

# Optimizing Bucket-Based Software Rejuvenation in Microservices Subject To Variable Workload

Leonardo Scommegna  
*University of Florence*  
Florence, Italy  
leonardo.scommegna@unifi.it

Alberto Avritzer  
*EsulabSolutions Inc.*  
NJ, USA  
beto@esulabsolutions.com

Laura Carnevali  
*University of Florence*  
Florence, Italy  
laura.carnevali@unifi.it

Andrea Janes  
*Free University of Bozen-Bolzano*  
Bolzano, Italy  
andrea.janes@unibz.it

Helena Rodrigues  
*University of Minho*  
Guimarães, Portugal  
helenad@dsi.uminho.pt

Enrico Vicario  
*University of Florence*  
Florence, Italy  
enrico.vicario@unifi.it

**Abstract**—Modern microservices architectures are designed for continuous operations but remain highly susceptible to software aging, a progressive degradation that causes increased response times and potential system failures. While software rejuvenation (i.e., controlled restart) is a common countermeasure, determining optimal rejuvenation instants is difficult under time-varying workload, as transient load spikes can easily be mistaken for aging symptoms. Existing bucket-based monitoring algorithms can filter out latency fluctuations due to load, but lack systematic guidelines to configure the number and depth of buckets.

We propose a Generalized Stochastic Petri Net (GSPN) model of a microservice subject to variable workload and bucket-based software rejuvenation, using Bernstein Phase Type (BPH) distributions to represent stochastic durations. To quantitatively evaluate the interactions among workload profiles, aging accumulation, and bucket-based rejuvenation policies, we analyze the model under different workload patterns and bucket configurations. The experimental results provide actionable insights into how the bucket parameterization impacts the system throughput, availability, and Quality of Service (QoS) violations.

**Index Terms**—Microservices, software aging, bucket-based software rejuvenation, time-varying workload, stochastic models.

## I. INTRODUCTION

Modern software applications are increasingly built as microservices architectures, where functionality is decomposed into fine-grained, independently deployable services running in virtualized environments under the control of orchestration platforms [1], [2]. Each microservice is designed for continuous, long-term operation, which exposes it to software aging, i.e., a progressive degradation of its internal state, caused, for instance, by resource exhaustion, memory fragmentation, or the accumulation of latent errors, that manifests as increasing response times and, in the worst case, as system failure [3]–[5]. Empirical evidence confirms that aging affects microservice-based and containerized systems as well [6], [7]. The most widely adopted countermeasure is software rejuvenation, i.e., the proactive and controlled restart of the service to bring it back to a clean state before the degradation becomes critical [8], [9]. The effectiveness of rejuvenation depends crucially on when it is triggered: too frequent interventions reduce service

availability, whereas overly delayed interventions expose the system to quality of service (QoS) violations or to failures.

In realistic settings, systems operate under a workload that varies over time, affecting both the observed response times and the rate at which aging accumulates [10]–[12]. This raises a concrete problem for rejuvenation strategies driven by response-time monitoring: a physiological increase in latency, caused by a temporary spike in workload, may be mistaken for a symptom of aging, triggering unnecessary rejuvenations that remove capacity precisely when demand is highest. To mitigate this risk, Avritzer et al. [13] proposed monitoring algorithms able to distinguish degradation due to aging from degradation due to bursts in the arrival process or transient errors. Among these, the bucket mechanism accumulates evidence of response-time threshold violations and triggers rejuvenation only when the collected evidence exceeds a preset level. The effectiveness of this mechanism, however, depends on its parameterization, in particular on the number of buckets and their depth. To date, there is no systematic guidance on how to choose these parameters as a function of the workload profile to which the system is subjected. An inadequate parameterization translates into tangible operational costs: spurious rejuvenations under heavy load, or delayed intervention when aging is underway.

In this work, we propose a stochastic model of a microservice subject to software aging and operating under variable workload and the bucket-based rejuvenation mechanism. The model analysis enables the quantitative evaluation of the interaction among the workload profile, the accumulation of aging, and the rejuvenation policy, providing insight for parameterizing the mechanism (i.e., number and depth of buckets) starting from the system characteristics. We formalize the model as a Generalized Stochastic Petri Net (GSPN) [14], discussing how to define its stochastic parameters from measurements on a real system. The model captures both the stochastic dynamics of the workload and aging processes as well as the discrete logic of the bucket mechanism, and it features Bernstein Phase Type (BPH) [15] distributions to represent stochastic durations, e.g., response times of requests and rejuvenation times. In

particular, BPH distributions preserve important properties of the approximated distributions (e.g., monotonicity, upper and lower bounds) and are easy to derive and analytically tractable.

We solve the model by the ORIS tool and its Sirio library [16], whose Java API enables the parametric studies required to explore the space of bucket configurations. We evaluate the model under three realistic workload profiles: a predominantly medium load, a spiky load with abrupt transitions, and a typical daily cycle. We then analyze how bucket configuration affects system throughput, availability, and QoS violations. The results show that the optimal configuration strongly depends on the workload profile, and that, at equal capacity, many shallow buckets filter load-induced rejuvenations better than a few deep ones, at the price of a higher QoS violation rate.

