

Empirical Assessment of Software Aging and Rejuvenation in Microservices

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

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

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

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

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

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

Abstract—This paper presents an empirical assessment of software aging and rejuvenation using the TrainTicket benchmark deployed under multiple microservice configurations. We evaluate the combined effects of service replication, HTTP pooling, and measurement-based bucket rejuvenation policies including earlier rejuvenation thresholds, per-replica rejuvenation, and replica switching. The experiments monitor response time, Goodput/sec, CPU utilization, memory consumption, and a composite Power (Goodput/Response Time) metric that captures the trade-off between throughput, failures, and response time. The results demonstrate that software aging persists in microservice applications despite containerization and replication. Replication alone delays but does not eliminate performance degradation, while HTTP pooling substantially reduces the communication overhead associated with HTTP connection set-up and improves long-term performance. Among the evaluated rejuvenation strategies, independent per-replica rejuvenation and switching, with 2 replicas for all services, combined with HTTP pooling achieves the highest Power metric values.

Index Terms—software rejuvenation, bucket algorithm, microservices, empirical assessment.

I. INTRODUCTION

Software aging is the smooth degradation of software performance, which could occur as a consequence of the accumulation of rounding errors, resource exhaustion, memory fragmentation, unreleased resources, or other errors during a prolonged execution [1]. As software ages, the system might experience service disruptions, because these errors propagate to the service boundary as service failures. Software rejuvenation has been proposed to mitigate these failures by refreshing the system execution environment, for example by restarting processes or containers before service failures occur [2].

Although software aging has been extensively studied for monolithic and distributed systems [3], [4], comparatively little work has investigated how aging manifests in containerized microservice applications deployed under modern orchestration platforms [5]. In these environments, virtualization, heterogeneity, elasticity, replication, and container restarts, are commonly used, introducing new architecture challenges. For example HTTP pooling interactions makes it difficult to

determine when rejuvenation should occur. Existing studies typically evaluate rejuvenation in isolation and provide limited insight into the combined effects of replication, communication mechanisms, and runtime resource contention.

We present an empirical assessment of software aging and rejuvenation in a representative microservice application. Using the TrainTicket benchmark, we conduct controlled experiments under sustained workloads while varying the number of service replicas, the use of HTTP pooling, and several rejuvenation policies. The experiments monitor response time, throughput, failures, CPU utilization, and memory consumption, which are used to assess service degradation over a fixed execution period of one hour. Each configuration was executed once for one hour; therefore the reported statistics characterize temporal variability rather than experimental variability. We repeated the experiments in two different environments. The primary environment results are reported in this paper. The second set of results using double memory in the host are presented in the replication package [6].

We investigate the following research questions:

- **RQ1:** How do replication and HTTP pooling influence software aging?
- **RQ2:** Which rejuvenation policy provides the best long-term performance?
- **RQ3:** What deployment configuration maximizes the Power metric?

The experimental results presented in this paper demonstrate that software aging remains a concern in cloud-native microservices despite containerization and replication. Increasing the number of replicas delays but does not eliminate degradation, while HTTP pooling substantially reduces the severity of aging effects. Furthermore, conventional threshold-based rejuvenation policies alone are insufficient to stabilize long-running performance. The best results are achieved by combining HTTP pooling with per-replica rejuvenation policies, enabling targeted recovery of aging containers while minimizing disruption to healthy replicas.

This paper makes three contributions. First, it adapts the measurement-based bucket algorithm for cloud-native microservices by introducing per-replica rejuvenation, replica switching, and configurable rejuvenation thresholds. Second, it presents an empirical evaluation of these strategies using the TrainTicket benchmark under multiple deployment configurations. Third, it identifies deployment and rejuvenation combinations that maximize customer-oriented performance as measured by the Power metric introduced by Kleinrock [7], which tracks the tradeoff between Goodput and Response time. To the best of our knowledge, this is the first empirical study that jointly evaluates deployment configuration (replication and HTTP connection pooling) and measurement-based software rejuvenation policies in a cloud-native microservice benchmark.

The remainder of this paper is organized as follows. Section II reviews the background on software aging, software rejuvenation, and microservice systems. Section III presents the experimental methodology. Section IV reports the empirical results. Section V discusses the implications of the findings. Finally, Section VI concludes the paper and outlines directions for future work.

II. RELATED WORK

Software aging and rejuvenation have been active areas of research for over three decades. The research has evolved from demonstrating the existence of software aging phenomena in long-running systems, to developing analytical models, and adaptive rejuvenation strategies.

A. Software Aging and Rejuvenation

Software aging is the progressive degradation of software performance as a function of execution time. As software ages, systems exhibit increasing failure rates and eventually service outages. Huang et al. [1] introduced software rejuvenation terminology as a proactive fault-management technique in which software is periodically restarted or refreshed before aging-induced failures occur. Their work demonstrated that scheduled rejuvenation could significantly improve system availability compared with reactive recovery after failures. Cotroneo et al. [2] provided comprehensive surveys of software aging mechanisms and rejuvenation policies, covering deterministic, stochastic, and measurement-based approaches. These studies established software rejuvenation as a mature dependability technique applicable to operating systems, middleware, telecommunications, database servers, and distributed applications.

