Analysis of Current Trends in Software Aging: A Literature Survey

Tajmilur Rahman¹, Joshua Nwokeji¹ & Tejas Veeraganti Manjunath¹

¹ Gannon University, Erie, PA, United States

Correspondence: Tajmilur Rahman, Department of Computer and Information Science, Gannon University, Erie, PA, 16541, USA. E-mail: rahman007@gannon.edu

Received: July 4, 2022    Accepted: August 17, 2022    Online Published: August 22, 2022

doi:10.5539/cis.v15n4p19    URL: https://doi.org/10.5539/cis.v15n4p19

Abstract
Software aging and architecture degradation are important areas in software quality assurance. Existing research in these areas has developed mitigation strategies for software aging, other researchers have analyzed strategies for identifying software aging. Regarding architectural degradation, current studies have designed techniques for reducing degradation. However, there appears to be a paucity of studies on the causes of software aging and architectural degradation. Insight into the causes of software aging and architectural degradation can provide a critical perspective and further strengthen the research endeavors on prevention techniques. Using recursive literature review (RLR) and Bootstrapping techniques, this research identifies and analyzes the causes of software aging and architecture degradation in software systems. We found that besides many other causes, architectural degradation is one of the key reasons that cause software aging and acts as a barrier to the sustainability of software architecture.

Keywords: survey, software aging, architectural degradation, architectural drift

1. Introduction
Software failure is an undesirable quality of any software, especially from the end-users perspective, it threatens the reliability, availability, and ability of software to achieve the intended business objectives (Saraswat, 2008). Software aging is a phenomenon that contributes to software failures (Torquato, 2018). Software aging is defined as a deterioration in the state of a software resulting from the gradual accumulation of errors, naturally occurring aging-related bugs (ARBs) e.g., data corruption, memory bloating, computational errors, and resource depletion are the major causes of software aging (Castelli, 2001) (Cotroneo & Iannillo, 2020a).

In order to enhance reliability and availability, software engineering scholars have studied and proffered solutions to the many other factors that contribute to software failure. Some examples of these factors are bugs, hardware errors, and human errors (Yakhchi, 2015). However, because of the organic and subtle depletion that characterizes as ARBs, software degradation can be difficult to identify during testing (Cotroneo & Iannillo, 2020).

Indeed there has been an increasingly scholarly effort in the study of software aging, this is easily seen in the number of available peer-review publications. However, current efforts focus on approaches for predicting and detecting software aging. For instance, some authors (Liu & Meng, 2019) (Liu & Tan, 2019) have applied artificial intelligence and machine learning models to predict software aging. A growing number of studies focus on understanding the impact of software aging on the performance, availability, and reliability of computing systems. Some examples include the use of edge computing to analyze the impact of software aging on cyber-physical systems (Andrade & Machida, 2019) and the use of stress testing to investigate how software aging impacts the availability and reliability of Android applications (Cotroneo & Iannillo, 2020). Other scholarly efforts on software aging include the proposal for SWARE (Torquato, 2018). The main objective of SWARE is to provide an experimentation framework for investigating software aging symptoms and the effectiveness of rejuvenation’ as a technique for preventing software aging.

While existing scholarly work in software aging makes important contributions to knowledge, there is a need to provide insight into the trends in the research and practice of software aging. Such insight would be very helpful in articulating and defining future research direction, and in addition, provides critical perspectives that could inform practice. It is possible that new computing technologies such as machine learning and artificial intelligence which are now integrated into software development may have changed the causes and other trends in software aging. Therefore, a continuous review of literature is necessary to understand how and if the trends in software aging have changed, the factors that are causing such change in trends, and how the new trends can be leveraged to improve research and practice.

In this paper, we use bootstrapping and recursive literature review methodologies to articulate critical issues and trends in software aging that are often neglected in existing research. We reviewed a total of 236 papers that met our search criteria to identify: current problems and challenges in software aging and causes of software aging. More so, we report on the
citation trends, type, and rank of publication venue as well as the publication year. To meet the aim of our study, we set up the following research questions:

- **RQ1** - What are the publication trends in software aging in terms of rank, type, and year of publication?
- **RQ2** - What factors contribute to software aging and architectural degradation in software?
- **RQ3** - Is there a relationship between architecture degradation and software aging?
- **RQ4** - What are the problems or challenges in the area of software aging?

2. **Study Significance**

Insight into the critical issues covered in these research questions would be imperative to the research and practice of software aging for many reasons. First, understanding the factors that contribute to software aging would help design effective solutions for software aging. Moreover, such understanding could inform the development of preemptive strategies for software aging at the pre-implementation phase of SDLC i.e., during planning and requirements analysis. This would introduce a critical shift in the current approach to solving software aging problems i.e., solutions are designed post-implementation and deployment. This approach is reactive, costly, and inefficient. Secondly, it has been hypothesized that software architecture degradation contributes to software aging (De Silva & Balasubramaniam, 2012) (Heinkens & Graaf), however, the relationship between software aging and architecture degradation has not been adequately studied in literature and are often neglected. Our study fills this knowledge gap by providing insight into if and how software architecture degradation contributes to software aging.

The rest of the paper is outlined as follows. Section 3 gives a background of the study, Section 4 gives an overview of the related works in the domain of our study, Section 5 describes the methodology that we apply for this study, and Section 6 discusses the current literature from different aspects. Section 7 describes what we have observed and answering the research questions defined by us in Section 1 followed by a conclusion and future work in Section 8.

3. **Background**

David Parnas was the first to identify the phenomenon of Software Aging and published a paper proposing some best practices to prevent aging or at least increase the life span of a software system (Parnas, 1994). He stated – “Programs, like people, get old. We can’t prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable.”

