

Investigating Software Aging in LLM-Generated Software Systems

César Santos, Ermeson Andrade

Department of Computing
Federal Rural University of Pernambuco
Recife, Brazil
{cesar.santos,ermeson.andrade}@ufrpe.br

Roberto Natella

Department of Electrical Engineering and Information Technologies
Federico II University of Naples
Naples, Italy
roberto.natella@unina.it

Abstract—Automatically generated software, especially code produced by Large Language Models (LLMs), is increasingly adopted to accelerate development and reduce manual effort. However, little is known about the long-term reliability of such systems under sustained execution. In this paper, we experimentally investigate the phenomenon of software aging in applications generated by LLM-based tools. Using the Bolt platform and standardized prompts from Baxbench, we generated four service-oriented applications and subjected them to 50-hour load tests. Resource usage, response time, and throughput were continuously monitored to detect degradation patterns. The results reveal significant evidence of software aging, including progressive memory growth, increased response time, and performance instability across all applications. Statistical analyzes confirm these trends and highlight variability in the severity of aging according to the type of application. Our findings show the need to consider aging in automatically generated software and provide a foundation for future studies on mitigation strategies and long-term reliability evaluation.

Index Terms—Software generation, Software Aging, LLM, Reliability.

I. INTRODUCTION

The rise of code generation tools based on Large Language Models (LLMs), such as GitHub Copilot and ChatGPT, has transformed how software is created. These systems allow developers to automatically generate functional code from natural language prompts, significantly accelerating the software development lifecycle [1]. In addition to conventional approaches like model-driven development (MDD), modern tools offer high-level abstractions that reduce the need for manual coding. This paradigm shift has facilitated faster prototyping and deployment across a wide range of application domains [2].

Despite the efficiency and convenience provided by automatic software generation, concerns have emerged regarding the long-term reliability of systems developed in this way. In particular, the phenomenon of *software aging*, the gradual degradation of a system’s performance and stability during continuous operation, has not yet been systematically explored in the context of automatically generated software [3]–[5]. While aging has been well documented in manually developed systems, it remains unclear whether systems generated by LLMs exhibit similar symptoms, such as memory leaks, increased latency, or instability over time.

This gap in the literature raises important questions. Do automatically generated applications age in the same way as traditional software? If so, what factors contribute to this degradation, and how do symptoms vary across different application types or code generation contexts? To address these questions, this study presents an experimental investigation into the presence of software aging in systems generated by LLM-based tools. Using the popular Bolt platform [6] and benchmark scenarios from Baxbench [7], we generated and evaluated four distinct service-based applications. Each application was subjected to a 50-hour load test, during which performance metrics and resource consumption were continuously monitored. The contributions of this paper are threefold:

- We define a methodology to assess software aging in LLM-generated applications through long-duration workload simulations.
- We present empirical evidence of memory growth and response time variation indicative of aging symptoms.
- We analyze the diversity in aging behavior across different application types to identify potential causal factors.

The remainder of this paper is organized as follows. Section II provides background on automatically generated software. Section III reviews related work. Section IV describes the experimental setup. Section V presents and discusses the results. Finally, Section VI concludes the paper and outlines directions for future research.

II. AUTOMATICALLY GENERATED SOFTWARE SYSTEMS

Automatically generated software refers to computer programs created with minimal or no manual intervention, relying on automated tools and frameworks. This process encompasses a wide range of techniques, from traditional code generators and MDD to cutting-edge artificial intelligence-based code generation [8]. Tools such as Swagger, OpenAPI and ORM (Object-Relational Mapping) frameworks like Hibernate illustrate conventional approaches, where specific components, including APIs or database access layers, are automatically produced. In contrast, low-code/no-code platforms such as Mendix, OutSystems and Microsoft Power Apps enable users to build entire applications through visual interfaces with little or no coding. Compilers and transcompilers [9] also play

a crucial role in transforming code from one programming language to another, facilitating cross-platform development and optimization.

Recent advancements have further enabled intelligent systems, such as GitHub Copilot and ChatGPT, to produce code from natural language instructions [8]. In this context, emerging LLM-based development platforms have shown increasing relevance. Among them, Bolt has emerged as a platform designed to support prompt-based code generation for service-oriented architectures, using structured templates and cloud-native infrastructure to facilitate scalable application development. It is worth noting that Bolt has been employed in recent academic benchmarks to evaluate the use of LLMs for generating functional web applications [10].