B. Empirical Characterization

While early work focused primarily on demonstrating aging phenomena and optimizing rejuvenation intervals [8], subsequent research emphasized understanding the underlying performance degradation mechanisms through empirical experimentation and analytical modeling.

Avritzer and Weyuker [4] presented a strategy for determining when it is advantageous to implement process restarts

to restore a smoothly degrading system (i.e., an aging system) to full capacity by using process restarts (i.e., software rejuvenation). They demonstrated that long-running software systems exhibit gradual performance degradation that can be detected through continuous monitoring of operational performance metrics. Rather than relying solely on failure observations, their work showed that response time, throughput, and workload measurements provide early indicators of aging behavior, enabling proactive corrective actions before catastrophic failures occur.

Previous work [9], [10], [11], [12] demonstrated that software aging can be formulated as a performance assurance problem. Specifically, **the bucket algorithm** partitions recent response-time observations into a fixed number of sequential *buckets* to detect gradual software performance degradation caused by software aging. Rather than reacting to isolated response-time spikes, the algorithm identifies persistent upward trends in response time. When the performance degradation exceeds a predefined threshold relative to the system's performance objective, the algorithm triggers software rejuvenation. This bucket-based aggregation effectively filters transient workload fluctuations while remaining sensitive to sustained degradation, thus providing a tunable balance between false alarms and delayed rejuvenation. The algorithm is tuned by changing the bucket depth and the number of buckets used.

More recently, this line of research has been extended to heterogeneous virtualized environments, where monitoring-driven rejuvenation policies have been shown to improve resilience [5].

C. Software Aging in Cloud-Native Microservices

The adoption of microservice architectures and container orchestration platforms has significantly changed the operational characteristics of software systems. Applications are now composed of independently deployable services communicating through lightweight network protocols and executing within dynamically managed containers. Although container orchestration platforms facilitate rapid recovery through automated restarts and replica management, they do not eliminate software aging.

Recent studies have shown that long-running microservice applications continue to experience resource leaks, communication bottlenecks, memory growth, and performance degradation under sustained workloads [13], [14]. Unlike traditional distributed systems, aging effects in microservices may propagate across service dependencies, causing localized degradation to impact end-to-end application performance. Furthermore, replica management, load balancing, and communication optimizations, such as HTTP pooling, introduce additional factors that influence aging behavior and rejuvenation effectiveness.

Although the literature provides extensive theoretical foundations for software aging and rejuvenation, significant gaps remain in the context of modern cloud-native systems. There is a need to understand how deployment characteristics—including service replication, container-level rejuvena-

tion, and HTTP pooling—jointly influence aging behavior in microservice applications.

This paper builds upon the empirical software aging characterization developed in our previous work, while extending it to cloud-native microservices. Using the TrainTicket benchmark, we perform controlled 1-hour duration experiments across multiple deployment configurations and rejuvenation strategies.

III. METHOD

The adopted method is structured as a two-phase workflow, shown in Figure 1. In this work we study different adaptations of the Trainticket benchmark system, which is a microservice-based application that simulates a train ticket booking system. We use a fork developed by the Software Engineering Laboratory of Fudan University, which is available at [15]. To ensure reproducibility, the replication package [6] includes the transformations applied to the original Trainticket system to create the different system-under-test variants, as well as the workloads and scenarios used in the experiments.

In Phase 1, performance tests select a system-under-test configuration and a rejuvenation policy execution variant. The testing scenarios and load level (100 users) are fixed for all experiments (i.e., buying a ticket), while the system-under-test variants differ in the configuration (e.g., number of replicas, use of HTTP pooling or not) and the scenario execution variants differ in the rejuvenation policy approach used. In addition, for comparison, we also measured a no rejuvenation policy.

The testing framework executes the various experiments and collects measurements while the scenario execution interacts with the deployed system. In particular, the experiment observes response time and resource usage, records failures in buckets, keeps track of service degradation, and—for specific scenario execution variants—restarts aged containers. Each experiment was executed for one hour under the same workload profile.

The experimental design consists of two factors: deployment configuration and rejuvenation policy. In the software deployment configuration we selected the number of replicas assigned to the preserve microservice (one, two, four, eight), the HTTP pooling approach (pooling /no pooling), and all (that represents the assignment of two replicas to all microservices). In the software rejuvenation policy factor, we selected five policies: no rejuvenation; 3 buckets (bucket algorithm with three buckets); 2 buckets (bucket algorithm with two buckets); per-replica (an independent bucket system for each replica); and switching (the same independent per-replica bucket approach, with the load balancer redirecting traffic to another replica while the selected replica is being restarted).

IV. EMPIRICAL RESULTS

The following sections describe the empirical results obtained from the experiment executions.