We provide a replication package [17] available under the AGPLv3 license, containing the GSPN models and the code needed to reproduce the experiments reported in this paper.

## II. SYSTEM MODEL

We consider a microservice that serves requests issued by external clients. Requests are processed concurrently under a *processor time-sharing* policy that partitions the total processing capacity equally among all requests, so that the response time perceived by each request grows with the number of requests served in parallel. While being highly realistic in containerized microservices, this assumption considers purely CPU-bound tasks (e.g., no network delays). To protect the microservice from overload, an admission control policy bounds the number of concurrently served requests to a *maximum concurrency level*  $C$ , rejecting requests arriving when this limit is reached. While omitting the representation of a request queue notably reduces the state space of our model, it also reflects standard best practices in microservices architectures (e.g., load shedding and fail-fast policies). The system also features *time-varying arrival rate of requests*, alternating between different operational regimes (e.g., low-intensity periods and peak hours).

Since the microservice runs continuously, it is subject to *software aging*: as requests are processed, the internal state of the service progressively degrades (e.g., due to resource leaks, fragmentation, or accumulation of latent error conditions), yielding a stochastic increase of response times. Notably, the accumulation of aging in this regard is workload-dependent: the more requests the microservice processes, the faster degradation accumulates, so that periods of high load accelerate aging [9]. This coupling between workload and response times makes the detection of aging nontrivial: an increase in observed latencies may be caused by genuine degradation, but also by a transient peak of concurrent requests in a perfectly healthy system.

To detect software aging while filtering out latency fluctuations due to load, the microservice is monitored by the *bucket mechanism* proposed by Avritzer et al. in [13]. The mechanism is configured with the mean  $\mu$  and the standard deviation  $\sigma$  of the response times of the system in a healthy (non-aged) state while serving a single request, and with  $M$  buckets  $B_1, \dots, B_M$  characterized by depth  $D_1, \dots, D_M$ , respectively, and thresholds  $\mu + \sigma, \dots, \mu + M\sigma$  on response

times, respectively. At any time, the mechanism operates in a bucket  $B_m \in \{B_1, \dots, B_M\}$ . For each observed response time,  $B_m$  is incremented if the response time exceeds  $\mu + m\sigma$ , and decremented otherwise. When  $B_m$  reaches its depth  $D_m$ , it overflows and the mechanism moves to bucket  $B_{m+1}$ ; conversely, when  $B_m$  is emptied, the mechanism moves down to bucket  $B_{m-1}$ . Thus, escalating through the buckets requires a persistent pattern of threshold violations, which is the signature of aging, whereas sporadic violations caused by short-lived load spikes are reabsorbed by the decrement dynamics. When the last bucket  $B_M$  overflows, i.e., the bucket system is full, degradation is deemed severe and persistent enough to warrant an intervention, and a *software rejuvenation* is triggered.

The ability of this mechanism to react to genuine aging while ignoring load-induced latency fluctuations hinges on how the bucket system is configured, namely on the number and depths of buckets. Intuitively, larger number and depths make the mechanism more conservative: escalation demands a longer and more consistent history of threshold violations, which suppresses false positives caused by transient load spikes but also delays the reaction to real degradation, possibly to the point of triggering rejuvenation only after QoS violations have already occurred. Conversely, a shallower and shorter battery of buckets reacts promptly to aging but is more easily filled by load-induced violations, causing spurious rejuvenations that remove capacity precisely when demand is high. A well-chosen configuration therefore lets the mechanism activate essentially only in response to aging, whereas a poor one makes it fire too late or produce false positives. Crucially, the configuration that strikes this balance depends on the workload profile the microservice is subjected to, which motivates the analytical study of the bucket parameters carried out in this work.

## III. STOCHASTIC MODEL

We show how to derive a GSPN model of the microservice system of Section II, identifying the quantities that must be measured on a real system to synthesize the model parameters (Section III-A) and presenting the GSPN itself (Section III-B).

### A. Measurement-Based Parameterization

The parameters needed to represent the system of Section II as a stochastic model can be estimated by observing a real deployment of the microservice, complemented by configuration values that are known by design. We group them as follows.

a) *Workload profile*: The arrival rate of requests in each operational regime (e.g.,  $\lambda_{\text{low}}$ ,  $\lambda_{\text{mid}}$ , and  $\lambda_{\text{high}}$ ) and the transition rates between regimes (e.g.,  $\lambda_{\text{lowToMid}}$ ,  $\lambda_{\text{midToLow}}$ ,  $\lambda_{\text{midToHigh}}$ , and  $\lambda_{\text{highToMid}}$ ) can be estimated from traffic traces of the production system, by identifying the regimes (e.g., through the daily traffic pattern) and measuring, for each regime, the mean inter-arrival time and mean transition time to other regimes. The maximum concurrency level  $C$  is instead a configuration parameter of the microservice, known by design.

b) *Healthy-state response times*: The empirical response time distribution in a healthy (non-aged) state  $L_0$  is obtained from response times collected for a single request right after

a microservice restart. This distribution serves two purposes: *i)* its mean  $\mu$  and standard deviation  $\sigma$  define the thresholds  $\mu + m\sigma$  of the bucket system, as in the monitoring architecture of [13]; *ii)* it is fit by a BPH distribution  $F_0(t)$  with  $N$  phases.