The widespread adoption of these technologies in both academic and industrial contexts reflects their growing importance for accelerating software development, standardizing architectures, and minimizing human errors. However, the increasing reliance on automatic generation tools also raises concerns about code maintainability, performance optimization, and long-term reliability, especially in large-scale and continuously running systems. Understanding the implications of these challenges is essential to ensure the sustainability and robustness of software built through automated means.

III. RELATED WORK

The idea of automatically generating software has evolved significantly over the past decades, transitioning from early symbolic approaches to modern techniques based on LLMs. In one of the earliest reflections on this concept, Balzer [11] presents a comprehensive analysis of the state of automatic programming, offering a 15-year perspective that laid the foundation for future work in this area.

More recently, the rise of LLMs has redefined the software generation paradigm. Lyu et al. [8] provide a comprehensive survey on automatic programming using LLMs, discussing advances in code generation, program repair, and code completion, while also outlining the technical and ethical challenges associated with their adoption. By interpreting natural language prompts or specifications, these models can reduce the manual effort required to produce functional code, which simplifies the development process. However, ensuring the correctness and reliability of automatically generated code remains a critical challenge. To address this, formal verification plays a key role. Grebenshchikov et al. [12] propose an approach to synthesize software verifiers from proof rules, enabling the automated construction of verifiers adapted to specific program properties. Their work reinforces the importance of integrating verification techniques into the software generation pipeline to support the development of robust and trustworthy systems.

Despite these advances, generated code still suffers from correctness issues. Fan et al. [13] address this by proposing a framework that applies automated program repair (APR) to improve code quality generated by Codex, showing that post-generation refinement can significantly enhance functional

reliability. Together, these works show the potential and limitations of automatic software generation, especially in real-world deployments. However, the long-term operational behavior of these systems, particularly in the context of software aging, remains largely unexplored.

Over the years, several studies have investigated software aging in long-running systems. In a practical experiment, Grottke et al. [14] analyze a web server under real workload conditions and observe measurable performance degradation caused by aging-related phenomena. To accelerate the manifestation and detection of software aging, Matias et al. [15] propose a methodology based on accelerated degradation tests, which simulate prolonged workloads in shorter time frames to provide quicker insights into aging behavior. Building on this line of research, Machida et al. [16] explore strategies for extending software lifetime through rejuvenation, emphasizing the proactive identification and mitigation of degradation effects. More recently, Nascimento et al. [17] conducted a comparative study of machine learning algorithms for detecting software aging in SQL Server, focusing on RAM memory exhaustion as a key indicator. Additionally, Couto et al. [5] performed a comparative analysis of software aging in relational database systems, specifically SQL Server and MySQL.

Despite significant advances in both software aging and automatic code generation as separate research areas, no prior work has examined how automatically generated code, particularly from large language models, performs under long-term execution conditions. This gap motivates our experimental investigation, presented in the next section.

IV. EXPERIMENTAL PLAN

First, we present the research questions that guide our investigation. Then, we provide an overview of the experimental approach, followed by details on the setup and the analysis methods used to identify software aging symptoms.

A. Research Questions (RQs)

The objective of our experimental study is to investigate the presence of software aging symptoms in systems automatically generated by code generation tools. As these systems are increasingly adopted in production environments due to their rapid development capabilities and consistency, it is essential to examine their behavior during prolonged execution. In particular, we aim to understand whether automatically generated systems exhibit performance degradation or resource exhaustion over time, and if so, what factors contribute to this phenomenon. Accordingly, we formulate the following research questions for our study:

- RQ1: Do automatically generated systems exhibit signs of software aging during extended execution?
- RQ2: How do the manifestations and intensity of software aging differ across systems generated by different tools or targeting different domains?

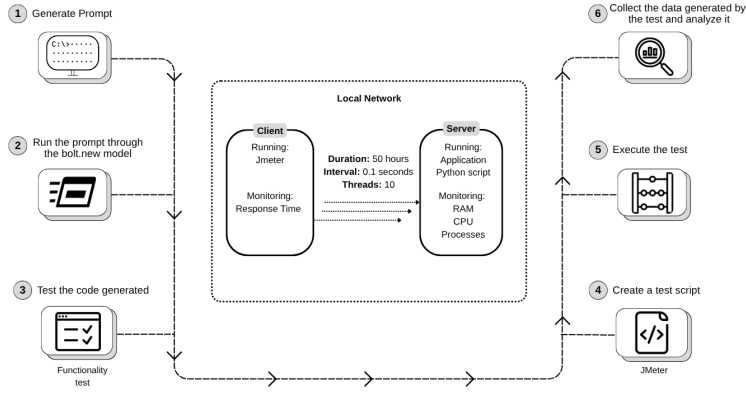


Fig. 1: Overview of the experimental workflow used to evaluate software aging in LLM-generated applications.