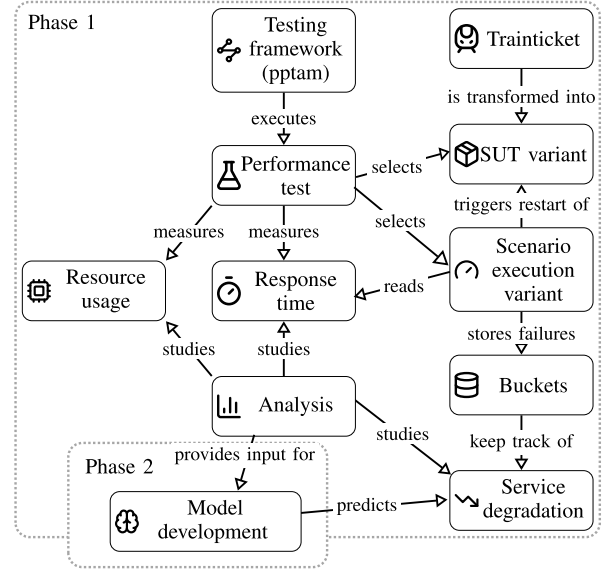


Fig. 1. Concept map of experiment execution, workload application, and runtime observation. Phase 2 is out of scope of this paper.

A. Summary of Empirical Results

Table I presents the detailed experimental design and the metrics derived from its execution. We evaluated Goodput/sec (GP/s) which is the throughput/sec minus failures/sec, the Standard Deviation of GP/s, Average Response Time in ms, Average Power defined as GP/s divided by Response Time in secs, and the Standard Deviation of the Power metric.

We evaluate the effectiveness of each software deployment configuration and software rejuvenation policy combination using the Average Power metric and its standard deviation, shown in Table I. High Power is associated with high Goodput and low response time, while lower standard deviation of Power indicates low variability of the Power metric.

The evaluated deployment configurations exhibit substantial differences in Goodput, response time, and Power values for the configuration/policies shown in Table I.

In the following, we illustrate the transient behavior of the reported metrics by presenting selected empirical result plots that represent the baseline, best case, and worst case. For compact paper figures, we used an event-aligned, peak-preserving temporal reduction: the time series was divided into fixed-size windows, retaining boundary samples and local extrema. We also preserved samples at relevant execution events, such as rejuvenation points and restart boundaries. This reduces visual density while preserving short peaks, drops, and event-related transitions. The plots showing the raw data are available in the replication package [6].

Each figure summarizes one variant of the (*preserve*) service over time. This service is the TrainTicket reservation service and has been identified as a TrainTicket performance bottleneck in related work [16].

The plots show CPU utilization and memory working set, aggregated request and failure rates, response-time behavior,

Configuration	Policy	Avg GP/s	Std GP/s	Avg RT (ms)	Avg Power	Std Power
pooling, 2 replicas (all)	3 buckets	16.216	7.023	3422.7	4.695	3.859
pooling, 2 replicas (all)	2 buckets	12.871	7.012	2736.0	4.646	4.728
pooling, 2 replicas (all)	per-replica ¹	14.637	6.016	1645.7	8.853	1.182
pooling, 2 replicas (all)	switching ²	14.361	5.608	1613.0	8.836	1.189
pooling, 2 replicas (all)	no rejuvenation	14.045	5.383	1725.2	7.915	1.408
8 replicas	3 buckets	5.855	7.342	9822.3	0.595	1.261
8 replicas	2 buckets	5.794	6.976	10785.3	0.557	1.108
8 replicas	per-replica	9.426	10.414	10346.3	0.909	0.308
8 replicas	switching	3.915	3.306	11386.8	0.344	0.099
8 replicas	no rejuvenation	5.305	6.585	9617.0	0.554	0.195
4 replicas	3 buckets	5.938	6.659	13608.3	0.438	2.127
4 replicas	2 buckets	4.935	5.336	23290.7	0.210	2.405
4 replicas	per-replica	5.376	6.590	12330.4	0.457	0.067
4 replicas	switching	3.456	2.679	11095.1	0.310	0.211
4 replicas	no rejuvenation	4.718	5.358	10962.6	0.444	0.067
1 replica	3 buckets	12.916	7.216	21881.8	0.580	5.624
1 replica	2 buckets	10.913	8.219	25894.8	0.407	5.721
1 replica	switching	11.227	7.588	26044.5	0.437	3.892
1 replica	no rejuvenation	4.804	3.711	19271.7	0.246	1.712
pooling, 8 replicas	3 buckets	11.776	4.424	3914.9	2.924	3.684
pooling, 8 replicas	2 buckets	12.056	4.471	3474.5	3.392	3.274
pooling, 8 replicas	per-replica	12.049	2.224	1899.3	6.209	1.065
pooling, 8 replicas	switching	11.988	2.177	1866.2	6.321	1.081
pooling, 8 replicas	no rejuvenation	13.210	4.437	1804.2	7.094	1.266
pooling, 4 replicas	3 buckets	12.470	4.873	3668.1	3.325	3.766
pooling, 4 replicas	2 buckets	12.140	4.939	4147.7	2.862	2.925
pooling, 4 replicas	per-replica	12.002	2.626	1730.2	6.829	1.258
pooling, 4 replicas	switching	12.003	2.077	1992.6	5.878	1.084
pooling, 4 replicas	no rejuvenation	12.244	2.772	1846.4	6.532	1.637
pooling, 1 replica	3 buckets	11.606	3.786	4271.4	2.637	3.511
pooling, 1 replica	2 buckets	11.977	5.392	5366.6	2.194	3.933
pooling, 1 replica	switching	12.114	2.075	1583.8	7.487	1.390
pooling, 1 replica	no rejuvenation	12.050	1.942	1754.1	6.726	1.593
pooling, 2 replicas	3 buckets	12.402	5.461	4095.6	2.956	2.787
pooling, 2 replicas	2 buckets	11.912	4.831	5907.6	1.995	3.350
pooling, 2 replicas	per-replica	11.936	2.458	1844.1	6.333	1.261
pooling, 2 replicas	switching	12.037	2.374	1855.8	6.280	1.130
pooling, 2 replicas	no rejuvenation	13.049	4.292	1945.1	6.591	1.516
2 replicas	3 buckets	10.742	8.874	28540.1	0.379	3.782
2 replicas	2 buckets	11.332	9.718	24779.7	0.453	4.676
2 replicas	per-replica	11.241	9.369	22428.4	0.502	0.084
2 replicas	switching	10.508	9.500	25626.8	0.408	0.110
2 replicas	no rejuvenation	7.945	8.217	20962.0	0.377	0.094

