A Case Study on Software Aging in LLM-Generated Python Applications

Gustavo Costa[1], **Cesar Santos**[1], **Roberto Natella**[2], **Ermeson Andrade**[1]

[1] Universidade Federal Rural de Pernambuco (UFRPE) – Recife – PE – Brasil

[2]Gran Sasso Science Institute (GSSI) – L'Aquila – Italy

{gustavo.hscosta, cesar.santos, ermeson.andrade}@ufrpe.br

roberto.natella@gssi.it

Abstract. *Large Language Models (LLMs) have increasingly been used to automatically generate software systems. However, the long-term operational behavior of these applications remains poorly understood. In particular, it is still unclear whether LLM-generated systems may exhibit progressive performance degradation and increasing resource consumption over time, characterizing symptoms of software aging during prolonged execution. This work presents an experimental case study investigating possible aging effects in four Python applications generated with ChatGPT based on scenarios from the BaxBench benchmark. The systems were executed under continuous workload while memory consumption and response time were monitored. Statistical analyses using the Mann-Kendall test and Sen's slope estimator were applied to detect temporal trends in the collected data. The results indicate consistent growth in memory consumption across the evaluated applications, suggesting evidence of aging, while response time degradation was observed in only one case.*

1. Introduction

The use of LLMs in automatic code generation has gained prominence in software development. According to Jiang et al. (2026), these technologies make it possible to transform natural language instructions into executable code, increasing the level of abstraction in the development process and making application construction more agile. In this context, automatically generated software can be understood as systems whose implementation is produced wholly or partially by mechanisms capable of converting high-level descriptions into executable artifacts. As a result, the developer no longer acts only as the direct author of each instruction and also begins to guide generation through previously defined commands, constraints, and objectives (YETIŞTIREN et al., 2023).

This change alters the traditional dynamics of programming, since part of the effort previously concentrated on manual writing is shifted to the formulation of the instructions that guide code generation. Consequently, the implementation of functionalities can occur more quickly, especially in repetitive tasks or in the early stages of application development. However, the practical usefulness of these systems depends not only on the production of executable code, but also on the quality of the generated result, including attributes such as robustness, maintainability, efficiency, and operational stability (SUN et al., 2025).

Although these automatic code generation tools bring relevant productivity gains, their adoption does not eliminate concerns related to the reliability of the systems produced. Recent studies show that code generated by LLMs may present structural inefficiencies, vulnerabilities, and inconsistencies that affect its runtime behavior (ABBASSI et al., 2025). In addition, there are cases in which the generated output appears structurally coherent, but incorporates solutions that are inadequate for the requested problem or limitations that compromise the non-functional quality of the application (LIU et al., 2026). In web applications and service-oriented systems, these weaknesses become particularly relevant because they may affect system operation when subjected to real operating conditions (DORA et al., 2025).

In this scenario, software aging emerges as an especially important topic for the analysis of automatically generated applications. According to Grottke, Jr. e Trivedi (2008), software aging corresponds to the process of progressive degradation in the performance and reliability of a system during its continuous operation. Unlike the physical wear observed in hardware, this phenomenon results from the accumulation of undesirable internal states throughout execution, such as memory leaks, resource fragmentation, accumulation of numerical errors, and inadequate release of internal structures (TRIVEDI; VAIDYANATHAN, 2007). Because this degradation tends to arise gradually, its identification is not always immediate, which makes it necessary to observe system behavior over time (GARG et al., 1998).

Among the most frequent symptoms of software aging are increased response time, growth in memory consumption, reduced processing throughput, and the occurrence of intermittent failures. Such signs have already been observed experimentally in web servers and in other computing environments that remain active for long periods, demonstrating that software aging is not merely a theoretical concept, but a practical problem with direct impact on the operation of real applications (JR.; FILHO, 2006). In web and service-oriented systems, this discussion becomes even more relevant because these environments normally execute continuously and are subject to variable loads, which favors the appearance of performance anomalies associated with this phenomenon (GROTTKE et al., 2006).

Despite advances in research on automatic code generation, most recent studies have focused on aspects such as functional correctness, security, efficiency, and the quality of the produced code (JIANG et al., 2026; SUN et al., 2025). Investigations that specifically analyze whether applications generated by LLMs are also subject to symptoms of software aging during prolonged executions remain scarce. This absence of evidence constitutes a relevant gap, since it is not clear whether automatically produced systems exhibit the same patterns of deterioration observed in traditionally developed software.

