10.51231/2667-9507-2026-005-12-57-77

# Self-Inflicted Denial-of-Service in Digital Healthcare Systems: A Simulation Study of Retry Storms, Metastable Failure, and Mitigation Policies

Teimuraz Sturua[1,2]

[1] *Georgian Technical University, Tbilisi, Georgia*
[2] *European University, Tbilisi, Georgia*

## Abstract

Modern healthcare increasingly depends on distributed digital services, including electronic health record systems, telemedicine platforms, patient portals, laboratory information systems, and clinical decision-support applications. Temporary overload or degradation of such services may trigger automatic client retries, potentially increasing system load and converting a short disturbance into a prolonged service disruption. This phenomenon represents a form of self-inflicted denial-of-service and may threaten the availability and continuity of digital healthcare services. This study evaluates the behavior of retry and load-shedding policies under transient overload and identifies conditions associated with metastable failure in distributed healthcare-oriented service architectures. A discrete-event simulation model of an open-loop client fleet and a multi-worker server was developed. The model incorporated client timeouts, retries, bounded server queues, wasted processing of abandoned requests, load spikes, and temporary capacity reductions. Eight policies were compared: no retry, immediate retry, exponential backoff, exponential backoff with jitter, retry budget, circuit breaker, server-side deadline checking, and a combination of retry budget with server-side deadline check-

ing. A total of 2,220 simulation runs were performed, varying trigger magnitude and duration, baseline utilization, server capacity, queue capacity, and retry-budget size. Under the reference scenario of a 2-fold load spike lasting 10 s at 70% baseline utilization, immediate retry, exponential backoff, and backoff with jitter resulted in persistent metastable failure in all 10 simulation runs, with request success of only 2.6-2.7% and retry amplification of approximately 3.9. In contrast, retry-budget and circuit-breaker policies prevented metastable failure under the tested conditions. Server-side deadline checking alone reduced wasted processing but resulted in a degraded state with approximately 22% request success. Combining retry-budget control with server-side deadline checking produced 93.9% request success and the lowest peak queue among the tested policies. A trigger-severity index based on accumulated backlog relative to the amount of work that can be completed during one client timeout period corresponded closely with the simulated transition to instability. In distributed digital healthcare services, retry timing mechanisms such as exponential backoff and jitter should not be considered sufficient protection against self-inflicted overload. Limiting the rate of retries, controlling queueing delay, and discarding requests that have exceeded their useful deadline can substantially improve service resilience. The findings provide a simulation-based framework for assessing the reliability of digital healthcare infrastructure and for designing safer fault-injection and resilience-testing strategies.

**KEYWORDS: digital healthcare; healthcare information systems; telemedicine; electronic health records; self-inflicted denial of service; retry storm; metastable failure; distributed systems resilience; retry budget; circuit breaker; simulation; healthcare IT infrastructure**

## Introduction

Digital technologies have become an essential component of modern healthcare delivery. Electronic health record (EHR) platforms, telemedicine services, patient portals, laboratory and diagnostic information systems, appointment platforms, and clinical decision-support applications increasingly rely on distributed software architectures.

The availability of these systems can therefore influence the continuity and timeliness of access to healthcare information and digital services.

The reliability of distributed healthcare systems is commonly discussed in terms of external cyberattacks, hardware failures, network disruptions, or software defects. However, a service may also become unavailable as a consequence of its own normal reliability mechanisms. A relatively short disturbance-such as a temporary increase in demand, degradation of a dependent service, infrastructure maintenance, garbage-collection pause, or partial system outage-may increase response latency beyond the timeout configured by clients. Automated clients may then retransmit failed requests. The additional requests further increase system load, causing additional delays and generating still more retries.

This positive feedback loop (Fig. 1) can result in a situation in which the original disturbance has already ended but the service remains overloaded. Importantly, the server may continue processing requests for which the requesting client has already stopped waiting. Processing capacity is consequently consumed by work that no longer produces useful output. In healthcare information systems, this type of failure may reduce the availability of patient-facing services or delay access to digital clinical information even though the initial technical disturbance was relatively short.

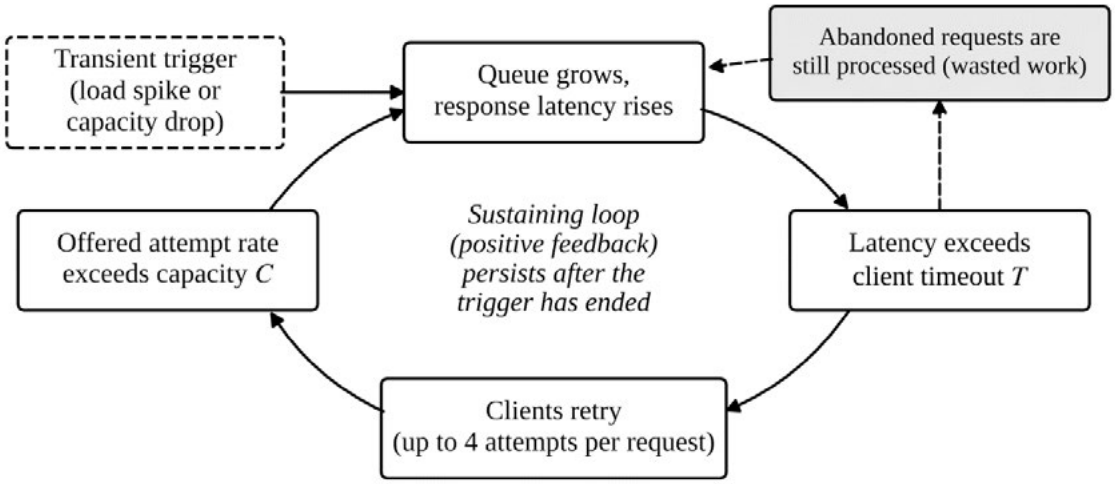



*Fig. 1. Retry-storm feedback loop underlying self-inflicted denial-of-service. A transient trigger (dashed box) increases queueing delay; once latency exceeds the client timeout, retries raise the offered attempt rate above server capacity, which sustains the overload after the trigger has ended. Processing of abandoned requests (grey box) consumes further capacity.*

Bronson et al. described this behavior as a ***metastable failure***: a system may remain in an undesirable state after the initial trigger has disappeared because a sustaining mechanism continues to generate load [1]. Huang et al. investigated 22 such incidents

across 11 organizations and identified retry policies as a common sustaining mechanism [2]. From the perspective of service availability, this phenomenon resembles a distributed denial-of-service event, although the additional traffic originates from legitimate clients of the service. In this study, the phenomenon is referred to as **self-inflicted denial-of-service (self-DDoS)** [1].