*c) Aging quantification:* Characterizing the system degradation requires observing and quantifying software aging, which is the most crucial aspect of our parameterization, as no single accepted metric of software aging exists. Thus, degradation must be inferred from indirect indicators, such as the consumption of system resources (e.g., resident memory) or the drift of response times once load effects are filtered out, and mapped onto a finite set of discrete aging levels  $L_1, \dots, L_E$ .

*d) Aging-state response times:* Building on this quantification, the empirical response time distribution must then be characterized under degraded conditions, i.e., for increasing aging levels. To this end, during the system execution, response times are recorded from a non-aged start until the maximum reachable aging level  $L_E$  is attained. In this way, response time observations are gathered for each aging level  $L_1, \dots, L_E$  and fit by BPH distributions  $F_1(t), \dots, F_E(t)$ , respectively.

*e) Aging accumulation rate:* The same observation experiment also provides the aging accumulation probability  $p_e$ , i.e., the probability that serving a request increases the aging level from  $L_e$  to  $L_{e+1} \forall e \in \{0, \dots, E-1\}$ , which can be estimated as the fraction of requests served between successive increments of the aging indicator. Note that expressing aging accumulation per served request, rather than per unit of time, makes the model dependent on workload by construction: the higher the load, the more requests are processed, and the faster degradation accumulates.

*f) Threshold violation probabilities:* For each healthy and aging level  $L_e \in \{L_0, L_1, \dots, L_E\}$ , each concurrency condition  $c \in \{1, \dots, C\}$ , and each bucket  $B_m \in \{B_1, \dots, B_M\}$ , the probability that a completed request exceeds the threshold  $\mu + m\sigma$  of bucket  $B_m$  is not measured directly, but rather derived from the quantities above as  $q_{ecm} = 1 - F_e((\mu + m\sigma)/c)$ . In the experimental evaluation of Section IV, we computed  $q_{ecm}$  for  $e \in \{0, \dots, 4\}$  and  $c, m \in \{1, \dots, 4\}$ , yielding 80 distinct values of  $q_{ecm}$ . We validated this analytical derivation against simulation for all 80 values of  $q_{ecm}$  used in Section IV, obtaining a 96.25% coverage of the 95% confidence intervals, consistent with the chosen confidence level. More details are provided in the replication package [17].

*g) Rejuvenation time:* The empirical distribution of the rejuvenation time (independent of the response time) is obtained by repeatedly measuring the duration of controlled microservice restarts and it is fit by a BPH distribution  $G(t)$  with  $K$  phases.

## B. GSPN Model

Figure 1 illustrates the GSPN model of a microservice described in Section II. It consists of five submodels (red-border boxes) representing: *i)* the workload process, *ii)* the aging process, *iii)* the arrival and service processes, *iv)* the bucket system, and *v)* the rejuvenation process. The workload and aging submodels are independent. The arrival and service process submodel depends on the workload and the aging

process submodels as the intensity of request arrivals depends on the load, and the response time distribution depends on the aging level. The bucket system submodel depends on the aging process submodel, since threshold violations depend on the response time distribution and thus on the aging level; and the rejuvenation process submodel depends on the bucket system submodel, since a rejuvenation is triggered as soon as the bucket system becomes full.

Two additional submodels (identified by blue-border boxes in Fig. 1) represent the BPH distributions of the response times ( $N$  phases) and rejuvenation times ( $K$  phases). A BPH distribution with  $N$  phases is a chain of  $N$  exponential transitions with rates  $1, \dots, N$ , where the weights of  $N$  immediate transitions encode the initial phase probabilities, so that different BPH distributions differ only in these weights [15].

The *workload submodel* represents the fluctuations in user demand during different operational periods (e.g., peak hours, idle periods) and consists of three states modeled by places Low, Mid, and High. The transitions between these states are characterized by the exponential durations modeled by transitions lowToMid, midToLow, midToHigh, and highToMid, with rates  $\lambda_{\text{lowToMid}}$ ,  $\lambda_{\text{midToLow}}$ ,  $\lambda_{\text{midToHigh}}$ , and  $\lambda_{\text{highToMid}}$ , respectively, estimated in Section III-A. While this model with three alternating states represents one specific workload profile capturing typical daily traffic variations, other workload patterns could be represented as well.

In the *arrival and service process submodel*, the exponential transition arrive represents the arrival of requests, the response time submodel represents the response time BPH distribution, and place RequestsBeingServed encodes the total number of concurrent requests currently being processed. In particular, the weights of the immediate transitions  $p_1, \dots, p_N$  in the response time submodel depend on the marking of place AgingLevel, so that each aging level identifies a different response time BPH distribution (i.e., if AgingLevel contains  $0, 1, \dots, E$  tokens, then the weights are those of the distribution  $F_0, F_1, \dots, F_E$ , respectively, estimated in Section III-A). Moreover, given that the microservice serves multiple requests concurrently using a time-sharing policy, the total processing capacity is shared equally among all active jobs. Consequently, the firing rates of the exponential transitions within the response time submodel are marking-dependent. Specifically, the firing rate  $\rho_n = n \cdot \text{RespPhn} / \text{RequestsBeingServed}$  of the  $n$ -th transition respPhn is equal to the base rate  $n$  of the  $n$ -th phase, multiplied by the number of tokens in place RespPhn to account for concurrent service of requests, and divided by the number of tokens in place RequestsBeingServed to account for processor time-sharing among requests.