The phenomenon of software aging has grown tremendously in both academic and industrial areas. Its causes and effects have been a subject of investigation, detection, and research. A few of the primary reasons why Software aging is caused are memory bloating, memory leakage (Cassidy, 2002) (Cotroneo, 2016) (Cotroneo & Iannillo, 2020), as well as data corruption (Vaidyanathan & Gross, 2003) (Cotroneo, 2010), and unreleased file locks (Garg, 1998) (Vaidyanathan & Trivedi, 2005).

The biggest cause of software failure is unplanned system outage as mentioned by Castelli et al. (Castelli, 2001). Hardware failure does not cause a significant impact on the development of software failures. A number of factors such as numerical error accumulation, data corruption, and unlimited resource consumption can play an enormous role in the increasing failure rate and it causes software failure. The software can never be fully deemed bug-free. Aging arises due to the complexity in the software that is never free of errors. The development of software tends to be managed by the need to meet the deadlines of the release rather than to ensure that the software is reliable. Designing software that can be immune to aging is difficult (Castelli, 2001). The rate of aging can differ from software to software.

Android OS Architecture consists of a stack of software components that can be divided into Linux kernel (ex. display driver, binder driver .etc), Android Libraries (ex. software manager, media framework .etc), Android Runtime (ex. Dalvik Virtual Machine .etc), Application Framework (high-level services such as memory utilization and consumption, and garbage collection .etc), and Application or Task (stock application or user-installed applications). According to the paper “Software Aging Analysis of the Android Mobile OS” by Domenico Cotroneo et al (Cotroneo & Iannillo, 2020). Mobile devices utilizing the Android OS are significantly complex, having rich and heavily customizable features. Thus, they are prone to software reliability and performance issues (Cotroneo, 2016). The devices are degraded in responsiveness over a period of time and eventually fail. This degradation in the device is categorized as Software aging in Android mobile OS.

The authors considered different metrics such as memory, storage utilization, garbage collection, and tasks performed by the application, that can be correlated with one key metric, “Launch Time”. Launch time is the time between the user interaction to the start of an application and the appearance of the user interface of the application. An increasing trend in launch time was exhibited by Android applications that were triggered by experiments that showcased software aging
effects. The type of workload processed on the device contributes to the amount of software aging observed. An increase in memory consumption causing memory degradation was directly proportional to the degradation of the launch time for specific processes of the android system. Similarly, garbage collection time and the launch time, which is the time spent on the Android Runtime to reclaim heap memory from Android processes, are highly correlated with each other.

According to task analysis, certain components like the activity manager in the Android OS frequently caused a trend in terms of virtual memory utilization and CPU, which showed behavioral deviations in the presence of software aging. Android processes that are part of the Android AOSP (Android Open Source Project) codebase, are used by a variety of devices and vendors. The focus of vendor customization is on device drivers and apps, with very minor changes to the underlying AOSP operations. As a result, software aging-related problems that are identical are likely to impact devices from a variety of vendors. Vendor customizations may, however, cause additional software aging problems. From the above observations and considering a transitive relationship, we can identify that architecture also plays a role in software aging.

The authors Erik Whiting and Sharon Andrews of the paper “Drift and Erosion in Software Architecture: Summary and Prevention Strategies” talk about how architectural degradation is caused due to system complexity increasing gradually over time and changes in the requirement set. Architectural degradation can be classified into two types: architectural drift and architectural erosion (Whiting & Andrews, 2020). Architectural drift is the gap between the planned and current architecture of the software system that does not violate the intended architecture. On the other hand, software erosion is similar to software drift but violates the intended design. They say that both forms of these degradations threaten the integrity of the system’s architecture, hence the project and team as a whole. Software that has undergone architectural erosion or decay may be of less quality, more complex, and more difficult to maintain. As the software application receives oncoming changes, it becomes increasingly difficult to comprehend the software architecture that was originally intended (Whiting & Andrews, 2020).

Gurp et al. discussed in the paper “Design erosion: problems and causes” how erosion affects a software system in the evolution of its design (Gurp & Bosch, 2002). They found that despite being cautious about the design, software erosion was caused over a period of time, and the only solution was redesigning it from scratch. They have also shown how design decisions and requirements changes can affect the software system, which in turn also changes the architecture, resulting in erosion.

According to Gurp et al., architectural decisions that address architecturally significant requirements are referred to as design decisions. The type of application, the distribution of the system, the architectural styles to be employed, and how the architecture should be documented and evaluated are all architectural design decisions. The non-functional aspects of a system are influenced and impacted by architectural decisions. Each architectural decision refers to a specific, architecturally significant design problem that has several alternative solutions. An architectural decision documents the conclusion of a conscious, typically collaborative option selection process and provides a design explanation for the decision. Gurp and Bosch considered an example of a simulator of a bank machine that replicates the functionality of an Automated Teller Machine (ATM).

In the first version, a simple work of the system with the basic requirements of an ATM was considered. It was a compact version but with low maintainability, design, and flexibility, it was not a very ideal system. Later on, four updates were made to the initial version to address these issues. To achieve this, they changed the program structure by adding new classes, moving around blocks of code, and other optimization techniques. The design of version five had the same functionality as the first version, but it was much more complex and heavy. The newly added code structure improved the flexibility but compromised the maintainability of the code by making it harder to understand for the developers. The final version’s features are the outcome of the design decisions made in the previous versions, such as adding GUI etc. The options were no longer considered ideal for version five due to the changes in the requirements. As a result, version five was not the best design decision for the requirements that they took into account. However, building an ideal system would require discarding much of the code that they had already written in previous versions.