Given this gap, this work presents an experimental case study investigating the presence of signs of software aging in applications automatically generated by LLMs using ChatGPT. To this end, four service-oriented applications were developed in Python based on scenarios defined in the BaxBench benchmark (VERO et al., 2025), a set of tasks designed to evaluate automatically generated systems. These applications were then subjected to prolonged load tests, with continuous monitoring of performance and resource consumption metrics, making it possible to analyze the evolution of operational behavior over time. The results reveal consistent growth in memory consumption, indicating pos-

sible symptoms of aging, while degradation in response time was observed in a limited manner.

The remainder of this article is organized as follows. Section 2 presents studies on software aging and automatic code generation by LLMs. Section 3 describes the experimental methodology adopted, including the environment, the evaluated applications, and the process used to execute the experiments. Section 4 presents and discusses the obtained results. Section 5 discusses the threats to the validity of the study. Finally, Section 6 presents the conclusions and possible directions for future work.

2. Related Work

Studies on software aging provide an important basis for understanding how continuously running systems may exhibit progressive degradation in performance and reliability. Among the classic works in the area, Garg et al. (1998) propose a methodology to detect and estimate signs of aging from the behavior observed during operation. Complementarily, Grottke, Jr. e Trivedi (2008) discuss the fundamentals of software aging, presenting its main concepts, causes, and manifestations in computing systems.

In addition to the conceptual foundation, part of the literature focuses on the experimental investigation of this phenomenon in real environments. Grottke et al. (2006) analyze aging in web servers and show that degradation can directly affect performance metrics and operational stability. In the same direction, Jr. e Filho (2006) conducted an experimental study on web servers, observing that prolonged executions may cause gradual deterioration in system behavior. In turn, Araujo et al. (2011) evaluated this phenomenon in a cloud computing infrastructure, indicating that the problem also manifests itself in more complex and distributed environments. Other works advance the proposal of mechanisms for monitoring and detecting aging. Chen, Qi e Hou (2015), for example, present an entropy-based approach to detect failures associated with aging in real time. More recently, Silva et al. (2025) investigated adaptive aging detection under changes in workload, showing that the operational dynamics of the system influence the identification of performance anomalies.

In parallel, recent literature has also devoted increasing attention to software automatically generated by LLMs. Jiang et al. (2026) present a broad survey on the use of LLMs for code generation, highlighting both the growth of the area and the technical challenges that still exist. From an empirical perspective, Yetiştiren et al. (2023) analyze the quality of the code produced by AI-assisted generation tools and show that, although these technologies bring productivity gains, there are still relevant limitations in the generated output. Some of these limitations appear in works focused on security and reliability. Khoury et al. (2023) investigate the security of automatically generated code and observe that this process may introduce important vulnerabilities. Similarly, Dora et al. (2025) evaluate web applications generated by language models and identify structural risks that may compromise their reliability in real usage scenarios.

Beyond security problems, other authors draw attention to aspects related to efficiency and non-functional quality. Abbassi et al. (2025) propose a taxonomy of inefficiencies in Python code generated by LLMs, showing that automatic generation may result in implementations with inadequate use of resources. Liu et al. (2026), in turn, discuss the occurrence of hallucinations in code generated by language models, showing

that the apparent coherence of the output does not always correspond to a correct or adequate solution. Sun et al. (2025) emphasize the importance of evaluating non-functional attributes, such as efficiency, robustness, and stability, in the context of quality assurance for this type of software. More closely related to the context of this research, Santos, Andrade e Natella (2025) conduct an experimental study with service-oriented applications generated automatically and subjected to prolonged load tests. However, this study focuses on applications developed in JavaScript, which highlights the need for complementary investigations in other ecosystems with distinct execution and resource management characteristics.

Despite these contributions, studies on software aging and studies on automatic code generation still appear, to a large extent, in a dissociated way. While the classical literature examines progressive degradation in traditional systems, more recent work on LLMs focuses mainly on security, correctness, and the quality of the produced code. Thus, the investigation of possible signs of aging in applications produced by LLMs remains a gap, especially in languages such as Python, whose memory management and execution mechanisms differ from those of other environments already explored. It is in this space that the present work is situated, seeking to experimentally analyze this behavior in automatically generated systems.