As specified in Table I, transition arrive has an enabling function that evaluates to true if the number of requests currently being served is lower than the number  $C$  of tokens in place MaxConcurrency (i.e.,  $\text{RequestsBeingServed} < \text{MaxConcurrency}$ ) and if the system is not undergoing a rejuvenation (i.e.,  $\text{RejEnd} == 1$ ). Additionally, the rate of arrive scales dynamically with the marking of the workload submodel, e.g., in Fig. 1, rate  $\lambda_{\text{low}}$ ,  $\lambda_{\text{mid}}$ , or  $\lambda_{\text{high}}$  (estimated in

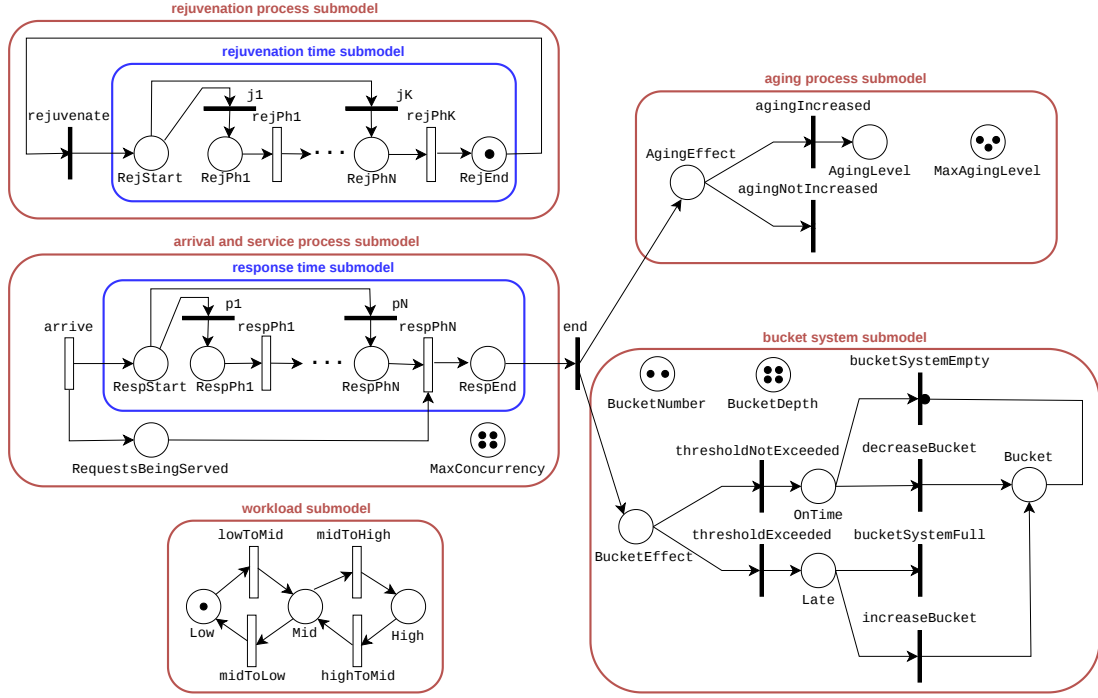


Fig. 1: GSPN model of a microservice.

| Transition       | Enabling Function                                        | Update Function                                                       |
|------------------|----------------------------------------------------------|-----------------------------------------------------------------------|
| arrive           | $RequestsBeingServed < MaxRequests \ \&\& \ RejEnd == 1$ | -                                                                     |
| agingIncreased   | $AgingLevel < MaxAgingLevel$                             | -                                                                     |
| increaseBucket   | $Bucket < BucketNumber * BucketDepth$                    | -                                                                     |
| bucketSystemFull | $Bucket == BucketNumber * BucketDepth$                   | -                                                                     |
| rejuvenate       | $Bucket == BucketNumber * BucketDepth$                   | $Bucket, RespStart, \dots, RespEnd, RequestsBeingServed \leftarrow 0$ |
| rejPhK           | -                                                        | $AgingLevel \leftarrow 0$                                             |

TABLE I: Enabling functions and update functions of the transitions of the GSPN model shown in Fig. 1.

Section III-A) is assigned if place Low, or Mid, or High contains a token, respectively. This model effectively implements a Markov-Modulated Poisson Process (MMPP), capturing time-inhomogeneous behaviors (e.g., periodic patterns).