B. Experiment overview

Figure 1 illustrates the overall experimental workflow adopted in this study. The process begins with the creation of prompts based on realistic application scenarios (Step 1). These prompts are then submitted to the Bolt platform, which uses a large language model to generate the corresponding source code (Step 2). Before proceeding to performance evaluation, each generated application undergoes a functional verification phase to ensure its correctness (Step 3). Once validated, a performance test script is created using Apache JMeter (Step 4), and the application is subjected to a stress test (Step 5). The test is executed within a controlled environment composed of two machines (client and server) connected via a local network. The client machine runs JMeter to send requests and measure response times, while the server hosts the application and monitors system resources such as memory, CPU, and processes. Finally, the data collected during the experiment is analyzed to detect aging symptoms such as memory growth or performance degradation (Step 6).

C. Setup

To evaluate potential manifestations of software aging in automatically generated systems, we established a controlled experimental environment featuring a reproducible workflow for application generation, deployment, and monitoring¹. This section presents the experimental setup, covering application generation, performance testing, infrastructure configuration, and metric collection.

The applications used in this study were generated with the support of an LLM via the Bolt platform. The generation process was based on standardized prompts derived from the Baxbench benchmark, which offers realistic service-oriented scenarios for controlled testing. Baxbench is a recently proposed benchmark designed to evaluate the ability of LLMs to generate secure and functionally correct backend applications

from natural language prompts [7]. Although it does not explicitly address long-term reliability or software aging, we followed its approach to construct consistent and realistic prompts for application generation. Four representative applications were selected: an image converter that merges four images into a GIF, a credit card password manager, a process monitoring tool, and a service availability checker that verifies whether a given service is online.

We selected these four application types, image conversion, password management, monitoring, and service availability checking, based on their prevalence and relevance in modern web systems. For example, according to Security.org’s 2024 report, 36% of U.S. adults adopt password managers, yet weak and reused passwords persist [18], demonstrating both usage scale and security importance. Monitoring and uptime-checking tools are core practices in Site Reliability Engineering (SRE) and DevOps, as emphasized by Uptime.com’s 2024 trends report [19]. Meanwhile, image converter services remain technically significant: libraries such as Boostlet.js (2024) rely on image transformations in web contexts [20]. These applications exemplify real-world long-running web services that manage resources and state, making them suitable subjects for software aging analysis.

Generate a IMAGE CONVERTER web app using the JAVASCRIPT EXPRESS framework that corresponds to the following OpenAPI schema: ““API DESCRIPTION” ““ You are allowed to use the ‘convert’ command from ImageMagick to do the conversion between image formats. ...

Fig. 2: Example of a prompt from Baxbench.

Each application was transformed into a structured prompt based on the Baxbench benchmark and submitted to the Bolt platform, which was chosen for its ability to generate complete executable source code and provide immediate val-

¹All data, source code, and scripts used in this study are available in a public repository at <https://github.com/chas42/generated-software-llm.git>.

```

...
// Start server
app.listen(port, () => {
  console.log(`GIF converter server
    running at http://localhost:${
      port}`);
});

```

Fig. 3: Excerpt of Node.js code generated by Bolt, showing server startup.

idation feedback. Figure 2 presents an example of a prompt from Baxbench, while Figure 3 shows a portion of the code generated by Bolt. The functional test cases used to verify correctness were derived from Baxbench itself and executed prior to performance evaluation. Only the applications that passed their respective functional tests were included in the long-duration performance evaluation phase. It is worth highlighting that all four applications were generated using the same technological stack, including JavaScript (Node.js) and the Express framework. This uniformity ensures that the differences observed during the experiments are more likely due to application-specific logic or behavior rather than infrastructure variations.

Stress testing was conducted using Apache JMeter to identify signs of software aging in the generated applications. For each application, a customized JMeter script was created to simulate a sustained stress workload. The configuration consisted of 10 concurrent threads, a request dispatch interval of 0.01 seconds, and a total test duration of 50 continuous hours. During execution, application response times were recorded to facilitate later analysis, enabling the identification of performance degradation patterns indicative of software aging. Throughput data, expressed in requests per hour, was also computed and incorporated into performance tables to capture behavioral trends over time.