From this, they conclude that even an optimal design strategy during the design phase does not deliver an optimal design. The reason is that changes in the requirements will occur in future evolution cycles which may cause the design decisions taken earlier to be less optimal. This proves that architectural drift and architectural erosion are parts of software development, no matter how meticulously the software is developed (Gurp & Bosch, 2002).

From the above studies, we understand that the causes of software aging are related to the design decisions taken over a period of time, which in turn causes architectural degradation. The resulting effects of both software aging and architecture degradation are homogeneous. Therefore, according to existing studies, architectural degradation is one of the key factors of software aging.
4. Related Works

Baabad et al. (Baabad, 2020) did a systematic literature review (SLR) on the existing studies on the architectural erosion of the Open Source Software (OSS) projects. They collected the research papers from eight major online databases (ACM, Springer, ScienceDirect, Taylor, IEEE Explorer, Scopus, Web of Science, and Wiley) and ended up with a total of 74 papers for their study. They found that rapid software evolution, frequent changes, and the lack of developers’ awareness are the most common reasons behind the architecture degradation for OSS projects. The repeated architectural erosion shortens the lifetime of the system affecting the architecture entirely which results in redesigning the architecture of the system from scratch (rejuvenation) (Li and Long, 2011). This phenomenon is also known as software aging (Parnas, 1994).

Another similar study has been performed by Pietrantuono (Pietrantuono & Russo, 2020). They reviewed the effort conducted by the software aging and rejuvenation (SAR) community in the cloud domain. From a total of 105 existing research papers from three major digital libraries, they characterized the existing literature according to four dimensions: the publication trends, the aging analysis methods, and metrics, the rejuvenation solutions, and the validation approach. Compared to their study, we analyze the existing studies into “citation trend”, “year of publication”, “publication venue”, “reasons behind software aging”, and “solutions proposed”.

Controneo et al. (Cotroneo, 2014) did a survey on software aging and rejuvenation. They discussed the concepts such as aging-related failures, aging effects, aging bugs, and aging indicators in short. Similar to our approach they also followed bootstrap and recursive closure. Controneo classified the software aging and rejuvenation papers into four dimensions: type of system, aging effects, aging indicators, and rejuvenation techniques. They discussed different approaches that were conducted by researchers. For example, model-based analysis, measurement-based analysis, and hybrid analysis. In model-based analysis, they mostly took theoretical scenarios rather than real case scenarios. They make assumptions about a system that may not apply to all real-world scenarios. In measurement-based analysis, they collect system data such as memory space allocations and swap memory during the run-time or while starting the system. They compare the current measurement values with the previous or initial values and analyze if there is a degradation in the system which is caused by aging. They also discuss the threshold-based approach in which the system is run or ignored until a certain amount of fault or problems occurs. The hybrid analysis is a combination of model-based analysis and measurement-based analysis. Although the study approach is similar, the objective of our study is different where our survey focuses on what are the primary causes identified for software aging and what are the common solutions proposed by the existing literature.
5. Methodology of Our Analysis

This study aims to synthesize and present a critical perspective on the current research in the field of Software Aging and Architectural Degradation. We followed the methodology applied by Cotroneo et al. (Cotroneo, 2014) in a similar study on a survey of software aging and rejuvenation. Cotroneo et al. explored the fundamental ideas of Software Aging and Rejuvenation (SAR). Figure 1 shows the steps of our methodology at a glance. They discuss how software aging impacts a system and the necessity of rejuvenation.

They briefly examine aging-related failures, aging impacts, aging bugs, and aging indications. The primary goal of the paper is to conduct a literature analysis on SAR. Cotroneo applied two approaches to gather the information that they named “Bootstrap” and “Recursive Closure”. Bootstrap comprises of gathering papers using a keyword-based search, and recursive closure is to collect papers from the references that are related to the topic they are interested in for their study. However, they selected only the relevant papers from the top most conference and journal publications. A thorough review of Software Aging and Rejuvenation studies is provided by them and the articles are divided into four categories based on the “Method of Analysis”, “Type of the System”, “Aging Effects” and “Aging indicators” approaches.

5.1 Bootstrap

Table 1 represents the number of papers we collect from the bootstrap method. “Keyword” column lists all the keywords that we used to search for the initial set of papers. The column “Default Search Result” shows the original search result using the corresponding keyword in the first column. The column “Result After Filtering” shows the number of papers after applying the inclusion (IC) and exclusion (EC) criteria.

Table 1. Table of Keyword-based search results

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Source</th>
<th>Default Search Result</th>
<th>Result After Filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Software Aging”</td>
<td>IEEE Xplore</td>
<td>109</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>ACM Digital Library</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>“Software Aging” and “Rejuvenation”</td>
<td>ACM Digital Library</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>“Architectural Degradation”</td>
<td>ACM Digital Library</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>“Architectural Drift”</td>
<td>ACM Digital Library</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>“Architectural Erosion”</td>
<td>ACM Digital Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>“Architecture Degradation”</td>
<td>ACM Digital Library</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>“Architecture Drift”</td>
<td>ACM Digital Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>“Architecture Erosion”</td>
<td>ACM Digital Library</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Wiley Online Library</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Search result for different keywords in different databases of publications. The default search result includes papers that matched with the keyword but actually not inline with our the context of this study.