As soon as a request completes its final phase of service, the firing of transition `respPhN` adds a token to place `RespEnd`, enabling the immediate transition `end`. Its firing removes a token from place `RequestsBeingServed` to reflect the service completion, and adds a token to places `AgingEffect` and `BucketEffect` to update the aging level and the bucket system, respectively. Specifically, the *aging process submodel* captures the gradual degradation of the microservice as it processes requests over time. The weights of the concurrent immediate transitions `agingIncreased` and `agingNotIncreased` are the probability  $p_e$  of aging accumulation (estimated in Section III-A) and its complementary probability  $1 - p_e$ , respectively. Thus, they depend on the current aging level  $L_e$ , represented by the number of tokens in place `AgingLevel`, so that, as the microservice accumulates damage, the probability of moving to a higher aging state changes, e.g., in the model of Fig. 1, we consider the linear relation with weight  $p_e =$

$(AgingLevel+1)/(MaxAgingLevel+2)$ . Additionally, as detailed in Table I, transition `agingIncreased` has an enabling function that evaluates to true if the aging level is lower than the maximum level  $L_E$  modeled by the number  $E$  of tokens in place `MaxAgingLevel`, i.e., once the maximum degradation state is reached, no further aging increments can occur until a rejuvenation process is executed.

The *bucket system submodel* captures detection of performance degradation under different aging levels and workload conditions. The number of tokens in places `BucketNumber` and `BucketDepth` encode the number  $M$  and depth  $D$  of buckets, respectively (note that the same depth is considered here for all buckets only to reduce the cluttering). The number of tokens in place `Bucket` is an aggregate variable modeling the level of the current bucket  $B_m \in \{B_1, \dots, B_M\}$  as an integer ranging between  $(m-1)D+1$  and  $mD$ . According to this, the bucket system is full when the number of tokens in place `Bucket` becomes equal to  $MD$ . The weights of the concurrent immediate transitions `thresholdExceeded` and `thresholdNotExceeded` model the probability  $q_{ecm}$  (estimated in Section III-A) that the response time of the completed request exceeded the threshold of the current bucket  $B_m$ ,

and its complement  $1 - q_{ecm}$ . They thus depend on the marking of places `AgingLevel`, `RequestsBeingServed` and `Bucket`, identifying the aging level  $L_e$  (and thus the response time BPH distribution), the concurrency level  $c$  and the current bucket  $B_m$ , respectively. If the threshold was exceeded, the tokens in place `Bucket` are increased (unless the bucket is full), otherwise decreased (unless the bucket is empty). Accordingly, as specified in Table I, transitions `increaseBucket` and `bucketSystemFull` have enabling functions that evaluate to true if the bucket is not full and if it is full, respectively. To avoid nondeterminism and to account for the current aging level in the bucket update, the immediate transitions of the bucket system submodel have higher priority than those of the aging process submodel.

Finally, the *rejuvenation process submodel* triggers a rejuvenation whenever the bucket system is full. In fact, as specified in Table I, the immediate transition `rejuvenate` has the same enabling function as transition `bucketSystemFull`. To represent the system restart, transition `rejuvenate` also has an update function that flushes place `Bucket` and all the places of the arrival and service process submodel except for `MaxConcurrency`. Moreover, transition `rejPhK` has an update function that flushes place `AgingLevel` (this place is not flushed by the firing of `rejuvenate` so that we can observe the current aging level when rejuvenation is triggered). The rejuvenation time is characterized by the BPH distribution  $G(t)$  with  $K$  phases estimated in Section III-A, which is represented by the rejuvenation time submodel.

#### IV. STOCHASTIC ANALYSIS

##### A. Research Questions

The analysis addresses the following research questions:

- **RQ1:** What is the impact of the number and the depth of buckets on throughput, availability, and QoS violations?
- **RQ2:** What is the impact of the number and the depth of buckets on the ability to avoid unnecessary rejuvenations caused by transient load peaks rather than software aging?

##### B. Setup

We considered 1 healthy system state and 4 aging levels, i.e.,  $E = 4$ , modeling the corresponding response time distributions  $F_0(t), \dots, F_4(t)$ , respectively, as stochastically ordered BPH distributions with  $N = 5$  phases (response times increase with aging). The rejuvenation time was also characterized by a BPH distribution with  $K = 5$  phases. The initial probability of each phase of each distribution is reported within the artifact [17]. Moreover, we set a maximum concurrency of  $C = 4$  requests.

We considered 48 model configurations obtained by varying the number  $M$  and depth  $D$  of buckets, i.e.,  $M, D \in \{1, \dots, 4\}$ , and the workload profile, i.e., 3 variants *MostlyMid*, *Spiky*, and *DailyPattern* of the workload submodel of Fig. 1, that share the same arrival rate  $\lambda_{\text{low}} = 1$ ,  $\lambda_{\text{mid}} = 3$ , and  $\lambda_{\text{high}} = 6$  in the low-load, medium-load, and high-load states, respectively:

- *MostlyMid*: a three-state profile (Low, Mid, High) in which the system predominantly operates at a medium

load, with sporadic transitions towards the low- and high-load states (both with rate 0.05), followed by a quick return to the mid-load state (rate 0.5 in both cases).