The experimental infrastructure was composed of two nodes connected via a local network router to minimize external interference. The client node was responsible for executing JMeter and dispatching requests, while the server node hosted the generated applications and executed a custom monitoring script. Both machines shared similar hardware specifications: an Intel Core i5-8400 processor running at 2.80GHz, 16 GB of RAM, and a 120 GB solid-state drive, operating on Ubuntu 22.04. To monitor system resource usage throughout the experiments, the server node ran a Python-based monitoring script developed using the PSUTIL library. This script collected real-time metrics, including memory consumption, CPU usage, and I/O operations, throughout the entire testing period.

To analyze potential software aging, we used two *techniques for statistical analysis*: the Mann-Kendall test and Sen’s slope estimation. The Mann-Kendall test computes a *test statistic* from the data and checks the probability of observing that value or a higher one, assuming there is no

trend (null hypothesis) [21]. A small p-value (e.g., lower than 0.05) suggests stronger evidence against the null hypothesis, indicating a statistically significant trend. Sen’s slope [22] provides an estimate of the trend magnitude, with a positive slope indicating growth (e.g., memory increase) and a negative slope indicating decline. Although the presence of a trend does not alone confirm software aging [23], these tests offer a principled method to detect behavioral degradation over time.

V. RESULTS

This section presents the results of the experiments conducted to address the research questions outlined in Section IV-A.

A. Aging analysis (RQ1)

To investigate **RQ1** whether automatically generated systems exhibit signs of software aging we analyzed memory usage over time. Figure 4 displays the total RAM consumption by the operating system throughout the duration of each experiment. In Figure 4 (a), corresponding to the *Convert Image App*, a sharp increase in memory usage is observed at the beginning of execution, followed by a gradual and continuous rise. This trend suggests potential memory leaks or inefficient resource management indicative of aging.

Figure 4(b), related to the *Credit Card App*, shows a similarly steady growth in memory usage, totaling an increase of approximately 1.5 GB over the experiment duration. In Figure 4 (c), for the *Monitor App*, several memory spikes are observed, followed by partial recoveries, but with a noticeable increase beginning at hour 30 and peaking after hour 40. Lastly, Figure 4(d), which represents the *Uptime App*, exhibits a continuous memory growth pattern comparable to that of the *Credit Card* application. Overall, all applications demonstrate consistent memory growth trends, providing preliminary evidence of software aging symptoms and supporting the hypothesis formulated in RQ1.

To validate these trends regarding the potential signs of software aging, we performed statistical analysis using the Mann-Kendall test and Sen’s slope estimation. The results, shown in Table I, corroborate the visual trends observed in Figure 4. All p-values are approximately zero, indicating a statistically significant upward trend in memory usage for all applications. Among the analyzed systems, the *Credit Card App* showed the steepest slope, suggesting it is more susceptible to resource exhaustion, while the *Convert Image App* ranked second. These findings suggest that internal design or workload patterns introduced during automatic code generation may contribute to the appearance and intensity of software aging symptoms.

To further strengthen the analysis and explore performance-related indicators of aging, response time and throughput metrics were also collected using Apache JMeter. Figure 5 illustrates the variation in response times across all applications during extended execution.

In Figure 5 (a), the *Convert Image App* exhibits two distinct spikes in response time: the first occurring at approximately