Several engineering mechanisms have been proposed to reduce retry-related overload, including exponential backoff, jitter, retry budgets, circuit breakers, deadline propagation, and server-side load shedding [3-7]. These mechanisms are frequently recommended for resilient distributed systems. However, their individual contribution under comparable operating conditions is less frequently quantified.

This issue is particularly relevant to digital healthcare because healthcare services may involve large numbers of simultaneously connected users and automated systems. For example, a temporary increase in requests to a patient portal or telemedicine platform may cause repeated client requests, while an overloaded backend may continue processing requests that have already exceeded the useful response deadline.

The objective of this study was therefore to investigate, using a reproducible discrete-event simulation, how different retry and load-shedding policies affect the stability of distributed services that may support digital healthcare applications.

The study had four main objectives:

1. To develop a reproducible simulation model of a distributed service with client timeouts, retries, bounded queues, and multiple server workers.
2. To compare eight, retry and load-shedding policies under systematic variations of transient overload and service capacity.
3. To determine whether exponential backoff and jitter are sufficient to prevent metastable failure in an open-loop client population.
4. To identify simple indicators that can help estimate the vulnerability of a digital service to self-inflicted overload.

## Background and Related Work

### *1. Metastable failures in distributed services*

Metastable failure describes a state in which a system that was initially functioning normally remains degraded after the original trigger has disappeared [1]. The distinction between stable, vulnerable, and metastable states is particularly useful for distributed healthcare systems because a temporary infrastructure disturbance may become prolonged when interacting with automatic recovery mechanisms (Fig. 2).

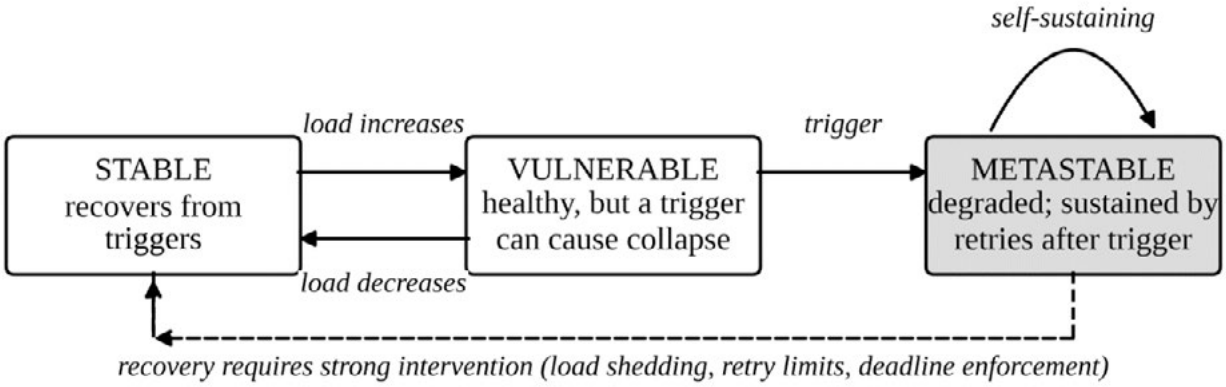



*Fig. 2. Stable, vulnerable and metastable states of a distributed service, following the state model of Bronson et al. Movement between the stable and vulnerable states depends on load; a trigger can move a vulnerable system into a self-sustaining metastable state, from which recovery requires explicit intervention.*

Huang et al. analysed real-world metastable failures and identified retry behaviour as one of the most frequent sustaining mechanisms [2]. Other research has developed probabilistic, queueing, and formal models for identifying conditions under which distributed systems become vulnerable [8-11].

These findings suggest that resilience cannot be evaluated only by asking whether a system survives an initial fault. It is also necessary to determine whether the system can recover after the fault has been removed.

## *2. Retry mechanisms and overload control*

Exponential backoff with jitter is a widely recommended mechanism for reducing synchronized retries [3]. Backoff increases the interval between successive attempts, while jitter introduces randomization. These mechanisms can reduce simultaneous request bursts.

However, backoff does not necessarily reduce the total number of attempts generated by a large population of independent clients. Brooker has argued that backoff can defer rather than eliminate work in such environments [5]. Retry token buckets and circuit breakers provide an additional mechanism by limiting the amount of retry traffic reaching the service [4].

The Google Site Reliability Engineering literature recommends retry budgets, deadline propagation, and load shedding as mechanisms for reducing cascading failures [6]. The circuit-breaker pattern provides another approach by temporarily preventing requests from reaching a failing service [7].

### *3. Queueing and response deadlines*

Large queues can transform overload into excessive latency [12]. A request may remain technically processable while becoming clinically or operationally useless because its response arrives after the client has timed out. This phenomenon is related to the concept of buffer bloat in computer networks, where excessive buffering produces large delays [13]. Queue-delay control mechanisms such as CoDel demonstrate that limiting delay can be more important than simply maximizing buffer utilization [14]. The same principle may apply to healthcare information systems: a request that remains in a queue beyond the useful response deadline may consume computing resources without contributing to effective service delivery.

### *4. Relevance to digital healthcare*

The increasing dependence of healthcare organizations on distributed digital infrastructure makes availability an important component of healthcare service continuity. A disruption in an EHR interface, telemedicine platform, appointment system, laboratory information system, or patient portal does not necessarily represent a clinical failure in itself, but it can interfere with access to information and digital services.