- *Spiky*: a two-state profile (Low, High) in which the system mostly runs at a low load, while sporadic spikes (with rate 0.05) drive it into a temporary overload before returning to the low-load state (with rate 0.1).
- *DailyPattern*: a five-state profile reproducing a typical daily cycle, consisting of a low nighttime load (Low), a medium morning load (Mid1), a high lunchtime load (High1), a medium afternoon load (Mid2), and a high evening load (High2). The transition rates reflect the corresponding phase durations: 10 night hours (Low  $\rightarrow$  Mid1, rate 0.1), 4 morning hours (Mid1  $\rightarrow$  High1, rate 0.25), 2 lunchtime hours (High1  $\rightarrow$  Mid2, rate 0.5), 4 afternoon hours (Mid2  $\rightarrow$  High2, rate 0.25), and 4 evening hours (High2  $\rightarrow$  Low, rate 0.25).

Given that a rejuvenation completely resets the system, we performed transient analysis of each model between consecutive rejuvenations, i.e., with stop condition `RejStart==1`, with variable time limit from model configuration to model configuration (such that the probability that a rejuvenation is triggered by that time is not lower than 0.999). We evaluated the following measures, which can be expressed as rewards in Sirio (i.e., expressions combining constants and token counts so as to define real-valued functions over the set of markings):

- *Throughput*  $T(t)$ : the expected number of requests completed by the system per unit of time. It is expressed by the reward  $\text{RespPhN} \cdot \lambda_{\text{respPhN}}$ , i.e., the number of tokens in the last phase of the response time BPH distribution multiplied by the exponential rate of the last phase.
- *QoS violation rate*  $V(t)$ : the expected number of requests completed per unit of time, whose response time exceeds the threshold  $\theta = \mu + \sigma$  of the first bucket  $B_1$ , where  $\mu$  and  $\sigma$  are the mean and standard deviation of the response time in the healthy state, respectively. Since  $\theta$  and  $\sigma$  are independent of the bucket level,  $V(t)$  measures the perceived QoS against a constant target. It is expressed by the reward  $T(t) \cdot q_{ec1}$ , where  $q_{ec1}$  is the probability that a request completed under the current aging level  $L_e$  and concurrency  $c$  exceeds  $\theta + \sigma$ . For each value of  $L_e$  and  $c$ ,  $q_{ec1}$  is precomputed as discussed in Section III-A.
- *Time-to-rejuvenation-due-to-load distribution*  $J_{\text{load}}(t)$ : the probability that a rejuvenation is triggered by time  $t$  while the aging level is not maximum, which occurs if the bucket system fills up because a high load temporarily worsens performance. This measure is expressed by the reward `RejStart==1 && AgingLevel<MaxAgingLevel`.
- *Availability*  $A$ : the expected fraction of time during which the system is up, computed as the expected value of the time-to-rejuvenation divided by the expected value of the sum of the time-to-rejuvenation and the rejuvenation time.

For  $J_{\text{load}}(t)$ , we consider the value  $P$  at the time limit, i.e., the probability of rejuvenation due to load. For  $T(t)$  and  $V(t)$ , we consider the average values  $T$  and  $V$  over time, respectively.

### C. Results

#### RQ1: Impact on throughput, availability, and QoS violations:

Figure 2 reports the throughput-availability trade-off across the three workload profiles. Each configuration is identified by its number of buckets  $M$  (marker color, from light to dark) and bucket depth  $D$  (marker shape). The gray line traces the Pareto front in each panel. We denote by  $M_iD_j$  the configuration with  $M = i$  and  $D = j$  and refer to  $Z = MD$  as its *capacity*, i.e., the difference between the number of late and on-time completions the system tolerates before triggering rejuvenation.

For the *Spiky* and *DailyPattern* profiles, the Pareto front collapses onto a single point: the largest configuration  $M_4D_4$  dominates in both throughput  $T$  and availability  $A$  (e.g., *Spiky*:  $T = 2.23$ ,  $A = 0.991$ ), so no trade-off arises. Throughput is bounded by arrivals not by service degradation: arrivals are intermittent, so even an aged server keeps pace with the offered load and rarely saturates the concurrency limit, leaving throughput insensitive to aging. Aging is not avoided with large buckets: rejuvenation is almost always triggered by reaching the maximum aging level, but its cost falls entirely on response times. Accordingly, the violation rate  $V$  grows from 0.58 to 0.91 for *Spiky* (0.60 to 0.92 for *DailyPattern*) as capacity increases. Deferring rejuvenation thus trades per-request quality for the downtime it avoids, and, on the throughput-availability plane, larger buckets are clearly preferable.

The *MostlyMid* profile behaves differently. Here a genuine trade-off emerges and the Pareto front spans 11 configurations. Throughput is maximized at intermediate capacity ( $M_1D_4$ : 3.61 and  $M_2D_2$ : 3.55, both with  $Z = 4$ ) and then declines as capacity grows ( $M_4D_4$ : 2.91, a 19% reduction relative to the peak). Under sustained arrivals the server is rarely idle: aged service keeps the concurrency limit saturated, so incoming requests are blocked and aging limits throughput directly. Postponing rejuvenation keeps the system longer in its slow, aged states, and the induced slowdown exceeds the downtime saved by rejuvenating less often. As a consequence, the throughput-optimal configuration lies in the interior of the front, well separated from the availability-optimal one, and selecting an operating point becomes a real design decision rather than a matter of maximizing capacity.