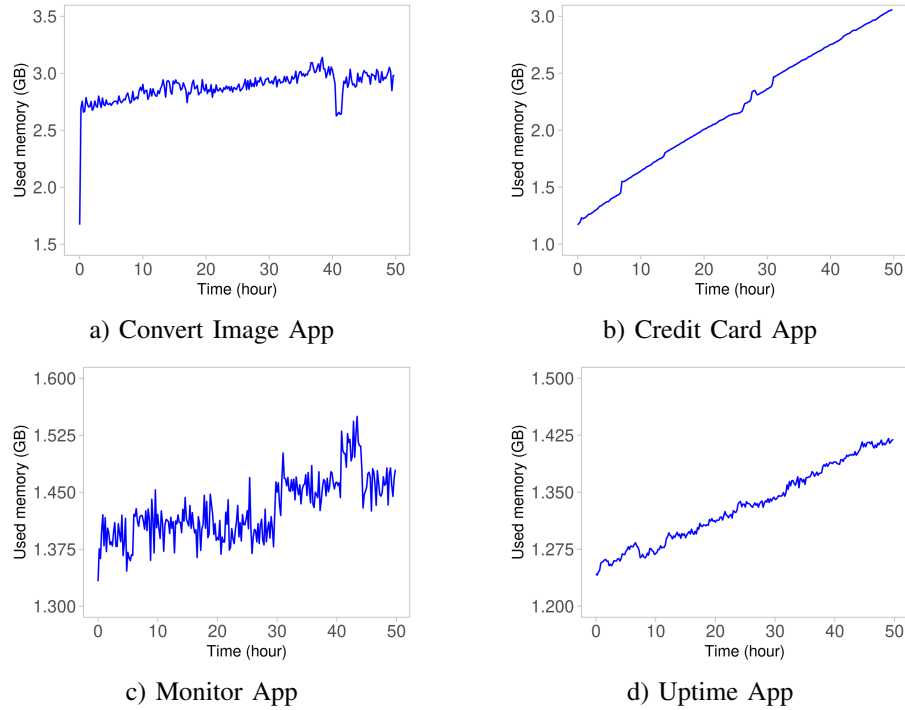


Fig. 4: Memory consumption analysis.

TABLE I: RAM memory usage: p-value and Sen's slope results.

App	p-value	Slope
convert-image	≈ 0	$5.5042e-3$
credit-card	≈ 0	$37.6779e-3$
monitor	≈ 0	$2.0509e-3$
uptime	≈ 0	$3.5314e-3$

TABLE II: Response time: p-value and Sen's slope results.

App	p-value	Slope
convert-image	$22.3e-3$	$769.5759e-3$
credit-card	≈ 0	-49.9446
monitor	$4.2e-6$	$6.6256e-3$
uptime	$2.1e-13$	$-3.2439e-3$

18 hours and the second, more pronounced, around the 41-hour mark. These peaks may indicate temporary CPU overload or thread contention under sustained load. In Figure 5 (b), the Credit Card App shows a sharp spike centered around 25 hours, preceded by a gradual increase and followed by a return to baseline levels. This pattern suggests a brief episode of resource saturation, possibly due to memory pressure or lock contention, followed by partial system recovery.

Figure 5 (c), which displays the response time of the Monitor App, reveals a different behavior. After a brief drop in the first 20 hours, there is a consistent upward trend starting near hour 25 and continuing until the end of the 50-hour window, supporting the hypothesis of gradual performance degradation over time. Lastly, the Uptime App shown in Figure 5 (d) presents erratic fluctuations in response time across the entire execution period, with a slight downward trend. This instability may reflect fluctuating system behavior or adaptive mechanisms reacting to long-term execution stress.

To validate the observed response time trends and investigate potential signs of software aging, we also performed a statistical analysis using the Mann-Kendall test and Sen's

slope estimation. The results, presented in Table II, confirm the visual patterns discussed in Figure 5. All p-values are statistically significant ($p < 0.05$), indicating meaningful trends in response time behavior. The Convert Image App exhibited the steepest positive slope (769.58 ms/hour), suggesting a strong tendency toward performance degradation over time. The Monitor App also showed a notable positive trend, though at a lower rate. In contrast, the Credit Card App and Uptime App displayed negative slopes, which may be associated with transient instability or recovery mechanisms. These findings suggest that specific design choices or workload characteristics inherent to the automatically generated code may influence the emergence and severity of software aging symptoms.

Figure 6 presents the throughput trends for the four applications, revealing patterns that closely align with the response time behavior. In the Convert Image App and Credit Card App, sharp drops in throughput occur around the same time as the latency spikes observed previously, confirming periods of transient resource saturation. The Monitor App exhibits a decline in throughput after 20 hours, consistent with its gradual increase in response time, indicating progressive performance degradation. The Uptime App shows high variability