¹ Every replica has its own bucket system

² Every replica has its own bucket system and is switching replica before rejuvenating

TABLE I

AVERAGE GOODPUT, RESPONSE TIME, AND POWER WITH CONDITIONAL FORMATTING. DARKER GREEN INDICATES HIGHER AVERAGE GOODPUT, LOWER AVERAGE RESPONSE TIME, AND HIGHER AVERAGE POWER. DARKER RED INDICATES HIGHER STANDARD DEVIATION OF GOODPUT AND POWER.

Goodput and Power. For response time, the figures distinguish the last observed response time from the maximum response time since the most recent restart. In rejuvenation experiments, the figures additionally show bucket fill levels and mark restart or rejuvenation events with vertical dashed red lines.

B. No Pooling, Without Rejuvenation (Baseline)

Figure 2 illustrates the detailed metrics for the Baseline Train-Ticket configuration with one replica, no HTTP pooling and no rejuvenation. It shows progressive response-time degra-

ation, increasing memory consumption, and eventual service failure, with a very low Power value (0.246).

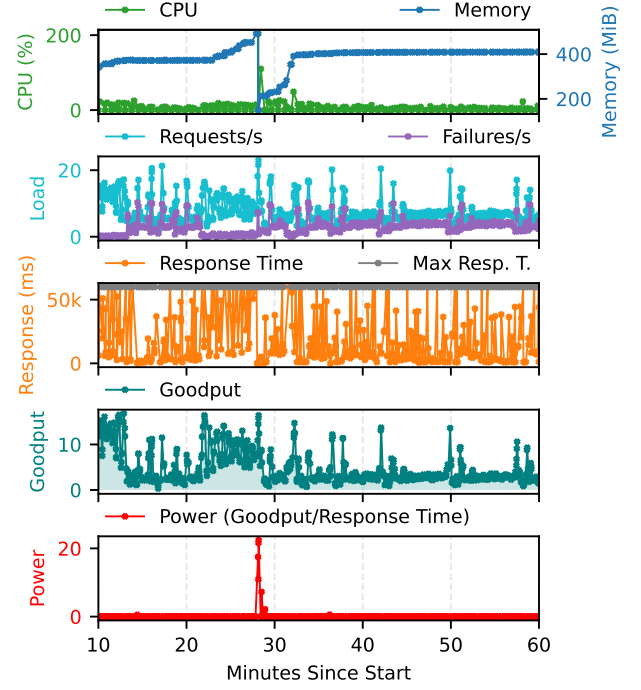


Fig. 2. No Pooling, Without Rejuvenation (Baseline)

C. With HTTP pooling, all services replicated with 2 replicas

In this subsection, we describe the empirical results obtained for the deployment configuration with HTTP pooling when all microservices are replicated with two replicas for the four rejuvenation policies and the no-rejuvenation evaluated. The top five rows in Table I contain the empirical results obtained from these experiments. In these experiments, we observed the highest value of the Power metric and the lowest standard deviation by using two strategies: pooling, two replicas (all) deployment with per-replica and switching software rejuvenation policies. For these two combinations we obtained the Power metric value of 8.8 and standard deviation of 1.18. In contrast, for the same deployment configuration and no rejuvenation we observed a small degradation in the Power metric (from 8.8 to 7.9) and higher variability in the Power metric standard deviation (from 1.18 to 1.4).