Taken together, these results indicate that the value of a large bucket system depends on how demanding the arrival process is. When arrivals are bursty and intermittent, as in *Spiky* and *DailyPattern*, aging does not limit throughput and generous capacity pays off on both objectives at once (higher throughput and higher availability), at the price of a higher fraction of late requests. When arrivals are sustained, as in *MostlyMid*, aging becomes the binding constraint and additional capacity trades throughput for availability, so a moderate configuration is preferable and the operating point must be chosen deliberately.

**RQ2: Ability to avoid load-induced rejuvenations:** Figure 3 reports one heatmap per workload profile. In each heatmap, the axes are the number of buckets  $M$  and the bucket depth  $D$ , and the color encodes the probability of a load-induced rejuvenation, i.e., the probability that a rejuvenation is triggered while the

aging level has not yet reached its maximum. This probability is taken as the asymptotic value reached by the transient analysis of the *time-to-rejuvenation-due-to-load* distribution reward.

Across all three workload profiles, the heatmaps share the same qualitative pattern: low values of  $M$  and  $D$  yield a high probability of premature rejuvenation. However,  $M$  and  $D$  are not interchangeable, as the heatmaps are not symmetric about their diagonal. At equal capacity (e.g.,  $M_1D_4$ ,  $M_2D_2$ ,  $M_4D_1$ ), many shallow buckets systematically yield a lower probability of load-induced rejuvenation than a few deep ones (*MostlyMid*:  $\sim 0.50$ ,  $\sim 0.33$ ,  $\sim 0.12$ ), and the mean time to rejuvenation and the availability follow the same ordering. This reverses for the fraction of requests exceeding the QoS threshold: at equal capacity, larger  $M$  keeps the system aged longer and thus degrades per-request quality the most (*Spiky*,  $Z = 4$ : 0.75 for  $M_1D_4$  vs. 0.83 for  $M_4D_1$ ). Many shallower buckets are then preferable for availability, at the cost of lower service quality.

The gap between equal-capacity configurations stems from the bucket-overflow mechanism and the defined thresholds. Each successive bucket raises the threshold a response time must exceed to add a token, making it harder to fill. Splitting capacity into many shallow buckets thus imposes progressively looser thresholds that defer rejuvenation, letting response times drift further from the fixed QoS target in the meantime.

**Guidelines:** For bursty workloads, the total bucket capacity should be maximized to boost throughput and availability. For sustained workloads, an intermediate capacity should be used to prevent severe aging-induced slowdowns. When splitting a fixed capacity, many shallow buckets prioritize system uptime, while fewer deep buckets enforce response-time quality.

### V. CONCLUSIONS AND FUTURE WORK

We analyzed the bucket-based rejuvenation mechanism in microservices subject to time-varying workload. To this end, we developed a GSPN model capturing both the discrete logic of the bucket system and the stochastic dynamics of variable workloads and aging accumulation, providing guidance on how to derive the GSPN stochastic parameters from measurements. We evaluated the model under three representative workload profiles to explore how different number and depth of buckets affect system throughput, availability, and QoS violations. The analysis answers both RQs: their impact (RQ1) is workload-dependent, with large capacities dominating under intermittent arrivals and a trade-off arising under sustained load; and the ability to avoid load-induced rejuvenations (RQ2) is governed not only by the total capacity  $MD$  but also by its split, with many shallow buckets outperforming a few deep ones at equal capacity, at the cost of a higher QoS violation rate.

Notably, the intuition that larger bucket systems uniformly perform better, as suggested by random sampling arguments on the threshold statistics, holds only under intermittent arrivals: violations are correlated through aging and workload dynamics, and under sustained load the optimum lies at intermediate capacity, which motivates a model-based analysis.

Future work will validate our model-based approach through integration with a real microservice deployment, supporting

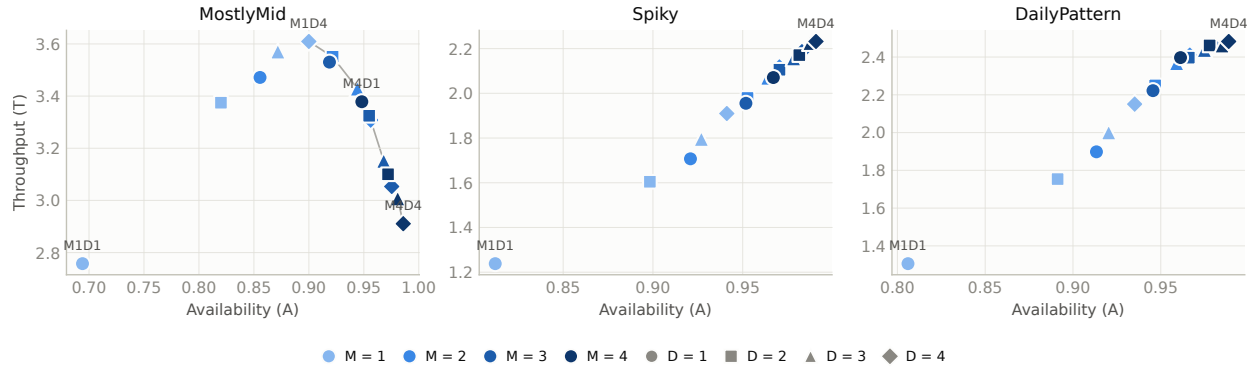


Fig. 2: Throughput-availability trade-off of the bucket configurations across the different workload profiles.