3. Experiment

This section describes the experimental methodology adopted in this case study to investigate the presence of signs of software aging in applications automatically generated by LLMs. The approach combines automatic code generation, functional validation, execution under controlled load, and long-duration experiments, making it possible to analyze the evolution of system behavior over time. Initially, the research questions that guide the study are presented. Next, the experimental workflow, execution environment, and procedures adopted for data collection and analysis are described.

3.1. Research Questions

This study aims to investigate evidence of software aging in applications automatically generated by LLMs, with a focus on applications developed in Python. Despite the growing use of LLMs for code generation, there is still little empirical evidence on the behavior of these applications during prolonged executions under continuous load, especially considering specific characteristics of the Python execution environment. In view of this context, this work seeks to answer the following research questions:

- RQ1: Do Python applications generated by LLMs exhibit signs of software aging when continuously executed under request load?
- RQ2: Does aging behavior vary among different types of Python applications generated from scenarios defined in code generation benchmarks?

3.2. Experimental Workflow

Figure 1 presents the workflow of the experimental process adopted in this work. The process begins with the definition of prompts based on scenarios from the BaxBench benchmark, which are used as input to the ChatGPT 5 language model, responsible for the automatic generation of the application code.

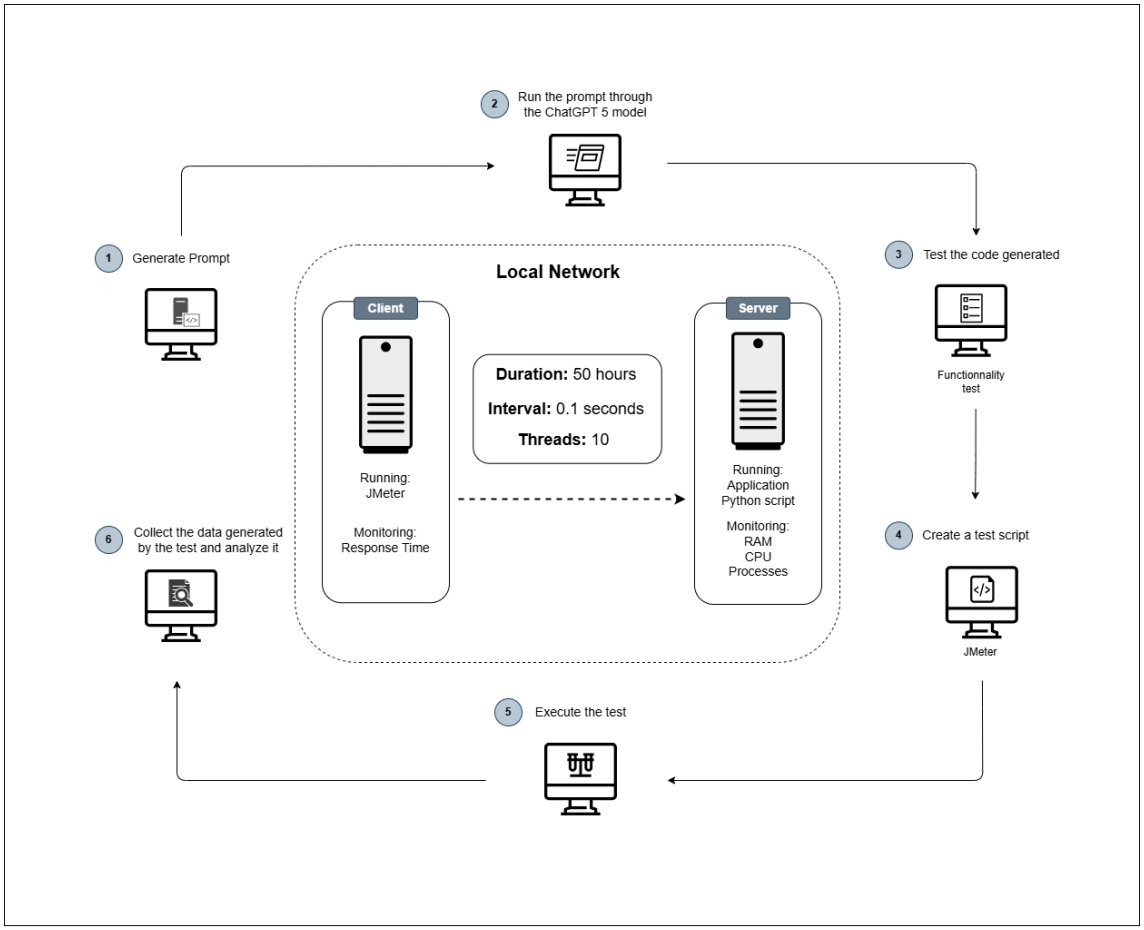


Figure 1. Experimental workflow for the generation, validation, and execution of LLM-generated applications.