The bootstrap phase uses a keyword-based search for the initial set of papers. Keyword-based search is the process of finding and researching specific search terms in search engines to look up a database of a collection of papers, journals, and conferences. First, we collect the initial set of papers based on keyword-based search as shown in Figure 1. We use the following keywords that are closely related to our research goals and questions to find papers related to the field or subject and assess the impact of the papers:

- “Software Aging”
“Software Aging” and “Rejuvenation”
“Architectural Degradation”
“Architectural Drift”
“Architectural Erosion”
“Architecture Degradation”
“Architecture Drift”
“Architecture Erosion”

For our search with the keywords, we are interested in the high volume, and highly popular publication databases since they have a rich collection of research papers, journals, and articles.

While performing the keyword-based search it is also important to understand the context of where the keyword is being used. For example, while searching for the term “Software Aging”, the search engine may find results that include the words “Software” and “Aging” in different parts of the abstract in the paper. However, these papers may not discuss the concept of software aging. Hence, it is crucial that we find papers that are apt for our study.

With the help of advanced search, we skim through multiple databases that help us connect various different ideas across domains through prominent publications and citations. We performed our search in three different databases, IEEE Xplore, ACM Digital Library, and Wiley Online Library. IEEE Xplore is by far the most comprehensive academic database in the field of engineering and computer science. It is possible to search for not just journal articles, but also conference papers, standards, and books that have met rigorous quality with billions of cited references. The ACM Digital Library provides a complete collection of publications and articles in the fields of computing and information technology. Wiley Online Library focuses on academic publications and instructional materials and has one of the largest collections of online journals, research resources, and books on a large variety of subjects.

Each keyword is subject to an advanced keyword-based search in all three of the research databases i.e IEEE Xplore, ACM Digital Library, and Wiley.

To the default set of papers that were produced by the databases after the search, several inclusion criteria (IC) and exclusion criteria (EC) were applied to filter the papers.

The final selection of papers is then collected when these conditions were fulfilled for the next step which is recursive closure. For example, when the keyword “Software Aging” was searched in all three databases, IEEE Xplore generated 109 papers, ACM Digital Library generated 5 papers and Wiley generated 6 papers, bringing the total initial set of papers generated for the keyword Software Aging to 120. After applying the inclusion criteria (IC) and exclusion criteria (EC), IEEE Xplore generated 98 papers, ACM Digital Library generated 5 papers and Wiley generated 5 papers for the final set of papers bringing the total of the final set of papers generated from the keyword “Software Aging” to 108.

A total of 192 papers have been obtained in the bootstrap phase, and after applying the inclusion and exclusion criteria the initial set of papers was 165.

5.2 Recursive Closure

The recursive closure phase is based on gathering further papers from the references of the initial set of papers obtained during the bootstrap phase. We examine these papers and scrutinize them to see whether they are relevant to our field of research. All relevant referenced papers are added to the gathered collection of papers once each paper’s references are reviewed. A search is conducted to see whether the references contain any further relevant publications. If there are, the papers are subjected to the selection criteria and the procedure is restarted until no more relevant articles are found. The final selection of papers contains all of the gathered publications and brought the total number of papers to 236 papers.

Following are the inclusion criteria (IC) and exclusion criteria (EC) using which the papers were filtered:

- IC1: Papers discussing studies and research on “Software Aging”, “Software Aging and Rejuvenation”, “Architectural Degradation”, and “Architectural Drift and Erosion”.

- IC2: Papers that have only been published in conferences and journals are considered.

- IC3: Papers that are indirectly related to the topics and are relevant to our study.

- EC1: Language in which the paper was published. In our survey, we only considered the papers that were written in English.

- EC2: Dissertations or thesis have been excluded.
• EC3: The previous versions of a paper are excluded and only the most recent versions are included.
• EC4: Studies that are incomplete.
• EC5: Papers that are not discussing “Software Aging”, “Software Aging and Rejuvenation”, “Architectural Degradation”, and “Architectural Drift and Erosion” in the domain of “Software Engineering”.

After evaluating the final set of papers and gathering a substantial number of papers in both domains, we start accounting for various qualities such as venue of publication, year of publication, and the number of cites.

6. Our Findings

6.1 Year of Publication

The phenomenon of Software Aging was first conceptualized in the year 1994, and the first paper related to this subject was published at the International Conference on Software Engineering (ICSE). Since then, there has been exponential growth concerning research in the field of Software Aging and Rejuvenation. Even though Software Aging and Rejuvenation is not a new topic, it has been constantly gaining importance throughout the years. There have been several conferences and workshops held to publish and discuss topics related to Software Aging and Rejuvenation.

Figure 2 represents the timeline of the number of publications, we can observe that there are multiple inconsistencies in the number of publications each year.

![Figure 2. Timeline of the number of publications in the field of Software Aging](image)

A few of the prominent years in which there were a high number of publications are 2020, 2012, 2016, 2019, and 2013 respectively, and the prominent locations of those publications are the IEEE International Conference on Software Reliability Engineering Workshops (ISSREW), the International Symposium on Software Reliability Engineering (ISSRE), the IEEE International Workshop on Software Aging and Rejuvenation (WoSAR), and Software Quality Journal. In the years 2014, 2015, and 2018 most of the prominent venues did not conduct a conference. For example in 2015, only 7 papers were published with the IEEE International Conference on Software Reliability Engineering Workshops (ISSREW) publishing 2 papers. This caused a sudden drop in the number of papers published during that year.

Figure 3 shows the timeline of the number of publications in Architectural Degradation.

Architectural Degradation is an important area to be researched but it is quite neglected. Even though Architectural Degradation has been around for a very long time, the number of papers published in this area is inadequate when compared to Software Aging. There is very little prominence given to Architectural Degradation compared to Software Aging even though they have common effects on the same software.