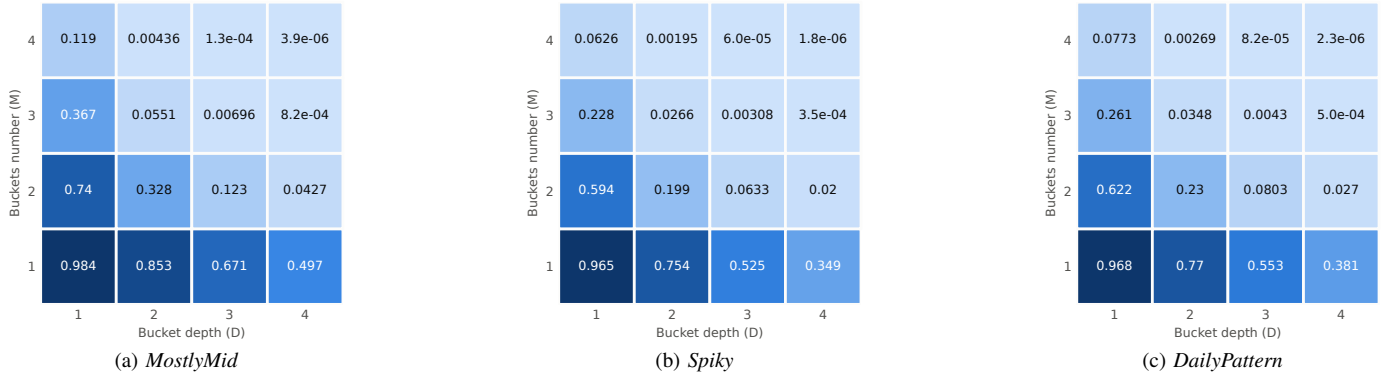


Fig. 3: Heatmaps of the probability of rejuvenation due to load across the different workload profiles.

adjustment of the bucket configuration at runtime based on actual monitoring data and workload transitions. Additionally, we will study how to extend our approach to model multiple microservice replicas and multiple microservices, investigating how cascaded performance degradation across chained services influences the optimal rejuvenation policy.

## REFERENCES

- [1] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *J. of Network and Comp. Appls.*, vol. 60, pp. 54–67, 2016.
- [2] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th international conference on software quality, reliability and security (QRS)*. IEEE, 2019, pp. 176–185.
- [3] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)*. Ieee, 2008, pp. 1–6.
- [4] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, pp. 1–34, 2014.
- [5] T. Dohi, K. S. Trivedi, and A. Avritzer, *Handbook of software aging and rejuvenation: fundamentals, methods, applications, and future directions*. World scientific, 2020.
- [6] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A study on the aging and fault tolerance of microservices in kubernetes," *IEEE Access*, vol. 10, pp. 132 786–132 799, 2022.
- [7] L. Scommegna, A. Avritzer, L. Carnevali, and E. Vicario, "Quantitative modeling and evaluation of software aging and rejuvenation in microservices," in *2025 IEEE 36th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2025, pp. 322–329.
- [8] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Int. symp. on fault-tolerant computing. Digest of papers*. IEEE, 1995, pp. 381–390.
- [9] R. Pietrantuono and S. Russo, "A survey on software aging and rejuvenation in the cloud," *Softw. Quality J.*, vol. 28, no. 1, pp. 7–38, 2020.
- [10] Y. Bao, X. Sun, and K. S. Trivedi, "A workload-based analysis of software aging, and rejuvenation," *IEEE Transactions on Reliability*, vol. 54, no. 3, pp. 541–548, 2005.
- [11] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, "Workload characterization for software aging analysis," in *IEEE Int. Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 240–249.
- [12] D. Wang, W. Xie, and K. S. Trivedi, "Performability analysis of clustered systems with rejuvenation under varying workload," *Performance Evaluation*, vol. 64, no. 3, pp. 247–265, 2007.
- [13] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyerker, "Performance assurance via software rejuvenation: Monitoring, statistics and algorithms," in *International conference on dependable systems and networks (DSN'06)*. IEEE, 2006, pp. 435–444.
- [14] M. Ajmone Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems," *ACM Trans. on Comp. Sys.*, vol. 2, no. 2, pp. 93–122, 1984.
- [15] A. Horváth, I. Horváth, M. Paolieri, M. Telek, and E. Vicario, "Approximation of cumulative distribution functions by bernstein phase-type distributions," *Performance Evaluation*, vol. 168, p. 102480, 2025.
- [16] M. Paolieri, M. Biagi, L. Carnevali, and E. Vicario, "The oris tool: quantitative evaluation of non-markovian systems," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1211–1225, 2021.
- [17] L. Scommegna, A. Avritzer, L. Carnevali, E. Vicario, A. Janes, and H. Rodrigues, "Replication package for the paper "optimizing bucket-based software rejuvenation in microservices subject to variable workload"," Jul. 2026. [Online]. Available: <https://github.com/oris-tool/WoSAR-2026-bucket-system-rep-pkg>