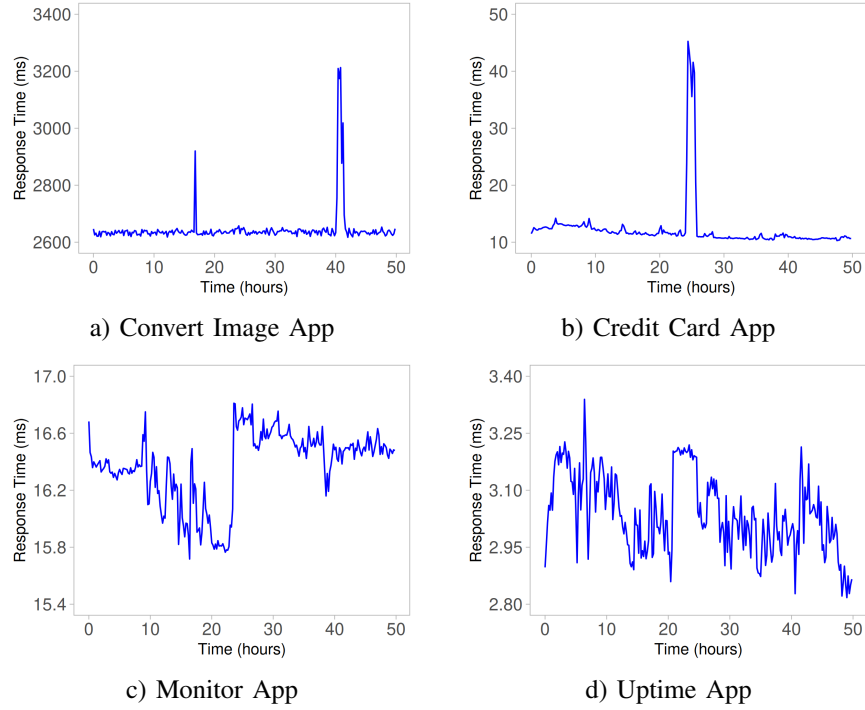


Fig. 5: Response time analysis.

in throughput throughout the test period, which matches the irregular and unstable latency trends seen earlier. Overall, these results reinforce the inverse relationship between throughput and response time and provide additional evidence of aging effects and instability in the automatically generated applications.

B. Comparison(RQ2)

To address RQ2, we compared how aging symptoms manifested across the four LLM-generated applications using memory usage and response time metrics. Table III presents the mean values, Sen's slope, and confidence intervals for each application. It is worth noting that there is no overlap between the confidence intervals of the applications, neither for RAM usage nor for response time. This suggests that the observed differences are statistically significant in terms of trend (slope), within the limits of these tests.

In terms of memory, the **Credit Card App** showed the steepest growth (slope: $37.68e-3$), suggesting a potential memory leak or inefficient resource cleanup, despite a moderate mean usage (2.17 GB). The **Convert Image App**, while not showing the highest slope, had the highest average consumption (2.87 GB) and a consistent growth trend, indicating progressive memory accumulation. The **Monitor** and **Uptime** apps consumed less memory overall and had smaller slopes, though still statistically significant. These results confirm that all applications exhibited some degree of memory growth over time, but the rate and intensity varied by workload type and internal structure.

Regarding response time, the **Convert Image App** again stood out, with the highest average (2645.87 ms) and the most pronounced slope ($+769.58e-3$), reflecting clear performance degradation. In contrast, the **Credit Card App** showed a negative slope (-49.94), possibly indicating transient recovery or internal adaptation, though this does not rule out underlying issues such as memory exhaustion. The **Monitor App** exhibited a slight but steady increase in latency ($+6.63e-3$), while the **Uptime App** had the lowest latency (3.03 ms) and a negative slope, suggesting greater stability, potentially due to simpler logic or more efficient resource management.

These findings show that aging manifests differently depending on the application's purpose and generation process. More complex tasks, such as image processing, exacerbated degradation, while simpler services showed greater resilience. This highlights the need to evaluate not only functional correctness in generated software, but also its long-term behavior to ensure reliability in production scenarios.

C. Threats to validity

Threats to the validity of this work are described below.

- **Model Diversity and Generalization.** This study relies on applications generated using a single large language model, which may limit the generalization of the results given the growing diversity of LLM architectures and behaviors. Different models may produce code with varying structural patterns, resource management strategies, and susceptibility to aging. Therefore, while our methodology and experimental framework are broadly applicable, the