As we see in Figure 3 only a few papers published on this subject before 2003 with the first paper published in the year 1998. The year 2020 had the highest number of papers published related to Architectural Degradation with 5 papers published from different venues. The most number of papers published before that was in the year 2003, where 4 papers
that are related to Architectural Degradation were published. Apart from 2011, 2013-2015 there were only around one or two papers published related to this subject. However, there are several textbooks written and published on this subject.

6.2 Number of Publications.

6.2.1 Most Cited Paper in the Domain of Software Aging

Table 2 lists the papers that explicitly did study on software aging. The DOI is also associated to the paper titles.

<table>
<thead>
<tr>
<th>Title of the paper</th>
<th>DOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software aging</td>
<td><a href="https://doi.org/10.1109/ICSE.1994.296790">https://doi.org/10.1109/ICSE.1994.296790</a></td>
</tr>
<tr>
<td>Proactive management of software aging</td>
<td><a href="https://doi.org/10.1147/rd.452.0311">https://doi.org/10.1147/rd.452.0311</a></td>
</tr>
<tr>
<td>A methodology for detection and estimation of software aging</td>
<td><a href="https://doi.org/10.1109/ISSRE.1998.730892">https://doi.org/10.1109/ISSRE.1998.730892</a></td>
</tr>
<tr>
<td>Analysis of software aging in a web server</td>
<td><a href="https://doi.org/10.1109/TR.2006.879609">https://doi.org/10.1109/TR.2006.879609</a></td>
</tr>
<tr>
<td>The fundamentals of software aging</td>
<td><a href="https://doi.org/10.1109/ISSREW.2008.5355512">https://doi.org/10.1109/ISSREW.2008.5355512</a></td>
</tr>
<tr>
<td>An approach for estimation of software aging</td>
<td><a href="https://doi.org/10.1109/ISESE.2002.1166929">https://doi.org/10.1109/ISESE.2002.1166929</a></td>
</tr>
<tr>
<td>Modeling and analysis of software aging and rejuvenation</td>
<td><a href="https://doi.org/10.1109/SIMSYM.2000.844925">https://doi.org/10.1109/SIMSYM.2000.844925</a></td>
</tr>
<tr>
<td>A survey of software aging and rejuvenation studies</td>
<td><a href="https://doi.org/10.1145/2539117">https://doi.org/10.1145/2539117</a></td>
</tr>
<tr>
<td>A workload-based analysis of software aging, and rejuvenation</td>
<td><a href="https://doi.org/10.1109/TR.2005.853442">https://doi.org/10.1109/TR.2005.853442</a></td>
</tr>
<tr>
<td>Advanced pattern recognition for detection of complex software aging</td>
<td><a href="https://doi.org/10.1109/DSN.2002.1028933">https://doi.org/10.1109/DSN.2002.1028933</a></td>
</tr>
</tbody>
</table>

Table containing the research papers along with their doi, that discuss about software aging.

Figure 4 shows the most cited papers on Software Aging with DOIs on the Y axis instead of titles (due to space limitation) so that we can relate to the Table 2. The paper titled as “Software Aging” by Parna et al. (Parna, 1994) is the most cited paper in the history.

Parnas, in the paper “Software Aging”, states that the two main causes of software aging are the failures caused due to the owner’s unable to modify a product to meet its changing needs and also the result of changes that are made due to ignorant coding by developers.

Software aging cannot be prevented, but it can be slowed down (Parnas, 1994). The prominent method to slow down software aging is by documenting the code well and making notes of the major and minor changes as well as documenting which function or module can perform what functionality. This helps the maintainers of the code to make changes easily without creating further problems. It also prevents deterioration of the aging problem by adding new bugs without understanding the previous code that can break the product as a whole. Adding a new feature to the existing software should be done with background knowledge of the program. Software aging causes financial problems to the company if
the software is not able to meet the current standards or the requirements of the customer.

Parna et al. (Parnas, 1994) also states that we must design for change which is a concept where the program is tolerant to the new changes. The main code is separated from the code that can be changed or is susceptible to change. It also states that the programmer developing the program should be the one writing the documentation rather than a non-technical person. Finally, it states that the company should be ready for software aging well before the actual product is released, that is the company has to be financially and physically ready to face the old age of the program by taking precautions and making assumptions about the future.

Castelli et al. (Castelli, 2001) had the second most cited paper in our database. The paper states that a system-wide blackout is a result of multiple software failures. The main contributors to software failures are the exhaustion of system resources, data corruption, numerical error accumulation, storage space fragmentation, bloating and leaking of memory, unterminated threads, and unreleased file locks. These failures cause performance degradation over a period of time. This phenomenon of degradation is Software Aging. As the software grows over a period of time, the failure rate increases due to the increase of the factors mentioned earlier. Another factor is that the code becomes extremely complex over a period of time and contains bugs or errors that are settled down like a deep stain, then fighting against Software Aging becomes extremely complex.

Castelli et al. created a framework that detects, predicts, and manages rejuvenation techniques such as re-developing the entire system or restarting the middleware. They implemented the framework on the xSeries of IBM servers. The xSeries Software Rejuvenation Agent (SRA) was developed to track and examine the consumable resource and estimate the time that these resources would be exhausted and alert the organization to prepare for rejuvenation. This resulted in a significant improvement of the cluster system availability and reduced the downtime cost.