Consequently, resilience testing of healthcare IT infrastructure should consider not only external attacks but also failure amplification caused by legitimate automated clients.

Chaos engineering provides a structured approach for testing system resilience using controlled faults [15]. However, deliberately triggering a metastable failure in a production healthcare environment may be inappropriate because of the potential impact on users and services. Simulation can therefore be used as a preliminary method for identifying potentially unsafe parameter regions.

## System Model and Mitigation Policies

### *1. Simulation model*

The simulated system consists of three components: users, a client fleet, and a server (Fig. 3).

Users generate requests according to a Poisson process with rate λ. The client fleet sends each request to the server as an attempt. Each attempt has a client timeout T and may be retried up to three times, resulting in a maximum of four attempts per original request.

The server contains four parallel workers. Service times are exponentially distributed with a mean of 20 ms. The resulting nominal server capacity is 200 requests/s. Requests enter a FIFO queue with a finite capacity of 5,000 attempts.

The reference baseline utilization is 70%, corresponding to an incoming rate of 140 requests/s.

An attempt fails when its timeout expires or when the server queue is full. Unless server-side deadline checking is enabled, the server continues processing an abandoned attempt even after the client has stopped waiting. The response is subsequently discarded. This mechanism represents wasted processing capacity.

The model uses an open-loop client population: new users continue to generate requests at the specified rate independently of the current system state.

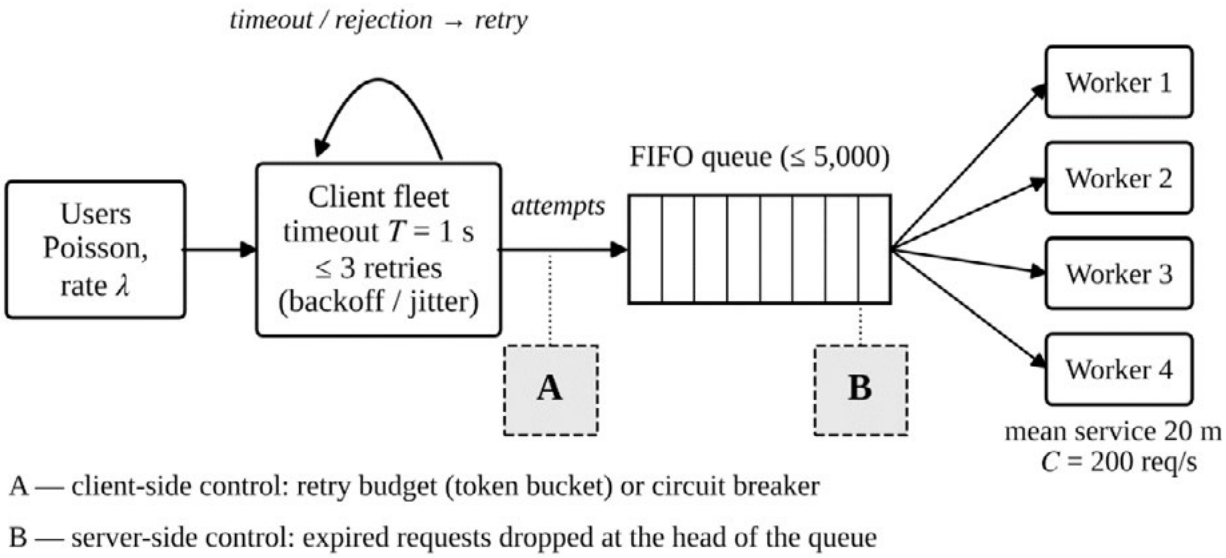



*Fig. 3. Simulation model. Users generate requests (Poisson process, rate λ); the client fleet sends attempts with timeout T and up to three retries to a bounded FIFO queue served by four parallel workers (nominal capacity C = 200 requests/s). Dashed boxes mark where the mitigation policies act: (A) client-side retry budget or circuit breaker; (B) server-side deadline check at the head of the queue.*

Two types of transient trigger were investigated:

- **Load spike:** the incoming request rate is multiplied by a specified factor.
- **Capacity drop:** the mean service time is multiplied by a specified factor, thereby reducing server capacity.

These triggers represent simplified versions of situations that may occur in digital healthcare infrastructure, such as sudden increases in portal or telemedicine traffic, temporary degradation of a dependent service, or partial loss of processing capacity.

## *2. Mitigation policies*

Eight policies were compared (Table 1):

*TABLE 1. RETRY AND LOAD-SHEDDING POLICIES COMPARED IN THE SIMULATION*

| Policy | Description |
|---|---|
| No retry | Baseline; failed requests are not retried |
| Immediate retry | Failed requests are retransmitted immediately |
| Exponential backoff | Retry delay increases exponentially |
| | |
| Backoff + jitter | Exponential backoff combined with randomised retry delay |
| Jitter + retry budget | Backoff with jitter plus a fleet-wide retry token budget |
| Jitter + circuit breaker | Backoff with jitter plus a client-side circuit breaker |
| Immediate + server deadline | Immediate retry with server-side rejection of expired requests |
| Budget + server deadline | Retry budget combined with server-side deadline checking |

All retrying policies use a maximum of three retries.

The retry budget uses a fleet-wide token-bucket mechanism. Each new request contributes tokens, while each retry consumes one token. This limits the proportion of traffic that can be generated through retries.

The circuit breaker opens for 2 s when more than 50% of attempts in a 1-s observation window fail, with a minimum of 20 observations. While open, new requests and retries fail immediately without reaching the server.

The server-side deadline mechanism discards a request when its deadline has expired by the time it reaches the head of the queue.

### *3. Analytical indicators*

Three quantities were used to characterize system vulnerability.

## Vulnerability condition

If each failed request generates four attempts, the offered attempt rate may substantially exceed the original request rate. When the resulting attempt rate exceeds server capacity, the system cannot drain its backlog.