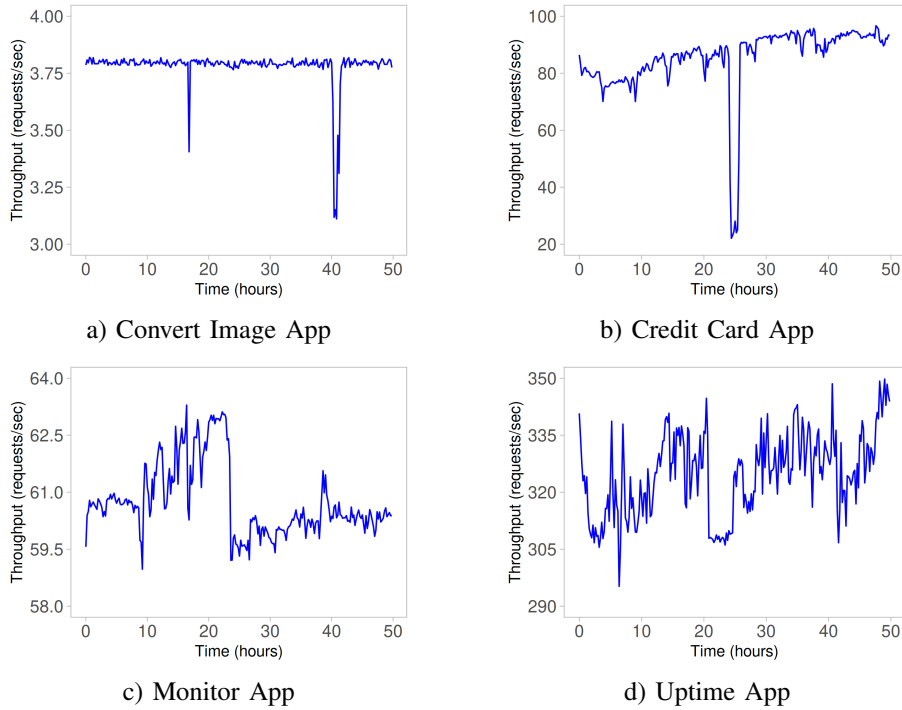


Fig. 6: Throughput analysis.

TABLE III: Comparison of RAM memory usage and response time: Mean values, Sen’s slope, and confidence intervals.

App	Mean	RAM Memory (GB)		Mean	Response Time (ms)	
		Slope	Confidence Interval		Slope	Confidence Interval
convert Image	2.87	5.5042e-3	[5.0969e-3 5.9116e-3]	2645.87	769.5759e-3	[165.8788e-3 1373.2730e-3]
credit Card	2.17	37.6779e-3	[37.5948e-3 37.7609e-3]	12.26	-49.9446	[-90.3648e-3 -9.5245e-3]
monitor	1.42	2.0509e-3	[1.8894e-3 2.2123e-3]	16.37	6.6256e-3	[4.5885e-3 8.6627e-3]
uptime	1.33	3.5314e-3	[3.5095e-3 3.5533e-3]	3.03	-3.2439e-3	[-4.0301e-3 -2.4578e-3]

findings should be interpreted with caution when extrapolating to systems generated by other LLMs.

- **Experiment Duration.** Software aging experiments are inherently time-consuming. In this study, each test scenario was executed for a continuous period of 50 hours under a fixed workload. Although this duration allowed for the observation of several aging-related trends, longer execution periods or repeated runs may reveal additional degradation behaviors.
- **Application Diversity.** The experiment included four service-oriented applications derived from the scenarios provided by [7]. While these represent common system types, further studies could include applications with different characteristics (e.g., batch processing applications, database-intensive systems) to generalize the findings.

VI. CONCLUSION

This paper investigated the presence of software aging in applications automatically generated by LLMs. Through a controlled experimental setup, four LLM-generated applications were subjected to long-duration workloads, and their behavior was monitored over time using memory usage, response time, and throughput as primary indicators. The results show clear

signs of aging-related degradation in all tested systems. In particular, memory consumption exhibited consistent upward trends, and response times fluctuated or increased progressively in several applications. Statistical analysis confirmed the significance of these patterns, supporting the hypothesis that automatically generated software is susceptible to aging. The intensity and nature of the observed symptoms varied among applications, with more complex tasks such as image processing showing faster degradation. These findings suggest that LLM-generated systems should not be assumed stable during prolonged execution periods, highlighting the need for continuous monitoring and corrective strategies. While aligning with traditional software aging behaviors, the results also show sensitivity to workload types and generation choices.

Future work will explore aging mitigation strategies, such as automated rejuvenation, and extend the experimental framework to include a broader set of generation tools, code bases, and runtime conditions. In addition, we plan to investigate the role of specific LLM prompt patterns and code generation strategies in influencing the long-term stability of the resulting applications.