As we see in Figure 4, Garg et al.’s paper “A methodology for detection and estimation of software aging” (Garg et al., 1998) was the third most cited paper in our database. In this paper, the authors claim Software Aging to be a phenomenon of failure or end-user crash which is caused by the accumulation of errors during the execution of the software. They also claim that software aging causes performance degradation. The factors affecting are similar to the previously discussed paper such as memory bloating and leaking, data corruption, storage space fragmentation, etc. The authors proposed an SNMP-based distributed monitoring tool that was built to collect data on the usage of the operating system and also the system activity on a UNIX Workstation. They proposed a new metric “Estimated time to exhaustion” which was used to detect and validate the existence of aging. They used techniques like slope estimation technique and statistical techniques for the detection and estimation of aging. They found that the metric was inversely co-related with aging that is as the value of the metric was high the effect and presence of aging were low. On the contrary, if the value of the metric was low then the presence and effects of aging were high. The metric also helped in comparing the effect of aging with respect to different system resources and the identification of the most prominent resource to be monitored and managed to prevent
aging.

6.2.2 Most Cited paper in the Domain of Architectural Degradation

Number of research works have been done on architectural degradation as well. Table 3 lists the papers that explicitly studied architectural degradation. The table lists the paper titles along with their DOIs.

Table 3. Architectural Degradation papers with DOI and Title

<table>
<thead>
<tr>
<th>Title of the paper</th>
<th>DOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recommending refactorings to reverse software architecture erosion</td>
<td><a href="https://doi.org/10.1109/CSMR.2012.40">https://doi.org/10.1109/CSMR.2012.40</a></td>
</tr>
<tr>
<td>Assessing architectural drift in commercial software development: a case study</td>
<td><a href="https://doi.org/10.1002/spe.999">https://doi.org/10.1002/spe.999</a></td>
</tr>
<tr>
<td>Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance</td>
<td><a href="https://doi.org/10.1109/ICSM.2011.6080779">https://doi.org/10.1109/ICSM.2011.6080779</a></td>
</tr>
<tr>
<td>Using architectural properties to model and measure graceful degradation</td>
<td><a href="https://doi.org/10.1145/2577080.2577087">https://doi.org/10.1145/2577080.2577087</a></td>
</tr>
<tr>
<td>Blending and reusing rules for architectural degradation prevention</td>
<td><a href="https://doi.org/10.1145/1869542.1869563">https://doi.org/10.1145/1869542.1869563</a></td>
</tr>
<tr>
<td>Stop the software architecture erosion: building better software systems Complementing model-driven development for the detection of software architecture erosion</td>
<td><a href="https://doi.org/10.1109/MiSE.2013.6595292">https://doi.org/10.1109/MiSE.2013.6595292</a></td>
</tr>
<tr>
<td>Lightweight prevention of architectural erosion</td>
<td><a href="https://doi.org/10.1109/IWPSE.2003.1231211">https://doi.org/10.1109/IWPSE.2003.1231211</a></td>
</tr>
</tbody>
</table>

Table containing the research papers that discuss about the architectural degradation along with their doi.

Figure 5 shows the number of citations of the papers in Architectural Degradation. The research paper by Lakshitha de Silva and Dharini Balasubramaniam is the most cited paper titled as “Controlling software architecture erosion: A survey” (De Silva & Balasubramaniam, 2012). In this paper, authors state that the most important properties and design elements of a software system are captured by software architecture. The modification to the system increases over a period of time and it violates the architectural principles which in turn degrades the system and its performance. Resulting in shortening the lifetime of the system. To combat and address these issues, the paper discusses the different strategies along with their advantages and disadvantages that can be used to detect and restore the eroded architecture of the system. They classify and combine different strategies to either minimize, repair, or prevent architectural erosion.

The second most cited paper titled “Recommending Refactorings to Reverse Software Architecture Erosion” authored by R Terra, M T Valente, K Czarnecki, R S. Bigonha, states that Architectural Erosion is a gap between the design architecture compared to the implemented architecture of the source code. Due to this the software architecture degrades the quality of the system and decreases maintainability, evolvability, extensibility, and reusability. Over a period of time,
these gaps increase transforming the software architecture into an unmanageable state (Terra, 2012). To counteract this, they proposed a tool that can locate the point of erosion in a software system that is detecting the source code violations from the implemented architecture to the proposed architecture. Also providing the solution to resolve the violations that are caused by the deviation from the proposed architecture. In the paper, they have considered an example of a web-based eCommerce system called “MyWebMarket”. They created an architecture to resemble a real-world small-scale project in which they used architecture erosion and used their approach to detect it. They considered four constraints that caused violations to the architecture. After applying their tool and correcting with the help of the recommendations, in three constraints all the violations were rectified whereas in the remaining one constraint 60% of the violations were rectified. This shows that their tool was very effective in reducing architectural erosion.

One of the most cited papers is “Assessing Architectural Drift in Commercial Software Development: A Case Study” by Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. In this paper, they have considered a two-year case study of software that was developed from scratch at IBM Dublin Software Lab. They say that software architecture is one of the most important attributes of system design. The main intent of the paper is to find Architectural Drift during software development and to evaluate if the detection leads to inconsistency removal. They have used an approach known as Reflexion Modeling which is used to find the architectural drift. Reflexion Modelling is an architectural recovery method that is used to extract the implemented architecture and can also visualize the differences in the planned architecture. They say that architectural drift occurred during the redevelopment of the system even when the developers had a clear vision of how the system had to be implemented at the architectural level. The inconsistencies were due to the oversight of the original design architecture, dependence on legacy code, and misplacing or duplication of methods or constants. Even though inconsistency identification was possible, the inconsistencies were not removed. There were many reasons for this. One of the reasons was that there would be a ripple effect i.e while removing a minor inconsistency the whole project could stop working (Rosik, 2011).