The detailed plots (Figures 3 – 7) illustrate the transient results for the five policy options evaluated, with HTTP pooling, all services replicated with 2 replicas.

D. No HTTP pooling, preserve service replicated with 2 repl.

In this subsection we present the empirical results for the deployment configuration with No HTTP pooling, when the preserve service is replicated with two replicas. These results correspond to the last five rows of Table I.

For these experiments, lower values of the Power metric (0.377-0.502) were observed when compared with the results

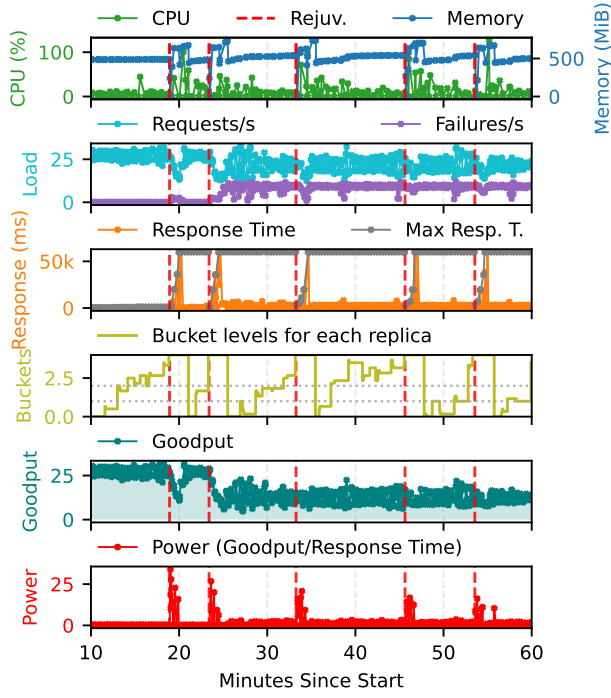


Fig. 3. With HTTP pooling, all services replicated with two replicas - Rejuvenation using Bucket algorithm (3 buckets)

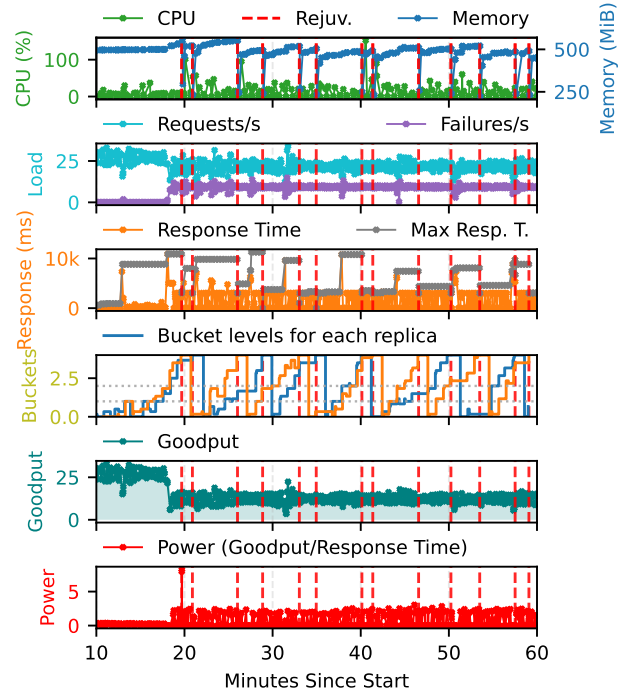


Fig. 5. With HTTP pooling, all services replicated with two replicas - Rejuvenation using Bucket algorithm, separate buckets per-replica

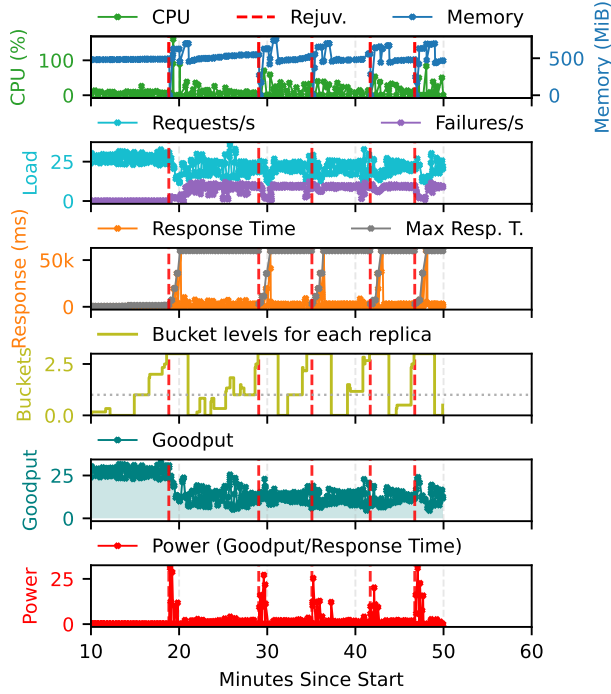


Fig. 4. With HTTP pooling, all services replicated with two replicas - Rejuvenation using Bucket algorithm, earlier threshold (2 buckets)