For a retry-budget ratio (r), the maximum retry contribution can be approximated as (r\lambda). The resulting stability condition can therefore be expressed as a relationship between baseline utilization and the permitted retry ratio.

For the reference utilization of 0.70, the theoretical retry-budget boundary is approximately:

$$r^* = \frac{1}{p_0} = 1 + 0.43 = 1.43$$

### Trigger-severity index

We define the trigger-severity index as the backlog accumulated during the trigger divided by the amount of work that the server can complete during one client timeout:

$$S = \frac{B_{trigger}}{C \cdot T}$$

where

$$B_{trigger}$$

is the additional backlog produced by the trigger, (C) is server capacity, and (T) is the client timeout.

A value of approximately 1 represents a situation in which the accumulated backlog corresponds to roughly one timeout period of server work.

## Queue-capacity threshold

The same reasoning suggests a queue-capacity threshold:

$$Q^* \approx CT$$

If the queue is maintained at or below this level, an attempt should generally not remain queued for longer than approximately one client timeout under the simplified model.

These indicators are first-order approximations and are not intended as exact clinical or operational thresholds.

## Experimental Methodology

#### *1. Simulator*

The simulator was implemented in Python 3.13 using a discrete-event architecture and NumPy for aggregation.

Each simulation run lasted 240 s (Fig. 4) and included:

- 60 s of baseline operation;
- the transient trigger;
- at least 150 s of post-trigger observation.

The simulator is deterministic for a given random seed.

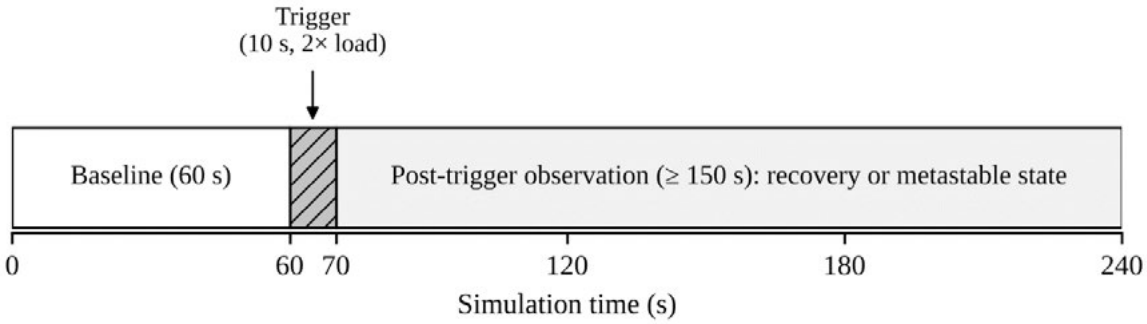



*Fig. 4. Timeline of a single simulation run, shown for the reference trigger (2-fold load spike lasting 10 s). Recovery time is measured from the end of the trigger.*

The principal reference parameters are summarised in Table 2:

***TABLE 2. PRINCIPAL PARAMETERS OF THE REFERENCE CONFIGURATION***

| Parameter | Reference value |
|---|---|
| Number of workers | 4 |
| Mean service time | 20 ms |
| Server capacity | 200 requests/s |
| Baseline utilisation | 0.70 |
| Client timeout | 1 s |
| Maximum retries | 3 |
| Backoff base/cap | 0.1 / 5 s |
| Retry-budget ratio | 0.1 |
| Token-bucket size | 50 |
| Circuit breaker | 1 s / 50% / 2 s |
| Queue capacity | 5,000 attempts |
| Trigger | 2× load spike |
| Trigger duration | 10 s |
| Simulation duration | 240 s |

### 2. Experimental design

Each configuration was executed with 10 independent random seeds.
Six experiments produced a total of 2,220 simulation runs (Table 3):

*TABLE 3. EXPERIMENTAL DESIGN: VARIED PARAMETERS, POLICIES AND NUMBER OF RUNS*

| Experiment | Variable | Values | Policies | Runs |
|---|---|---|---|---|
| E1 | Reference scenario | Reference configuration | 8 | 80 |
| E2 | Trigger magnitude/duration | 1.25–3×; 2–20 s | 8 | 1,280 |
| E3 | Baseline utilisation | 0.5–0.9 | 8 | 400 |
| E4 | Capacity reduction | 1.5–3× service time | 8 | 240 |
| E5 | Queue capacity | 50–5,000 | 4 | 160 |
| E6 | Retry-budget ratio | 0.05–1.0 | Budget | 60 |

### 3. Outcome measures

The following metrics were calculated:

- **Goodput:** requests successfully completed within the client timeout per second.
- **Recovery time:** time from the end of the trigger until goodput remained at or above the predefined recovery level for 10 consecutive seconds.
- **Metastable run:** a run that had not recovered by the end of the simulation.
- **Request success:** proportion of requests that ultimately succeeded after retries.
- **Retry amplification:** total attempts divided by original requests.
- **Wasted work:** proportion of completed service associated with responses arriving after client timeout.
- **p99 latency:** 99th percentile of end-to-end latency for successful requests.
- **Peak queue:** maximum queue length.

Means are reported with 95% confidence intervals calculated across the independent simulation seeds.

# Results

### *1. Reference scenario*

Under the reference trigger-a 2-fold load increase for 10 s at 70% baseline utilization-policies that allowed unrestricted retries entered persistent metastable failure (Fig. 5, Table 4).