6.3 Discussion Based on Publication Venue

In Figure 6, we see that out of many different publication venues, some of the prominent ones are the IEEE International Conference on Software Reliability Engineering Workshops (ISSREW) with over twenty-four papers published. The International Symposium on Software Reliability Engineering (ISSRE) with around ten papers published, the IEEE International Workshop on Software Aging and Rejuvenation (WoSAR) with about seven paper published, and Software Quality Journal with about six papers published where most of the work related to the field of Software Aging is published or curated.

The year 2020, had the highest number of papers related to Software Aging and Rejuvenation published. From the year 2013, different venues apart from the ISSREW, ISSRE, WoSAR, or Software quality journals also started publishing papers related to software aging and by the year 2020 many other different venues like the International Journal of Electrical and Computer Engineering (IJCE), the International Conference on Computer and Applications (ICCA), the IEEE International Conference on Systems, Man and Cybernetics (SMC), Journal of Information Technology Research (JITR), etc. also published papers related to Software Aging.
The International Symposium on Software Reliability Engineering (IEEE ISSRE) was first held in 1990 in Washington DC. It is an academic conference that covers the reliability engineering of software. The symposium runs usually for 4 days with strong industry participation and has also been held in over 18 cities outside The USA. The main properties of interest are reliability, safety, security, availability, and quality of the software. The symposium later started a multi-track program where it integrated workshops and tutorials like IEEE International Conference on Software Reliability Engineering Workshops (ISSREW). These workshops provide further added opportunities for exchanging information and collaborating. The main aim of these workshops is to discuss software challenges and research development in the early stages.

The IEEE International Workshop on Software Aging and Rejuvenation (WoSAR) is an international workshop that is usually co-located with The International Symposium on Software Reliability Engineering. The workshop provides a forum for practitioners and researchers to discuss recent advances or discoveries in the field of software aging and rejuvenation and to present their work on the practical and theoretical characteristics of software aging and its mitigation using software rejuvenation. Software Quality Journal is a journal published by Springer New York. First published in 1992, this journal has seen a rising impact since 2011. The main aim of the Software Quality Journal is to promote awareness of the crucial role of quality management in the effective construction of the software systems developed, used, and maintained by organizations in pursuit of their business objectives. It also provides a forum for the exchange of experience and information on quality management and the methods, tools, and products used to measure and achieve it. Finally, it provides a vehicle for the publication of academic papers related to all aspects of software quality.

In figure 7 we can see some of the most eminent venues of publication are European Conference on Software Architecture (CSA) with about three papers published, Working IEEE/IFIP Conference on Software Architecture (WICSA) with about two papers published, Association for Computing Machinery (ACM) with around two papers published, and Journal of Systems and Software (JSS) with only one paper published. There were also two papers published from other sources such as textbooks, articles, and handbooks.

7. Discussion and Answering to the Research Questions

In the previous section, we discussed the top-cited papers and the research that has been conducted in the domain of software aging and architectural degradation. We identify that the problems causing those phenomena are similar and tend to have homogeneous consequences. For instance, many researchers (Castelli et al., 2001) believe that multiple factors cause different types of software failures and some of the main contributors are data corruption, storage space fragmentation, exhaustion of system resources, unterminated threads, numerical error accumulation, bloating and leaking of memory, and unreleased file locks. These failures can cause performance and system degradation failure gradually over a period of time. This type of degradation is termed software aging.

Similarly, as part of the evolution of the software, it has to adapt to newer technologies and updated requirements while updating and modifying the software to the current needs. During the modifications, the design principles and the system architecture are ignored which leads to a gap between the conceptual and the concrete architecture. For instance, Rahman et al. found that the conceptual and the concrete architecture are fairly different in the Google Chrome web browser.
This gap can grow over time and become unmanageable, complex, and expensive to maintain. This type of degradation is known as architectural degradation.

Software aging and architecture degradation complement each other. One of the main reasons that a system declines due to software aging is because changes are made to it without sufficient documentation and comprehension of the code. A system that is prone to architectural deterioration is also prone to software aging eventually. As mentioned earlier, during the lifetime of a software, it undergoes several changes and updates at several intervals. These changes are done to allow the software to adapt to new requirements and for the software to provide functionalities with the advancement of technologies and the latest customer demands. However, the changes to the software do not always undergo proper planning and are done ignorantly without accounting for repercussions that the software might have in the future.

During a minor modification or update in the software, changes are not made to the existing architecture to incorporate them. Assuming that these changes can be documented during the next phase or release, the documentation will be either forgotten or ignored. As a result, gaps are then created between the intended architecture and the existing architecture. These minor changes which are unaccounted for might cause errors or failures. Whenever a developer is trying to make new changes, they might find it difficult to understand the current software system, and will be inefficient as the implementation is different from the documentation. It is also difficult to understand by just debugging the code as it is very complex and may contain a large number of files or dependencies. Without a proper understanding of the system, the modification may add more errors and failures causing a system-wide disruption. Documenting and understanding the architecture is very important. By regularly updating the documentation and structure of the software architecture, the system can be designed to be susceptible to change and can be modified to reduce the effects of aging and mitigate architectural degradation while also implementing features and functionalities that are not prone to additional failures or bugs.

We can also rejuvenate the system which increases the sustainability of the software for a longer period. Additionally, planning and designing beforehand helps to identify the problems that may occur before implementing it. When ignored, the aging of the software accelerates and the software becomes obsolete in nature.

7.1 Answer to the Research Question RQ1

As we discussed in Section 6, the research on Software Aging has begun in 1994 and the study has increased gradually since the late 1990s and early 2000. The popularity of Software Aging research works was steadily increasing until 2009. Since 2010 we observe a very high number of research works being published and a number of new venues are created for Software Aging studies. Even though there was a downfall in 2015 due to the reduced number of conferences, there still was a gradual increase in the research conducted. Depending on the number of prominent venues that held conferences that year, there was a significant rise or fall in the publications in the domain of software Aging. Even though there has been instability in the number of publications each year, the domain of software aging has been continuously growing to date.