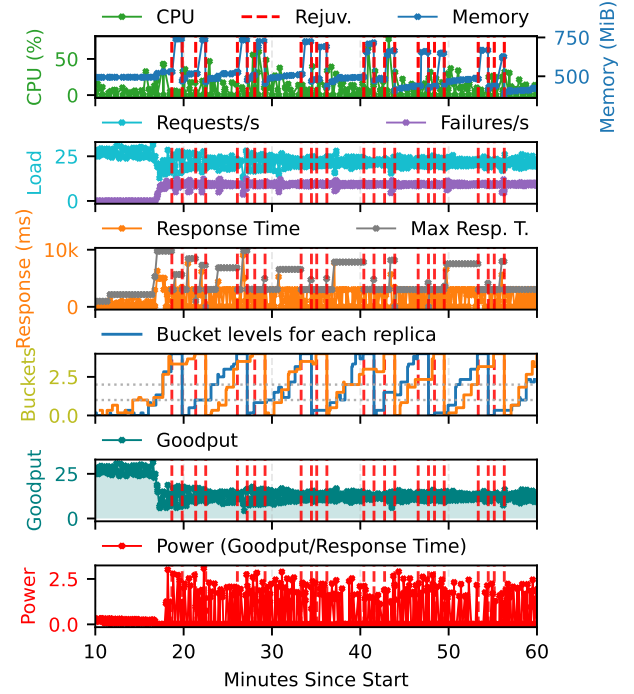


Fig. 6. With HTTP pooling, all services replicated with two replicas - Rejuvenation using Bucket algorithm, separate buckets per-replica, switching before rejuvenation

reported for the HTTP pooling variation, because of the large response times observed.

The detailed plots (Figures 8 – 11) illustrate four transient results for the software rejuvenation policy options evaluated in this section.

V. DISCUSSION

The empirical results demonstrate that software aging remains a significant concern in cloud-native microservice applications despite containerization and service replication. Across

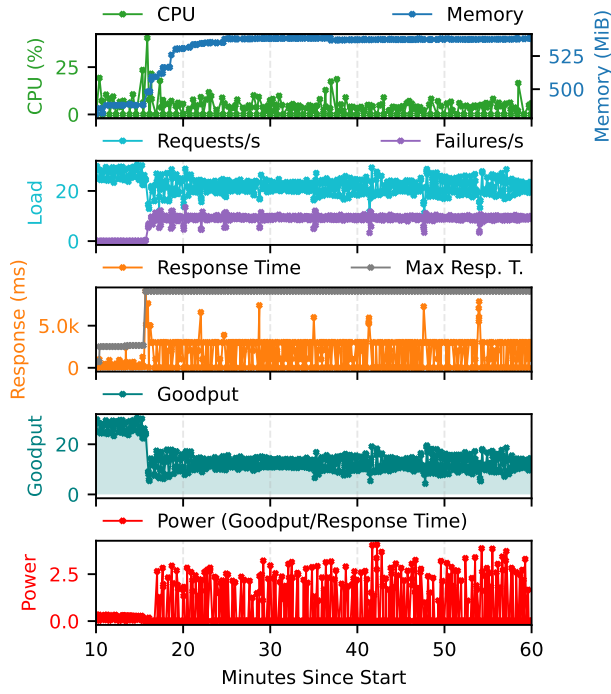


Fig. 7. With HTTP pooling, all services replicated with two replicas - No rejuvenation

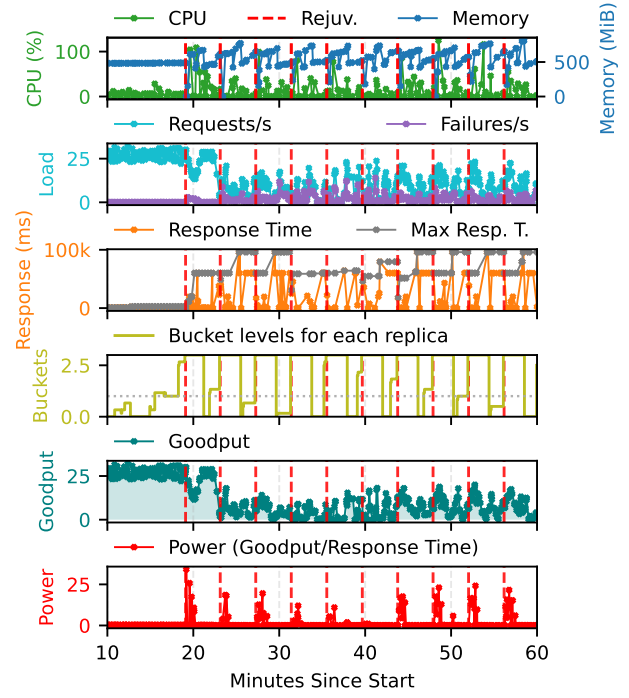


Fig. 9. No HTTP pooling, preserve service replicated with two replicas - Bucket algorithm, earlier threshold (2 buckets)