*TABLE 4. REFERENCE-SCENARIO OUTCOMES BY POLICY (2-FOLD LOAD SPIKE, 10 S, $P_0$ = 0.70; MEANS OF 10 RUNS)*

| Policy | Metastable runs | Recovery, median | Request success | Retry amplification | Wasted work | p99 latency | Peak queue |
|---|---|---|---|---|---|---|---|
| No retry | 0/10 | 14 s | 86.9%± 0.9 | 1.00 | 13.1% | 0.80 s | 813 |
| Immediate retry | 10/10 | — | 2.7% ± 0.3 | 3.91 | 98.0% | 0.98 s | 5,000 |
| Exponential backoff | 10/10 | — | 2.7% ± 0.3 | 3.90 | 98.0% | 0.98 s | 5,000 |
| Backoff + jitter | 10/10 | — | 2.6% ± 0.3 | 3.91 | 98.0% | 0.98 s | 5,000 |
| Jitter + retry budget | 0/10 | 24 s | 82.2% ± 1.6 | 1.02 | 19.4% | 0.84 s | 1,089 |
| Jitter + circuit breaker | 0/10 | 14 s | 93.8% ± 0.6 | 0.97 | 3.7% | 0.83 s | 428 |
| Immediate + server deadline | 10/10 | — | 22.2% ± 0.8 | 3.57 | 83.6% | 4.00 s | 1,149 |
| Budget + server deadline | 0/10 | 16.5 s | 93.9% ± 0.3 | 1.01 | 3.9% | 0.99 s | 371 |

Before the trigger, all policies produced similar behavior, with a p99 latency of approximately 0.11 s.

During and after the load spike, immediate retry, exponential backoff, and backoff with jitter generated approximately 560 attempts/s. This was approximately four times the original user request rate and substantially exceeded server capacity.

The server therefore remained continuously busy, while approximately 98% of completed processing represented work whose response arrived after the corresponding client timeout.

The similarity between exponential backoff and jitter indicates that these mechanisms altered the timing of retries but did not sufficiently reduce the total amount of retry-generated work in the simulated open-loop environment.

Server-side deadline checking substantially reduced the queue size but did not prevent the system from remaining in a degraded state. Request success remained approximately 22%, with p99 latency increasing to approximately 4 s.

Both retry-rate limiting approaches prevented metastable collapse. The retry budget maintained amplification near 1.02, while the circuit breaker reduced peak queue length and wasted work. Combining the retry budget with server-side deadline checking resulted in the highest request success (93.9%) and the lowest peak queue (371) in the reference experiment.

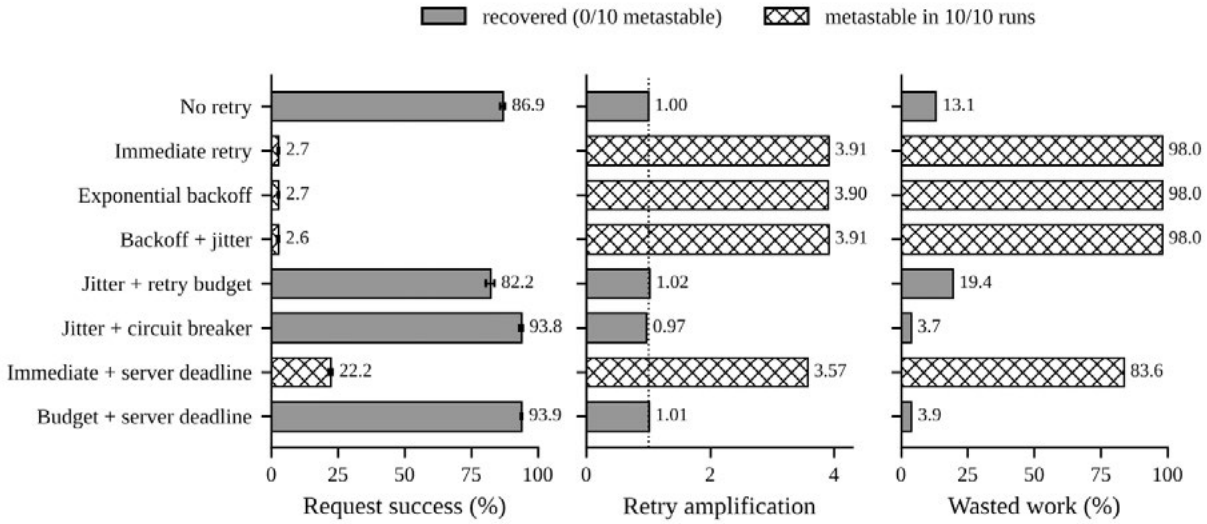



*Fig. 5. Reference-scenario outcomes by policy (2-fold load spike, 10 s, $\rho_0 = 0.70$; means of 10 runs). Error bars in the left panel show 95% confidence intervals. Hatched bars denote policies that remained metastable in all 10 runs; the dotted line in the middle panel marks an amplification of 1. Values correspond to the reference-scenario Table 4.*

### *2. Trigger magnitude and duration*

The probability of metastable failure increased sharply once the trigger generated a sufficiently large backlog.

At a trigger magnitude of 1.25, the increased load remained below server capacity and no runs collapsed.

For example, a 1.5-fold increase lasting 10 s generated a trigger-severity index close to the transition region. A 1.5-fold increase lasting 20 s generated an index of approximately 1 and resulted in metastable behavior in 6-7 of 10 runs.

For more severe triggers, all runs using immediate retry, exponential backoff, backoff with jitter, or server-deadline checking collapsed.

In contrast, the retry-budget, circuit-breaker, budget-plus-deadline, and no-retry

policies did not produce metastable runs in the tested trigger grid.

These results support the use of backlog relative to timeout-period capacity as a simple preliminary indicator of vulnerability (Fig. 6).