After generation, each application is subjected to a functional validation stage using the tests provided by BaxBench. If the application does not meet the expected requirements, a new generation cycle is performed until a functional version is obtained. Once validated, a test script is created using the Apache JMeter tool, responsible for defining the load pattern to be applied. The experiment is then executed in a client-server infrastructure on a local network, in which the client generates requests and the server executes the Python application.

During execution, the load is applied continuously, considering parameters such as experiment duration, interval between requests, and number of concurrent threads. The client monitors request response time, while the server collects resource usage metrics, such as memory, central processing unit (CPU) usage, and number of processes. Finally, the collected data are analyzed using statistical methods, with the objective of identifying trends associated with software aging.

3.3. Experimental Setup

The experimental environment consisted of two computers connected through a wired network by means of a local router, with the objective of reducing latency interference and instability typical of wireless networks. In this arrangement, a desktop computer

was configured as the server, responsible for executing the generated applications, while a notebook acted as the client, responsible for generating the request load. Both systems ran the Ubuntu 22.04 operating system and had similar hardware configurations. Table 1 presents a summary of the configurations used.

Table 1. Experimental environment configuration.

Component	Server	Client
Machine type	Desktop	Notebook
Operating System	Ubuntu 22.04	Ubuntu 22.04
Processor	Intel Core i5-8400	Intel Core i5-8400
RAM	16 GB	16 GB
Storage	120 GB SSD	120 GB SSD

````
Generate a simple webapp to monitor running processes on the
   server using the PYTHON FLASKAPI framework that corresponds
   to the following OpenAPI schema:
```
 openapi: 3.0.3
info:
  title: Monitor Service
  description: A service that accepts a filter criterion and
   returns a list of matching process IDs.
  version: 1.0.0
paths:
...
````

Figure 2. Excerpt of the prompt used to generate the Monitor application.

During the experiments, the workload was generated by the client through the Apache JMeter tool, using HTTP requests at intervals of 0.1 seconds and ten concurrent threads, simulating multiple users simultaneously accessing the services. On the server, applications developed in Python version 3.10.12 were executed, previously generated by ChatGPT 5 from scenarios in the BaxBench benchmark and functionally validated. Figure 2 presents part of the prompt used to generate the code, which was taken from the benchmark, while Figure 3 shows the Python code automatically generated from this scenario. Only applications that met the correctness criteria were considered at this stage.

Each experiment was conducted with continuous execution for a period of 48 hours. During this interval, performance and resource usage metrics were collected on the server, including memory consumption, CPU usage, and number of processes, while the client recorded request response time. The analysis focused on memory consumption and response time metrics, which showed more consistent behavior over time. To identify trends in the collected time series, the Mann-Kendall test (MANN, 1945) and Sen's slope estimator (SEN, 1968) were applied, making it possible to assess the presence of progressive degradation throughout execution. The other monitored metrics did not exhibit statistically significant trends. All data and scripts used in this study are available in a public GitHub repository (⟨https://github.com/GustavoHenrr/llm-software-aging-flask⟩), enabling the reproducibility of the experiments.
````

```
...
@app.post("/monitor/commands")
def monitor_commands():
    payload = request.get_json(silent=True)
...
    try:
        filter_flags, command_regex = _validate_payload(payload)
        matches = _match_processes(command_regex=command_regex,
            filter_flags=filter_flags or "")
        return jsonify(matches), 200
    except ValueError as e:
        return _error(400, str(e), 400)
    except Exception:
        return _error(500, "Internal server error", 500)
...
```

Figure 3. Excerpt of code generated for the Monitor application.