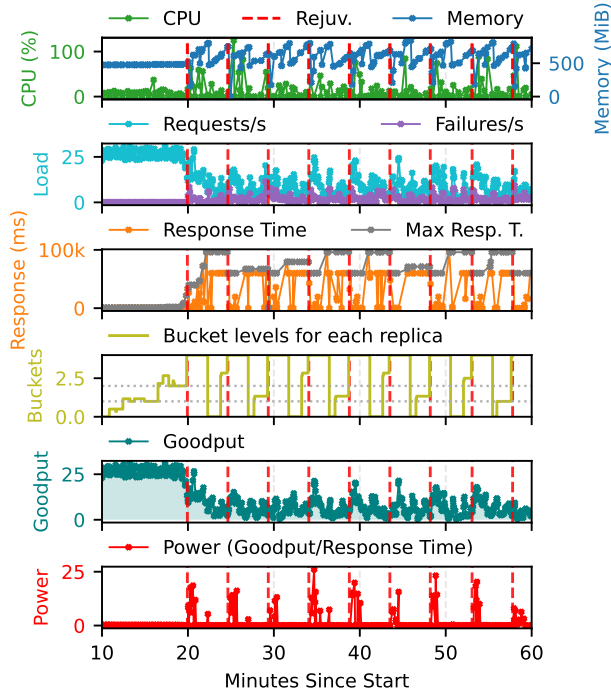


Fig. 8. No HTTP pooling, preserve service replicated with two replicas - Bucket algorithm (3 buckets)

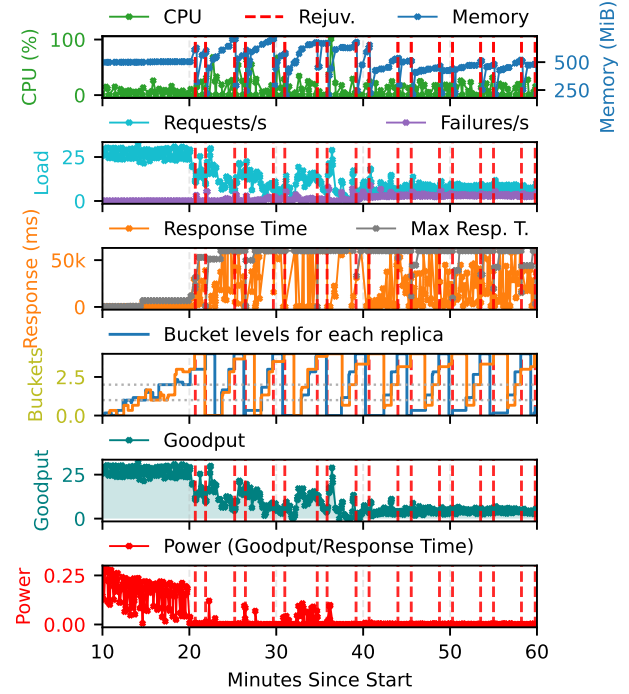


Fig. 10. No HTTP pooling, preserve service replicated with two replicas - Bucket algorithm, separate buckets per-replica

all experiments, sustained workloads produced progressive response-time degradation, increasing performance variability, and eventually service failures. Although the severity of aging depended on the deployment configuration, none of the eval-

uated architectural features completely eliminated the aging phenomenon.

Replication alone delays but does not prevent software aging. Without HTTP pooling, increasing the number of replicas postpones degradation but provides diminishing returns.

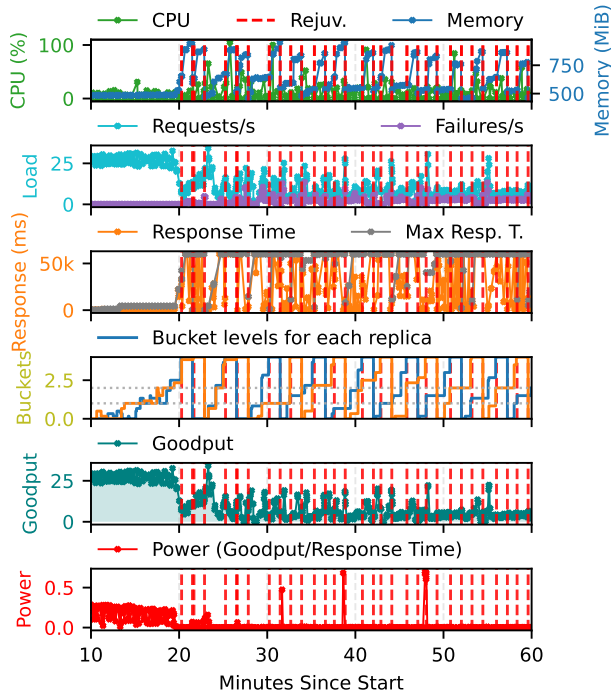


Fig. 11. No HTTP pooling, preserve service replicated with two replicas - Bucket algorithm, separate buckets per-replica, switching before rejuvenation

HTTP pooling had the largest positive impact on long-term performance. Across nearly all deployment configurations, pooling substantially reduced response time, increased Goodput, and improved the Power metric. The results suggest that communication overhead is an important contributor to performance degradation in microservice architectures and should be considered jointly with traditional resource-based aging mechanisms.