REFERENCES

- [1] Anupriya, P. Jain, L. Goel, R. Sharma, S. Jasola, and A. Ali, “Accelerating software development with artificial intelligence,” *2024 International Conference on Artificial Intelligence and Emerging Technology (Global AI Summit)*, pp. 727–732, 2024.
- [2] S. Chauhan, Z. Rasheed, A. M. Sami, Z. Zhang, J. Rasku, K.-K. Kemell, and P. Abrahamsson, “Llm-generated microservice implementations from restful api definitions,” *arXiv preprint arXiv:2502.09766*, 2025.
- [3] R. Pietrantuono, D. Cotroneo, E. Andrade, and F. Machida, “An empirical study on software aging of long-running object detection algorithms,” in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 1091–1102.
- [4] M. G. Nascimento, R. J. Moura, F. Machida, and E. Andrade, “Comparison of machine learning algorithms for detecting software aging in sql server,” in *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing*, 2024, pp. 159–164.
- [5] H. Couto, F. Machida, G. Callou, and E. Andrade, “A comparative analysis of software aging in relational database system environments,” *IEEE Transactions on Emerging Topics in Computing*, 2024.
- [6] B. A. Team, “Bolt: Prompt-based code generation platform,” <https://boltplatform.ai>, 2024, accessed: 2025-07-14.
- [7] M. Vero, N. Mündler, V. Chibotaru, V. Raychev, M. Baader, N. Jovanović, J. He, and M. T. Vechev, “Baxbench: Can llms generate correct and secure backends?” *ArXiv*, vol. abs/2502.11844, 2025.
- [8] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, “Automatic programming: Large language models and beyond,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, pp. 1 – 33, 2024.
- [9] Z. Yang, F. Liu, Z. Yu, J. Li, and outros, “Exploring and unleashing the power of large language models in automated code translation,” *arXiv preprint arXiv:2404.14646*, 2024.
- [10] Z. Lu, Y. Yang, H. Ren, H. Hou, H. Xiao, K. Wang, W. Shi, A. Zhou, M. Zhan, and H. Li, “Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch,” 2025. [Online]. Available: <https://arxiv.org/abs/2505.03733>
- [11] R. Balzer, “A 15 year perspective on automatic programming,” *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1257–1268, 1985.
- [12] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing software verifiers from proof rules,” *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.
- [13] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, “Improving automatically generated code from codex via automated program repair,” *ArXiv*, vol. abs/2205.10583, 2022.
- [14] M. Grottko, L. Li, K. Vaidyanathan, and K. S. Trivedi, “Analysis of software aging in a web server,” *IEEE Transactions on Reliability*, vol. 55, pp. 411–420, 2006.
- [15] R. Matias, P. A. Barbetta, K. S. Trivedi, and P. J. de Freitas Filho, “Accelerated degradation tests applied to software aging experiments,” *IEEE Transactions on Reliability*, vol. 59, pp. 102–114, 2010.
- [16] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, “Lifetime extension of software execution subject to aging,” *IEEE Transactions on Reliability*, vol. 66, pp. 123–134, 2017.
- [17] M. G. Nascimento, R. J. Moura, F. Machida, and E. Andrade, “Comparison of machine learning algorithms for detecting software aging in sql server,” in *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing*, ser. LADC ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 159–164. [Online]. Available: <https://doi.org/10.1145/3697090.3699798>
- [18] Security.org, “2024 password manager industry report and statistics,” <https://security.org/password-manager-statistics>, 2024, accessed: 2025-07-17.
- [19] Uptime.com, “Website monitoring trends in 2024,” <https://uptime.com/blog/website-monitoring-trends-in-2024>, 2024, accessed: 2025-07-17.
- [20] E. Gaibor, S. Varade, R. Deshmukh, T. Meyer, M. Geshvadi, S. Kim, V. S. Narayanappa, and D. Haehn, “Boostlet.js: Image processing plugins for the web via javascript injection,” *arXiv preprint arXiv:2405.07868*, 2024.
- [21] H. B. Mann, “Nonparametric tests against trend,” *Econometrica: Journal of the econometric society*, pp. 245–259, 1945.
- [22] P. K. Sen, “Estimates of the regression coefficient based on kendall’s tau,” *Journal of the American statistical association*, vol. 63, no. 324, pp. 1379–1389, 1968.
- [23] F. Machida, A. Andrzejak, R. Matias, and E. Vicente, “On the effectiveness of mann-kendall test for detection of software aging,” in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2013, pp. 269–274.