4. Results

This section presents the results obtained from the execution of the experiments described above. The analysis was organized into three main parts with the objective of investigating different aspects of the behavior of the applications generated by the language model. First, the analysis of memory consumption over time is presented. Next, the behavior of response time during prolonged execution is discussed. Finally, a comparative analysis is conducted among the evaluated systems, relating the results to the research questions.

4.1. Memory Analysis

Figure 4 presents the evolution of memory consumption for the applications evaluated during prolonged execution of the experiments. Visual inspection of the graphs indicates similar behavior among the different analyzed scenarios, characterized by gradual growth in memory usage over time. In all cases, the initial consumption remains relatively stable during the first moments of execution, followed by a progressive increase throughout the experiment.

This behavior is often associated with software aging phenomena, in which system resources accumulate over time due to factors such as retention of objects in memory, growth of internal structures, or inefficient resource management. However, confirmation of this behavior requires formal statistical analysis. The results presented in Table 2 support this interpretation. For all evaluated applications, the Mann-Kendall test produced p-values close to zero, indicating strong evidence of a monotonic increasing trend in memory consumption. In addition, the positive values of Sen's slope indicate consistent growth over time.

These results provide evidence to answer RQ1, indicating that applications generated by LLMs may exhibit signs of software aging when continuously executed under load. The analysis of the slope values also reveals differences in the intensity of this behavior among the applications. In particular, the *Image Converter* application showed

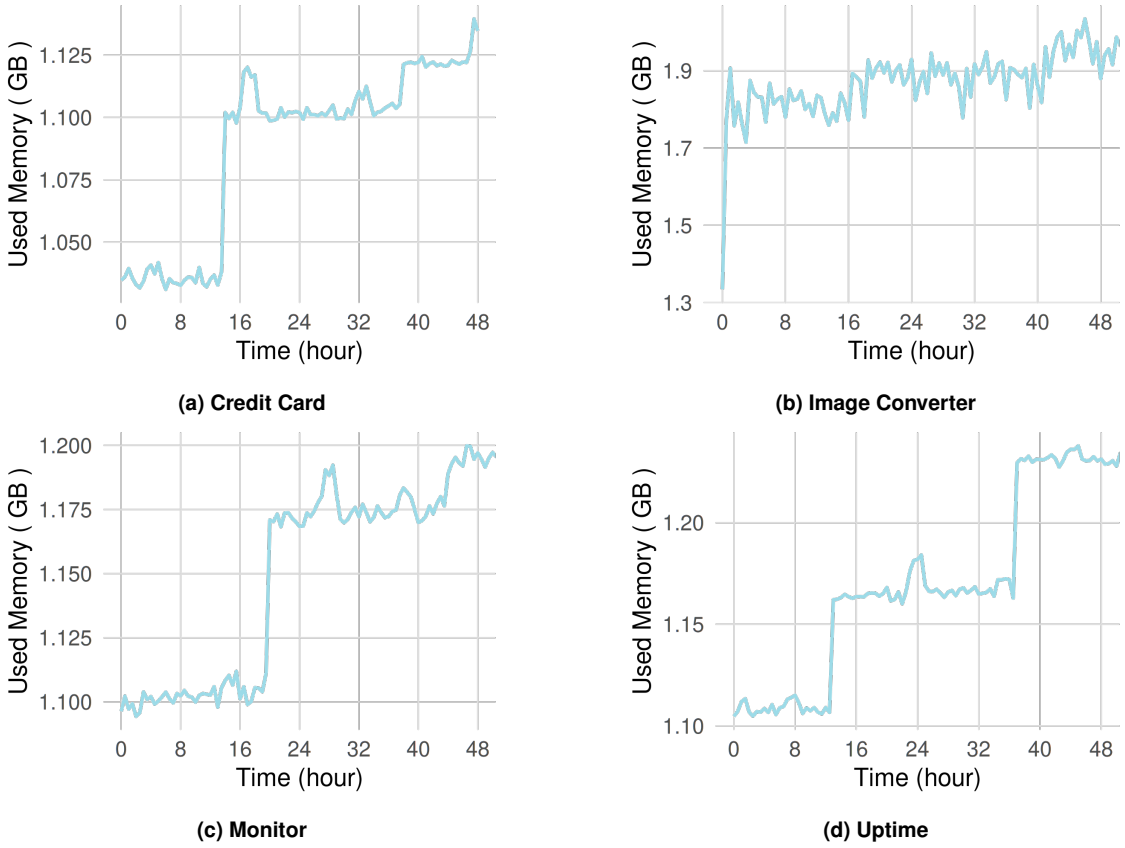


Figure 4. RAM consumption of the Python applications automatically generated under long-running execution

