$, intended to re-decorrelate breakers across rounds rather than only at round one. Re-running the full sweep with this scheme did not eliminate the transient overload, and in three configurations ($N=3, C=2$; $N=5, C=3$; $N=11, C=3$) it produced a larger transient excess than the static scheme. We report this as a negative result rather than omit it: the obvious fix does not work, and finding a jitter scheme that also guarantees the C bound on every later round, not just round one, is left open.

Threats to validity. The formal model is a discrete abstraction with a synchronized worst-case initial condition; a fleet with staggered failure detection would see a milder version of the same effect. The two overload semantics are stylized; a real dependency may exhibit a mixture, or a smoother capacity-degradation curve, rather than a sharp threshold. The prototype uses a single-process asyncio event loop rather than genuinely distributed processes, so it cannot capture the cross-machine clock skew and network delay variance that are intrinsic to real distributed deployments [23]; both would, if anything, further help decorrelate anunjittered real deployment relative to a perfectly synchronized abstract worst case, meaning our livelock result should be read as an achievable adversarial worst case rather than a guaranteed real-world outcome absent jitter.

8 Conclusion and Future Work

Whether jitter in a circuit-breaker fleet's backoff is a mere optimization or a correctness requirement depends on an assumption about downstream overload behavior that implementers rarely state explicitly. We verified, exhaustively for each tested configuration, that it is a correctness requirement under fail-together overload semantics, where its absence causes a permanent, self-inflicted livelock, and only an efficiency concern under admission-control semantics, where recovery always happens

but jitter measurably accelerates it. A deterministic modulo jitter assignment provably eliminates the livelock on the first retry round, cross-checked independently in Z3, while retaining a residual transient-overload limitation on later rounds that a natural rotating-offset fix did not resolve. Future work includes characterizing exactly which (N, C, m) triples avoid the residual transient overload, extending the model to mixed or gradually-degrading overload semantics, re-running the explicit-state check with a general-purpose explicit-state model checker such as SPIN [24] as a further independent cross-validation, and validating the prototype's findings against a genuinely multi-process or multi-host deployment using chaos engineering practices [25].

References

- [1] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2018.
- [2] M. Fowler, “CircuitBreaker,” [martinfowler.com](https://martinfowler.com/bliki/CircuitBreaker.html), 2014. [Online]. Available: <https://martinfowler.com/bliki/CircuitBreaker.html>.
- [3] Netflix, “Hystrix: Latency and Fault Tolerance Library.” [Online]. Available: <https://github.com/Netflix/Hystrix>.
- [4] resilience4j, “resilience4j: fault tolerance library designed for Java8 and functional programming.” [Online]. Available: <https://github.com/resilience4j/resilience4j>.
- [5] App-vNext, “Polly: .NET resilience and transient-fault-handling library.” [Online]. Available: <https://github.com/App-vNext/Polly>.
- [6] Envoy Proxy, “Outlier detection,” Envoy documentation. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/outlier.
- [7] Microsoft, “Circuit Breaker pattern,” Azure Architecture Center, Microsoft Learn. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>.
- [8] Kubernetes, “Liveness, Readiness, and Startup Probes,” Kubernetes Documentation. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/probes/>.
- [9] M. Brooker, “Exponential Backoff And Jitter,” AWS Architecture Blog, Mar. 4, 2015. [Online]. Available: <https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/>
- [10] M. Brooker, “Timeouts, retries, and backoff with jitter,” Amazon Builders' Library, Dec. 2019. [Online]. Available: <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>
- [11] M. Ulrich, “Addressing Cascading Failures,” in *Site Reliability Engineering: How Google Runs Production Systems*, B. Beyer, C. Jones, J. Petoff, N. R. Murphy, Eds. Sebastopol, CA: O'Reilly Media, 2016, ch. 22.
- [12] J. Dean, L. A. Barroso, “The Tail at Scale,” *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.
- [13] N. Bronson, A. Aghayev, A. Charapko, T. Zhu, “Metastable Failures in Distributed Systems,” in *Proc. Workshop on Hot Topics in Operating Systems (HotOS '21)*, Ann Arbor, MI, USA, ACM, 2021, pp. 221–227.
- [14] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, R. Yao, “Gray Failure: The Achilles' Heel of Cloud-Scale Systems,” in *Proc. 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, ACM, 2017.
- [15] M. Ileana, “A Dynamic Load Balancing Algorithm for Distributed Web Systems,” *Proceedings of the 62nd Annual Scientific Conference of Angel Kanchev University of Ruse*, vol. 62, book 3.2, 2023.

- [16] M. Ileana, P. Petrov, V. Milev, “Intelligent Intrusion Detection in IoT with Explainable AI and Distributed Web Systems,” in 2025 International Conference on Cybersecurity and AI-Based Systems (Cyber-AI), IEEE, 2025, pp. 124-127, doi: 10.1109/Cyber-AI66431.2025.11233693.
- [17] M. Ileana, P. Petrov, V. Milev, “Optimizing CRM Platforms with Distributed Cloud Architectures for Scalable Performance and Security,” in 2024 8th International Symposium on Innovative Approaches in Smart Technologies (ISAS), IEEE, 2024, pp. 1-6, doi: 10.1109/ISAS64331.2024.10845520.
- [18] D. A. Popescu, M. Ileana, N. Bold, “Analyzing the Performance of Distributed Web Systems Within an Educational Assessment Framework,” in Breaking Barriers with Generative Intelligence. Using GI to Improve Human Education and Well-Being, A. Basiouni, C. Frasson, Eds., Communications in Computer and Information Science, vol. 2162. Cham: Springer Nature Switzerland, 2024, pp. 102-115, doi: 10.1007/978-3-031-65996-6_9.
- [19] R. Alur, D. L. Dill, “A Theory of Timed Automata,” Theoretical Computer Science, vol. 126, no. 2, pp. 183-235, 1994.
- [20] G. Behrmann, A. David, K. G. Larsen, “A Tutorial on Uppaal,” in Formal Methods for the Design of Real-Time Systems (SFM-RT 2004), LNCS vol. 3185, Springer, 2004, pp. 200-236.
- [21] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Boston: Addison-Wesley, 2002.
- [22] L. de Moura, N. Bjørner, “Z3: An Efficient SMT Solver,” in Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), LNCS vol. 4963, Springer, 2008, pp. 337-340.
- [23] M. van Steen, A. S. Tanenbaum, Distributed Systems, 3rd ed., distributed-systems.net, 2017.
- [24] G. J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
- [25] C. Rosenthal, N. Jones, Chaos Engineering: System Resiliency in Practice. Sebastopol, CA: O'Reilly Media, 2020.

Copyright © 2026 Marian Ileana, Vassil Milev, Pavel Petrov. This is an open-access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.