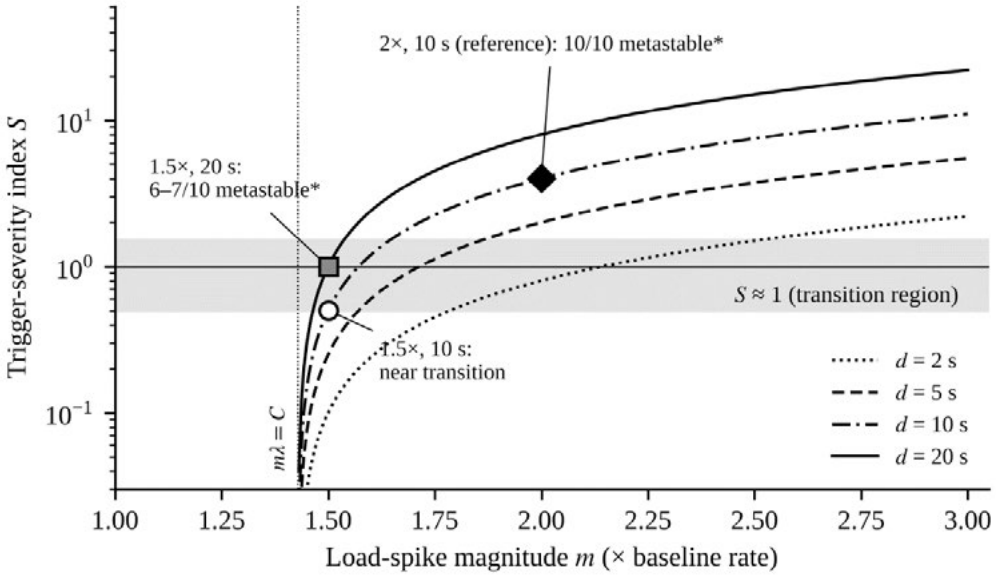



*Fig. 6. Trigger-severity index S = B_trigger / (C·T) for load spikes as a function of spike magnitude m and duration d (λ = 140 requests/s, C = 200 requests/s, T = 1 s), computed analytically from the model parameters. S = 0 when mλ ≤ C (dotted vertical line). Markers show configurations discussed in Section 5.2; * refers to policies without retry-rate limiting.*

### 3. Baseline utilization

Higher baseline utilization reduced the amount of spare capacity available for recovery (Fig. 7).

At 50% baseline utilization, the 2-fold load spike approximately reached server capacity, and all policies recovered with 100% request success.

At 60% utilization and above, the unlimited-retry policies consistently entered metastable failure.

The server-deadline policy remained functional at 60% utilization but collapsed from 70% upwards.

The retry budget remained stable up to approximately 80% utilization but collapsed at 90%. This corresponded to the theoretical stability boundary for the selected retry-budget ratio.

The circuit breaker and budget-plus-deadline combination maintained request success above 90% across the tested utilization range, including the 90% baseline-utilization condition.

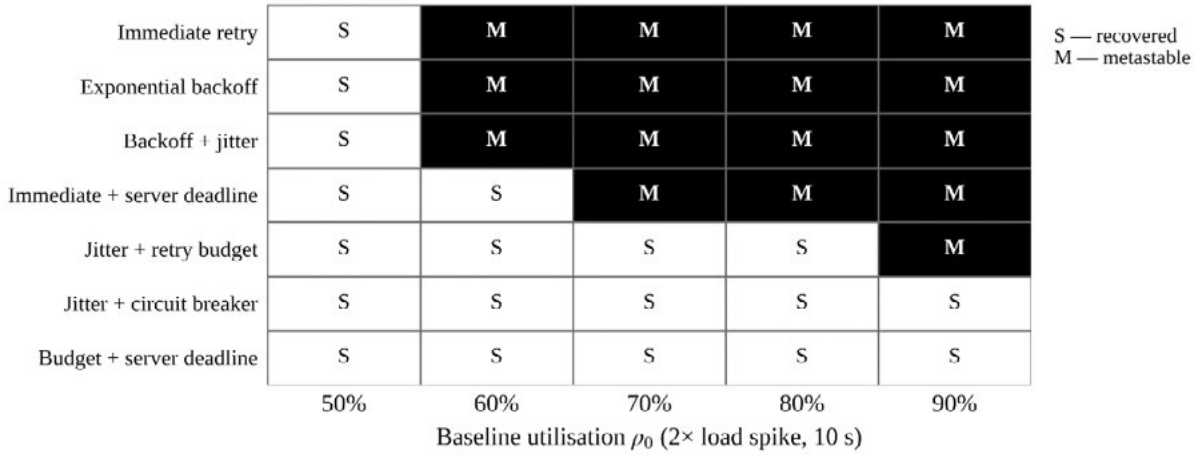



*Fig. 7. Outcome of a 2-fold, 10-s load spike across baseline utilisation levels for the seven retrying policies (Experiment E3). S – recovered; M – metastable.*

### *4. Capacity-drop trigger*

Temporary reductions in processing capacity produced a pattern similar to that observed with load spikes.

A 1.5-fold increase in service time reduced capacity to approximately 133 requests/s against a baseline workload of 140 requests/s. Under this condition, one of 10 runs collapsed for each of the immediate-retry, backoff, jitter, and deadline policies.

More severe capacity reductions resulted in collapse in all runs for these policies.

The retry budget, circuit breaker, and combined budget-plus-deadline approach prevented metastable failure in the tested configurations.

The combined approach achieved request-success rates of 99.9%, 97.1%, and 95.8% for the three capacity-reduction magnitudes.

### *5. Queue capacity*

Reducing queue capacity substantially improved resilience.

With queue capacities of 50 or 200 attempts, immediate retry and backoff with jitter recovered in every tested run, achieving approximately 94-97% request success.

With queue capacities of 1,000 and 5,000, both policies collapsed in every run.

The observed transition occurred around the analytically estimated threshold of approximately 200 attempts, corresponding to the amount of work that the server can complete during one 1-s client timeout.

Short queues also improved the no-retry baseline. At a queue capacity of 50, request success increased to 97.1%, compared with 86.9% at a queue capacity of 5,000. p99 latency decreased from 0.80 s to approximately 0.31 s.

These results indicate that excessive queueing may convert transient overload into prolonged delay and wasted processing.

### *6. Retry-budget size*

The retry budget behaved consistently with the theoretical stability boundary (Fig. 8).

At low retry-budget ratios, all runs recovered. At ratios of 0.5 and 1.0, all runs collapsed, with retry amplification increasing to 1.48 and 1.96, respectively.