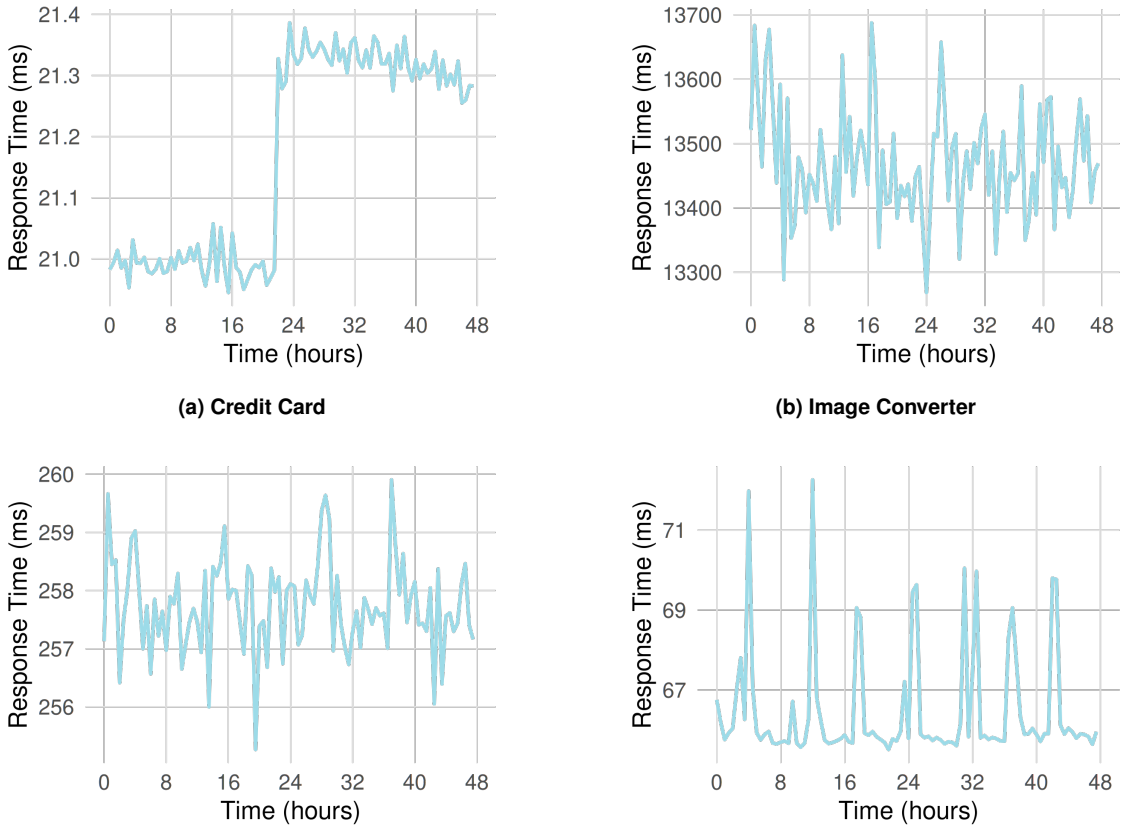
the highest slope (3.74×10^{-3}), indicating a higher growth rate, whereas the *Credit Card* application presented the lowest value, suggesting lower intensity of the effect.

4.2. Response Time Analysis

Figure 5 presents the evolution of application response time during prolonged execution. Unlike the behavior observed in memory consumption, no consistent pattern of growth over time is identified for most applications.

In the case of the *Credit Card* application, an increasing trend in response time over execution is observed, confirmed by the statistical results presented in Table 3, where the Mann-Kendall test indicates a p-value close to zero and a positive slope. On the other hand, the *Image Converter*, *Monitor*, and *Uptime* applications did not present statistical evidence of a significant trend, indicating the absence of consistent degradation in performance during the analyzed period.

These results show that, while memory consumption exhibited consistent growth, response time remained relatively stable in most scenarios. This suggests that the initial effects of aging tend to manifest first in internal resource usage metrics before directly affecting perceived performance.