7.2 Answer to the Research Question RQ2

The research papers we studied have discussed the possible reasons behind software aging. There are many factors identified by the authors that are affecting software quality, maintainability, architecture, and causing software aging. We observed or inferred the factors if they have been identified or discussed by the researchers as one of the responsible factors for either “Software Aging”, or “Architecture Degradation” or both.

We cited the papers where a particular factor has been explicitly mentioned as one of the responsible factors. We are calling it “inferred” when a factor has been indirectly inferred from the other factors. For example, Vaidyanathan and Gross (see Table 4) explicitly mentioned that data corruption is one of the factors causing Software Aging while they have also discussed at the same time that Software Aging happens eventually from Architecture Degradation which indirectly infers that data corruption causes architecture degradation eventually causing software aging.

Table 4 shows that there are several factors contributing to Software Aging as well as Architectural Degradation. The most frequently discussed factors by researchers include Code-Complexity and Maintainability, Data Corruption, Exhaustion of System Resources, Memory Bloating, Memory Leaking, Numerical Error Accumulation, Storage Space Fragmentation, Unplanned System Changes, Unreleased File Locks, and Unterminated Threads.
Table 4. Factors contributing to Software Aging and Architectural Degradation

<table>
<thead>
<tr>
<th>Factor/s</th>
<th>Software Aging</th>
<th>Architectural Degradation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Corruption</td>
<td>Observed (Vaidyanathan &amp; Gross, 2003) (Cotroneo, 2010)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Exhaustion of System Resources</td>
<td>Observed (Li, 2002)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Memory Bloating</td>
<td>Observed (Cassidy, 2002)(Cotroneo, 2016)(Cotroneo &amp; Iannillo, 2020)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Memory Leaking</td>
<td>Observed (Cassidy, 2002)(Cotroneo, 2016)(Cotroneo &amp; Iannillo, 2020)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Numerical Error Accumulation</td>
<td>Observed (Li, 2002)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Storage Space Fragmentation</td>
<td>Observed (Garg, 1998)</td>
<td>Inferred</td>
</tr>
<tr>
<td>Unreleased File Locks</td>
<td>Observed (Garg, 1998)(Vaidyanathan and Trivedi, 2005)</td>
<td>Inferred</td>
</tr>
</tbody>
</table>

Table containing the factors that have been identified by the researchers contributing to Software Aging, also factors that have been identified by the researchers contributing to architectural degradation.

7.3 Answer to the Research Question RQ3
We observe that Software Aging is a big concern for software systems. The aging of software and the deterioration of architecture are inextricably linked. Hence, we can determine that one of the major contributors to Software Aging is Architectural Degradation. Both of these can be mitigated to some extent by upgrading the architecture regularly to reflect system changes and modifications. When modifying or updating a software system, the changes must be planned and implemented in the architecture. It must also follow architectural design guidelines, which makes the software system more practical, viable, and sustainable in the long run.

7.4 Answer to the Research Question RQ4
One of the major challenges in the area of Software Aging is that it occurs in the software system over a long period of time and hard to see symptoms when the system is young. The moment the symptoms start, they are very negligible and on many occasions, developers ignore them and do not care for taking a big initiative to revise or investigate the current architecture of the system. The researchers also face difficulties producing real-time scenarios to identify all the factors that contribute to the problem.

8. Future Work and Conclusion
This study aims to provide insight into the current trends in the research and practice of software aging. Particularly as it relates to the problems, publication trends, and factors that contribute to software aging. We also provide a critical perspective on the relationship between software aging and architectural degradation. We believe that insight into the problems, trends, and contributing factors to software aging would help practitioners to design effective solutions. We found that the more our practitioners are growing with software development, the more researchers studying the phenomenon of software aging. In many cases researchers tend to intertwine the terms software aging and architecture degradation, in many cases, researchers tend to consider architecture degradation as a path to aging. Many factors have been identified in the literature that accelerates software aging and degrades the architecture at the same time. However, from this study, it is very clear that architecture degradation is a shadow of software aging. Many of the factors mentioned by researchers eventually degrade the architecture and when the architecture is degraded, software becomes unwelcoming to the new changes and at some point it becomes unmaintainable.

The takeaway from this study is that besides many other factors, software architecture is also one of the prominent factors that expedite software aging. Although practitioners take lots of initiatives to improve code quality, avoid data leaks, data corruption, and refactor code to avoid code complexity. We observed that revisiting architecture at a regular interval is a necessary task to watch for architecture degradation and forecasting software aging. Our future work as a follow-up to
this study will build a tool to visualize the current architecture of a software application at run-time. We will quantify the architectural drift by comparing the previous versions of the software system so that developers understand not just the difference between the conceptual/initial architecture and the current architecture but also they will be able to measure how much the architecture has drifted from the previous version. Additionally, the user can find the impact of the newly added changes they made to the architecture. This will help document the system better and mitigate Software Aging and Architectural Degradation.

References


Heinkens, R., & de Graaf, O. An introduction to software aging: How to deal with it? Computing Science University of Groningen, pp. 44.


Appendix A

Research Papers Used in the Survey


A systematic literature review. *IEEE Access*, 8, 173681173709.


Yakovyna, V., & Uhrynovskyi, B. (2020). *User-perceived response metrics in android os for software aging detection*. In 2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT), volume 1,


Copyrights

Copyright for this article is retained by the author(s), with first publication rights granted to the journal.

This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).