For the reference scenario, the estimated stability boundary was:

$$r^* \approx 0.43$$

Within the stable region, smaller retry budgets produced higher request success and shorter recovery times. This indicates that a retry budget should primarily be interpreted as a **safety mechanism** rather than a performance optimization.

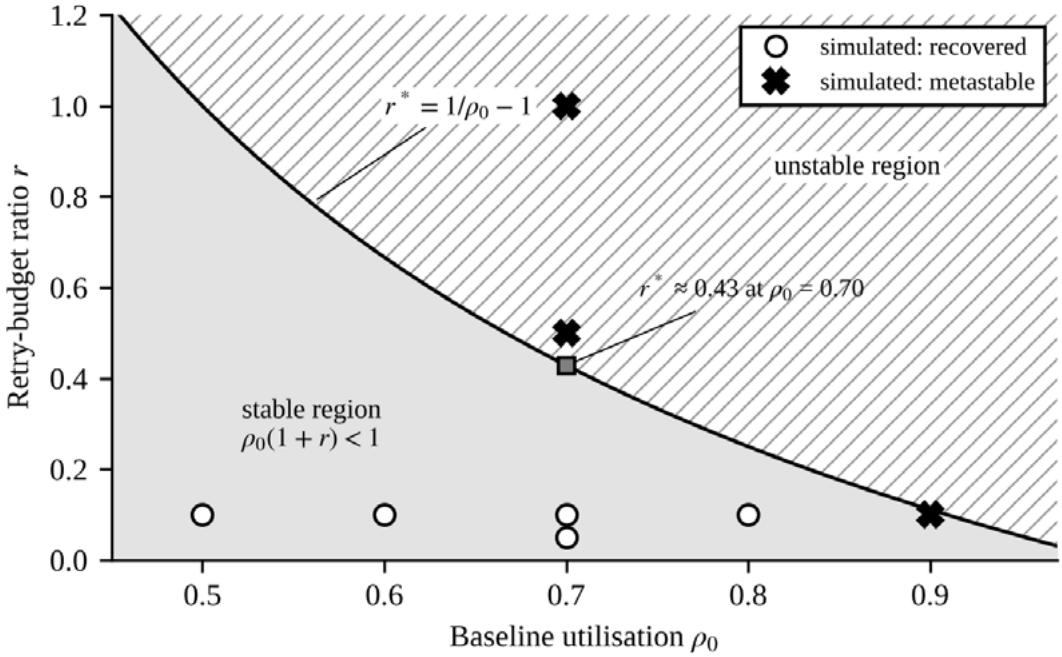



*Fig. 8. Theoretical retry-budget stability boundary $r^* = 1/\rho_0 - 1$ with simulated outcomes from Experiments E3 and E6 overlaid. The run at $\rho_0 = 0.9$, $r = 0.1$ lies close to the first-order boundary ($r^* \approx 0.11$) and collapsed in simulation.*

## Discussion

### *1. Relevance to digital healthcare*

The results demonstrate that automated recovery mechanisms can unintentionally amplify a transient disturbance in a distributed service. This finding has direct relevance to healthcare information infrastructure. Modern healthcare organizations often operate multiple interconnected digital services, and temporary degradation of one component may generate repeated requests from dependent systems, user interfaces, mobile applications, or automated integrations. For example, a temporary slowdown

in a patient portal backend could result in repeated browser or application requests. Similarly, a telemedicine platform experiencing transient latency may generate repeated connection or API attempts. An EHR interface or laboratory information system may also depend on multiple backend services, increasing the potential for cascading retry behavior [2,6]. The present study does not demonstrate that such failures cause specific clinical outcomes. Rather, it identifies a systems-engineering mechanism that may threaten the availability of digital healthcare services.

### *2. Exponential backoff is not sufficient by itself*

In the simulated open-loop environment, exponential backoff and full jitter did not prevent metastable failure. This does not imply that backoff and jitter are inappropriate. They remain useful for reducing synchronisation and instantaneous retry bursts [3]. However, the findings indicate that they should not be considered a complete defence against retry storms. For healthcare systems with strict availability requirements, backoff should therefore be combined with mechanisms that place an explicit upper bound on retry traffic.

### *3. Retry-rate limiting*

Retry budgets and circuit breakers were the most effective client-side mechanisms in the simulation.

A retry budget is particularly attractive because its operating principle is simple: only a controlled fraction of normal traffic may be converted into retries [4,6].

The appropriate budget should be derived from the peak expected utilization rather than the average utilization. A budget that is safe under moderate load may become unsafe when the service operates close to capacity. For healthcare organizations, this principle is relevant when configuring applications and integrations that communicate with shared backend services. Retry policies should be coordinated rather than independently configured by multiple client applications.

### *4. Queueing delay and deadline propagation*

The results also demonstrate the importance of preventing servers from processing requests that have already exceeded their useful deadline [6]. In a healthcare information system, the value of a response is time-dependent. A delayed response may be technically correct while no longer being useful to the requesting application or user. Server-side deadline checking reduces the amount of capacity spent on stale requests. However, the simulation indicates that deadline checking alone may not be sufficient because the system can continue generating excessive retries. The combination of retry-rate limiting and deadline checking therefore provides two complementary protections:

1. **client-side control** limits the amount of additional work entering the system;
2. **server-side control** prevents expired work from consuming processing capacity.

### *5. Queue configuration as a resilience parameter*

Large queues may appear desirable because they prevent immediate rejection of requests. However, the simulation shows that excessive buffering can increase latency and allow a large volume of stale work to accumulate [13,14].

For digital healthcare services, queue size should therefore be considered together with expected service time and client timeout rather than as an independent capacity parameter.

The relationship

$$Q^* \approx CT$$

provides a simple starting point for evaluating whether a queue can contain more work than the service can process within one client timeout. This relationship should not be interpreted as a universal operational rule because real healthcare systems have multiple service classes, priorities, dependencies, and latency distributions.