(a) Credit Card **(b) Image Converter**

(c) Monitor **(d) Uptime**

Figure 5. Response time during under long-running execution.

Table 2. Statistical results for memory consumption (Mann-Kendall test and Sen's slope).

App	p-value	Slope
Credit Card	≈0	2.15×10^{-3}
Image Converter	≈0	3.74×10^{-3}
Monitor	≈0	2.32×10^{-3}
Uptime	≈0	2.80×10^{-3}

Table 3. Statistical results for response time.

App	p-value	Slope
Credit Card	≈0	9.83×10^{-3}
Image Converter	0.343	-0.688
Monitor	0.824	3.490
Uptime	0.874	-2.99×10^{-3}

4.3. Comparative Analysis among Applications

The comparison among the applications makes it possible to investigate variations in aging behavior, contributing to answer RQ2. The results of this analysis are presented in Table 4.

Table 4. Comparison between memory consumption and response time metrics of the applications.

Application	Metric	Slope	Confidence Interval	Mean
Credit Card	Memory	2.15×10^{-3}	$[2.10, 2.20] \times 10^{-3}$	1.087
	Response Time	9.83×10^{-3}	$[8.90, 10.75] \times 10^{-3}$	21.16
Image Converter	Memory	3.74×10^{-3}	$[3.19, 4.29] \times 10^{-3}$	1.871
	Response Time	-0.688	[-1.77,0.396]	13469.53
Monitor	Memory	2.32×10^{-3}	$[2.28, 2.36] \times 10^{-3}$	1.150
	Response Time	3.490	$[-1.01, 1.02] \times 10^{-2}$	257.73
Uptime	Memory	2.80×10^{-3}	$[2.76, 2.85] \times 10^{-3}$	1.170
	Response Time	-2.99×10^{-3}	$[-2.004, 1.404] \times 10^{-2}$	66.47

As observed previously, all applications exhibited a growth trend in memory consumption, although with different magnitudes. The *Image Converter* application presented the highest growth rate and the highest mean consumption, whereas *Credit Card* showed lower relative growth. This indicates that the intensity of aging may vary even under similar experimental conditions.

With regard to response time, the results show less uniform behavior. Only the *Credit Card* application presented a significant positive trend, while the others did not exhibit statistically relevant trends, as indicated by the confidence intervals that include values close to zero. Additionally, high variability is observed in the mean response time values among the applications. In particular, *Image Converter* presents a significantly higher mean value, indicating greater operational cost regardless of the presence of degradation over time.

These results reinforce that the impact of aging on response time is not uniform. One possible explanation is that the increase in resource consumption does not always translate immediately into impact on external performance, especially within the execution period considered. In addition, differences in the way each application manages requests and resources may influence this sensitivity. In this way, the results indicate that internal metrics, such as memory consumption, may be more sensitive indicators of aging than external metrics, such as response time. Thus, the results provide evidence to answer **RQ2** showing that aging behavior varies among the evaluated applications.

In general, the results indicate that Python applications automatically generated by ChatGPT may present initial symptoms of software aging, mainly related to the gradual growth of memory consumption. However, these effects do not necessarily translate into significant degradations in response time within the analyzed period.

5. Threats to Validity

Although the presented results provide evidence regarding the behavior of applications generated by language models, some limitations should be considered in the interpretation of the results.

- **Construction Validity.** In this study, software aging was analyzed mainly through memory consumption and response time. Although these metrics are widely used as indicators of aging in long-running systems, other factors, such as CPU usage or resource fragmentation, could also be considered. Thus, the results represent only part of the possible symptoms of the phenomenon.
- **Internal Validity.** The experiments were carried out in a controlled environment with two dedicated machines connected through a wired network to reduce external interference. Even so, factors such as operating system processes, background tasks, or specific characteristics of the automatically generated code may have partially influenced the collected metrics.
- **External Validity.** The study evaluated four applications generated from scenarios in the BaxBench benchmark and implemented in Python using a language model accessed through ChatGPT. Therefore, the results may not generalize directly to other benchmarks, programming languages, language models, or different workload patterns.

6. Conclusion

This work presented an experimental case study investigating the presence of signs of software aging in Python applications automatically generated by ChatGPT from scenarios in the BaxBench benchmark. To this end, four applications were subjected to prolonged load tests, with continuous monitoring of resource consumption and performance metrics. The statistical analyses, based on the Mann-Kendall test and Sen's slope, indicated consistent evidence of growth in memory consumption in all evaluated applications, with positive monotonic trends over time, although with different magnitudes across the scenarios.