The rejuvenation policy also significantly influenced system behavior. Conventional bucket-based rejuvenation improved performance over no rejuvenation, while earlier thresholds (2 buckets) reduced response time by detecting degradation sooner. However, the most effective strategies were independent per-replica rejuvenation and switching, which localized recovery actions to aged replicas and avoided unnecessary restarts of healthy instances.

VI. CONCLUSION AND FUTURE WORK

This paper experimentally demonstrated that software aging remains an important concern in cloud-native microservice systems despite replication and containerization.

This empirical assessment illustrated the importance of evaluating software aging in a controlled empirical environment (1-hour tests with specific deployment configuration and software rejuvenation policy) using customer-oriented performance metrics that combine throughput and response time. The Power metric proved effective for comparing several deployment configurations and rejuvenation strategies.

The highest overall performance was obtained by combining HTTP pooling with per-replica rejuvenation, demonstrating

that deployment configuration and runtime rejuvenation should be designed together rather than optimized independently.

Future work will use the empirical observations reported in this paper to develop analytical models capable of predicting aging behavior and optimizing rejuvenation decisions. In addition, we have presented limited statistical analysis using only average and standard deviation. In the future we plan to run several repetitions of each experiment to compute confidence intervals.

REFERENCES

- [1] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: analysis, module and applications," *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pp. 381–390, 1995.
- [2] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *J. Emerg. Technol. Comput. Syst.*, vol. 10, no. 1, Jan. 2014.
- [3] M. Grottke and K. S. Trivedi, "Fighting bugs: remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.
- [4] A. Avritzer and E. J. Weyuker, "Monitoring smoothly degrading systems for increased dependability," *Empirical Softw. Engg.*, vol. 2, no. 1, p. 59–77, Jan. 1997.
- [5] A. Avritzer, A. Janes, A. Marin, C. Trubiani, A. van Hoorn, M. Camilli, D. S. Menasché, and A. B. Bondi, "Software aging detection and rejuvenation assessment in heterogeneous virtual networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 13, no. 2, pp. 299–313, 2025.
- [6] A. Janes, H. Rodrigues, A. Avritzer, L. Carnevali, L. Scommegna, and E. Vicario, "Replication package for the paper "empirical assessment of software aging and rejuvenation in microservices"," Jul. 2026. [Online]. Available: <https://doi.org/10.5281/zenodo.21342527>
- [7] L. Kleinrock, "Power and deterministic rules of thumb for probabilistic problems in computer communications," in *ICC 79: International Conference on Communications, Volume 3*. IEEE, 1979, pp. 4.3.1–4.3.8.
- [8] K. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and analysis of software aging and rejuvenation," in *Proceedings 33rd Annual Simulation Symposium (SS 2000)*, 2000, pp. 270–279.
- [9] A. Avritzer, R. G. Cole, and E. J. Weyuker, "Methods and opportunities for rejuvenation in aging distributed software systems," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1568–1578, 2010, software Dependability.
- [10] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker, "Performance assurance via software rejuvenation: Monitoring, statistics and algorithms," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '06. USA: IEEE Computer Society, 2006, p. 435–444.
- [11] A. Avritzer, A. Bondi, and E. J. Weyuker, "Ensuring stable performance for systems that degrade," in *Proceedings of the 5th International Workshop on Software and Performance*, ser. WOSP '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 43–51.
- [12] A. Avritzer and A. B. Bondi, "Dynamic tuning of a software rejuvenation method using a customer affecting performance metric," U.S. Patent 8,055,952 B2, Nov. 2011, assignee: Siemens Medical Solutions USA, Inc. Filed September 14, 2005. Issued November 8, 2011.
- [13] J. Bai, X. Chang, F. Machida, and K. S. Trivedi, "Understanding container-based services under software aging: Dependability and performance views," *IEEE Transactions on Sustainable Computing*, vol. 10, no. 3, pp. 562–575, 2025.
- [14] M. Santos, R. Matos, M. Vieira, and J. Araujo, "Software aging issues and rejuvenation strategies for a container orchestration system," *Future Generation Computer Systems*, vol. 179, p. 108274, 2026.
- [15] FudanSELab, "Train Ticket: A Benchmark Microservice System," 2022. [Online]. Available: <https://github.com/FudanSELab/train-ticket>
- [16] A. Avritzer, A. Janes, H. Rodrigues, Y. Cai, T. Schoyen, E. Pisch, C. Trubiani, A. B. Bondi, D. S. Menasché, and C. Zhang, "Automated assessment of the relationship between microservice architectures and performance," *Journal of Systems and Software*, vol. 237, p. 112857, 2026.