### *6. Trigger severity as a resilience-testing indicator*

The trigger-severity index provides a potentially useful planning metric for resilience testing. Before conducting controlled fault injection against a digital healthcare service, engineers could estimate whether the expected temporary backlog is likely to exceed the amount of work the system can process during one timeout period. This may help distinguish between relatively safe and potentially destabilizing chaos-engineering experiments [15]. Importantly, the indicator was derived and tested within the simplified simulation model and should therefore be independently validated before being used as an operational decision rule.

## Threats to Validity

Several limitations should be considered.

First, the model represents a single service tier with exponentially distributed service times and negligible network latency. Real healthcare IT systems often consist of multiple interconnected services, databases, caches, message brokers, authentication systems, and external dependencies. Such architectures may produce more complex amplification effects.

Second, the client population is open-loop. Real users and applications may com-

bine open-loop and closed-loop behavior. In a closed-loop system, backoff can reduce offered load because a client may wait before initiating another transaction [5]. Therefore, the present results should not be interpreted as demonstrating that backoff is ineffective in all environments. Third, only one set of circuit-breaker and backoff parameters was tested. Alternative thresholds and timeout values may change quantitative results. Fourth, the analytical indicators were evaluated using a relatively coarse experimental grid. For example, queue capacities between 200 and 1,000 attempts were not systematically evaluated. The indicators should therefore be considered hypotheses for engineering analysis rather than validated universal thresholds. Fifth, all findings are based on simulation. No clinical environment, patient data, or real healthcare service was used. The next stage of research should evaluate the proposed mechanisms using a containerized testbed that reproduces representative digital healthcare workloads.

## Implications for Healthcare IT Resilience

The findings suggest several practical principles for organizations operating digital healthcare services. First, retry policies should be treated as part of the overall reliability architecture rather than as isolated client configuration parameters. Second, healthcare applications should impose explicit limits on retry traffic. A failure condition should not automatically allow every client to multiply its request rate. Third, request deadlines should propagate through service layers so that servers can avoid processing work that is already obsolete. Fourth, queue capacity should be evaluated in relation to service capacity and client timeout rather than maximized without considering latency. Fifth, resilience testing should include scenarios in which a temporary fault triggers an increase in legitimate application traffic. Finally, healthcare organizations should consider self-inflicted overload as a distinct failure mode in business continuity and digital-service resilience planning.

## Conclusion

A distributed healthcare-oriented service can effectively deny service to itself when transient overload causes client timeouts and uncontrolled retries. In 2,220 simulation runs, immediate retry, exponential backoff, and backoff with jitter were unable to prevent metastable failure under sufficiently severe triggers. In the reference scenario, request success decreased to approximately 3%, while approximately 98% of completed server work was wasted because responses arrived after client timeout. Retry budgets and

circuit breakers substantially reduced this risk by limiting the rate at which additional requests entered the system. Server-side deadline checking reduced stale processing, although it was insufficient as a stand-alone mechanism. Combining retry-budget control with deadline checking produced approximately 93.9% request success in the reference scenario and the lowest peak queue. The results indicate that the resilience of digital healthcare services depends not only on increasing computational capacity but also on controlling the feedback mechanisms created by automated clients. Exponential backoff and jitter remain useful for controlling the timing of retries, but they should be complemented by explicit retry-rate limits, queue-delay control, and server-side deadline enforcement. The trigger-severity index and queue-capacity threshold provide simple simulation-derived indicators that may assist in preliminary resilience assessment. Their application to real healthcare infrastructures requires further validation. Future work should evaluate these mechanisms in multi-tier healthcare architectures, including EHR systems, telemedicine platforms, patient-facing applications, and service-to-service integrations. Validation on a containerized testbed with representative workloads and controlled fault injection will be an important next step.

## References


1. Gelishvili T, Kuchava G, Sturua T. Self-Inflicted Denial of Service (Self-DDoS) in Distributed Systems, Mechanisms and Mitigating Architectural Strategies: When a System „Attacks“ Itself, An Analysis of Retry Storms, Thundering Herds, and Cascading Failure. Computational and Applied Science. 2026;1(2):141-148. doi:10.68218/2ddp5705.
2. Huang L, Magnusson M, Muralikrishna AB, Estyak S, Isaacs R, Aghayev A, Zhu T, Charapko A. Metastable failures in the wild. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22). 2022:73-90.
3. Brooker M. Exponential backoff and jitter. AWS Architecture Blog. 2015.
4. Brooker M. Fixing retries with token buckets and circuit breakers. 2022.
5. Brooker M. What is backoff for? 2022.
6. Beyer B, Jones C, Petoff J, Murphy NR, eds. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA: O'Reilly Media. 2016.
7. Nygard MT. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh, NC: Pragmatic Bookshelf. 2018.

8. Isaacs R, Alvaro P, Majumdar R, Reddy K, Salamati M, Soudjani S. Analyzing metastable failures. In: Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '25). ACM. 2025.
9. Alvaro P, Isaacs R, Majumdar R, Muniswamy-Reddy KK, Salamati M, Soudjani S. Formal analysis of metastable failures in software systems. arXiv:2510.03551. 2025.
10. Habibi F, Lorido-Botran T, Showail A, Sturman DC, Nawab F. MSF-Model: Queuing-based analysis and prediction of metastable failures in replicated storage systems. In: Proceedings of the 43rd International Symposium on Reliable Distributed Systems (SRDS). IEEE. 2024.
11. Farahbakhsh A, Lu Q, Alvisi L, Haeberlen A, Van Renesse R. Characterizing metastable faults and failures. arXiv:2606.00942. 2026.
12. Dean J, Barroso LA. The tail at scale. Communications of the ACM. 2013;56(2):74-80.
13. Gettys J, Nichols K. Bufferbloat: Dark buffers in the Internet. Communications of the ACM. 2012;55(1):57-65.
14. Nichols K, Jacobson V. Controlling queue delay. Communications of the ACM. 2012;55(7):42-50.
15. Basiri A, Behnam N, de Rooij R, Hochstein L, Kosewski L, Reynolds J, Rosenthal C. Chaos engineering. IEEE Software. 2016;33(3):35-41.