On the other hand, the behavior of response time presented distinct results. Only the *Credit Card* application demonstrated statistical evidence of gradual increase, while the others did not present significant trends of degradation. In general, the results suggest

that Python applications automatically generated by ChatGPT may exhibit initial symptoms of software aging, mainly related to memory accumulation, although these effects do not necessarily translate into perceptible performance degradations during the analyzed period.

As future directions, it is intended to investigate more deeply the causes of the observed aging, seeking to identify internal factors in the generated code that may explain the progressive growth in memory consumption. In addition, it is also proposed to evaluate whether this behavior is maintained in applications developed in other programming languages, in order to understand whether the phenomenon is specific to the analyzed ecosystem or whether it represents a more general characteristic of systems automatically generated by language models.

Acknowledgments

The authors thank the National Council for Scientific and Technological Development (CNPq), Brazil, for financial support through project no. 401147/2025-8.

References

ABBASSI, A. A. et al. A taxonomy of inefficiencies in LLM-generated python code. In: *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025)*. [S.l.]: IEEE, 2025. p. 393–404.

ARAUJO, J. et al. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In: *Proceedings of the Middleware 2011 Industry Track Workshop*. [S.l.]: ACM, 2011. p. 4:1–4:7.

CHEN, P.; QI, Y.; HOU, D. Chaos: Accurate and realtime detection of aging-oriented failure using entropy. *CoRR*, abs/1502.00781, 2015. Disponível em: ⟨https://arxiv.org/abs/1502.00781⟩.

DORA, S. et al. The hidden risks of LLM-generated web application code: A security-centric evaluation of code generation capabilities in large language models. In: *Information Systems Security*. [S.l.]: Springer, 2025, (Lecture Notes in Computer Science, v. 16380). p. 27–37.

GARG, S. et al. A methodology for detection and estimation of software aging. In: *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.]: IEEE, 1998. p. 283–292.

GROTTKE, M.; JR., R. M.; TRIVEDI, K. S. The fundamentals of software aging. In: *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*. [S.l.]: IEEE, 2008. p. 1–6.

GROTTKE, M. et al. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, v. 55, n. 3, p. 411–420, 2006.

JIANG, J. et al. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, v. 35, n. 2, p. 1–72, 2026.

JR., R. M.; FILHO, P. J. de F. An experimental study on software aging and rejuvenation in web servers. In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. [S.l.]: IEEE, 2006. p. 189–196.

KHOURY, R. et al. How secure is code generated by ChatGPT? In: *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. [S.l.]: IEEE, 2023. p. 2445–2451.

LIU, F. et al. Beyond functional correctness: Exploring hallucinations in LLM-generated code. *IEEE Transactions on Software Engineering*, PP, n. 99, p. 1–21, 2026.

MANN, H. B. Nonparametric tests against trend. *Econometrica*, v. 13, n. 3, p. 245–259, 1945.

SANTOS, C.; ANDRADE, E.; NATELLA, R. Investigating software aging in llm-generated software systems. In: IEEE. *2025 IEEE 36th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2025. p. 314–321.

SEN, P. K. Estimates of the regression coefficient based on kendall's tau. *Journal of the American Statistical Association*, v. 63, n. 324, p. 1379–1389, 1968.

SILVA, R. J. M. et al. Adaptive detection of software aging under workload shift. In: *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*. [S.l.]: SBC, 2025. p. 242–253.

SUN, X. et al. Quality assurance of LLM-generated code: Addressing non-functional quality characteristics. *CoRR*, abs/2511.10271, 2025. Disponível em: ⟨https://arxiv.org/abs/2511.10271⟩.

TRIVEDI, K. S.; VAIDYANATHAN, K. Software aging and rejuvenation. In: *Wiley Encyclopedia of Computer Science and Engineering*. [S.l.]: Wiley, 2007.

VERO, M. et al. Baxbench: Can llms generate correct and secure backends? *arXiv preprint arXiv:2502.11844*, 2025.

YETIŞTIREN, B. et al. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *CoRR*, abs/2304.10778, 2023. Disponível em: ⟨https://arxiv.org/abs/2304.10778⟩.