A Systematic Review for Sustainable Software Development Practice and Paradigm

Varun Dixit[1], Davinderjit Kaur[2]

[1]Omnissa LLC, Palo Alto, CA
[2]University Of Alberta, Edmonton, AB

Received: 13.04.2024 Revised : 11.05.2024 Accepted: 25.05.2024

ABSTRACT
Sustainability is essential to current software development in order to guarantee the lifespan and quality of software systems. The present study delves into the many aspects of software sustainability, emphasizing architectural choices, sustainability measurements, and the use of software in scholarly investigations. It presents the idea of "sustainability debt," which results from poor design decisions and lowers the calibre of software. The significance of gathering and organizing architectural knowledge (AK) is covered in the article as a means of reducing this debt and enhancing decision-making. The paper discusses metrics for code complexity and maintainability as well as architectural metrics for modularity and design smells that are used to assess software sustainability at various abstraction levels. It also looks at new architectural knowledge measures that evaluate how long-lasting design choices are and how they affect the development of software. The study also emphasizes the importance of software in research and the problems associated with a lack of recognition and training in this area. It examines programs like the Software Sustainability Institute's Open Call for Projects and its endeavours to advance software practices and talent development. To sum up, attaining software sustainability requires a thorough strategy that incorporates strategic architectural choices, efficient use of metrics, and continuous enhancements in software processes. The study urges ongoing efforts to tackle these issues in order to maintain software systems' dependability and support long-term objectives for research and development.

Keywords: Software Sustainability, Architectural Decisions, Technical Debt, Metrics for Sustainability, Code Metrics, Architecture Metrics, Software Development in Research, Architectural Knowledge, Software Quality, Research Software Engineering

1. INTRODUCTION

Sustainability is the ability of development to meet existing requirements while also allowing future generations to meet their own needs. Within the field of sustainable software engineering, sustainability is understood to mean creating software for long-lasting systems that can accommodate the needs of both current and future generations while integrating the three pillars of sustainability—environment, economy, and society—in order to meet deadlines [1].

Sustainability development has been defined by a number of scholars, each with their own interpretation, point of view, and definition. [2], for instance, concentrate on how IT alters behaviour that impacts society and the environment in a way that is more sustainable. The concepts developed by [2] were expanded upon by [3] by adding the aspect of positive and negative influence on sustainable development. Next, in order to achieve sustainability, [4] added the components of ethical use of financial, human, and ecological resources. Subsequently, [5] added that resource usage can enhance the energy efficiency utilized in software product production, leading to sustainability.

The first study to look at this was [6]. They found that the sustainability of a long-lived software system depends on how well its resources are used. Later, in order to determine the qualities that would be utilized as indicators to assess the sustainability in software products, [6] take into consideration the composite and non-functional requirement that is employed in software quality standard. [5] concur that this concept relates to the quality attributes component of the software sustainability assessment. Researchers have used all of these concepts to develop their own definitions and interpretations of sustainability in software engineering, incorporating the original goal of sustainability to fulfil current requirements as well as those of future generations.

A. Overview

The definition of sustainability is "to be a sustainable action" and it emphasizes the idea, scope, and methods of sustainability development. The Bruntland Commission introduced sustainability development in 1987 with the goal of enhancing everyone's quality of life on Earth by using natural resources beyond what the environment can sustain without causing harm. In order to meet the requirements of both the sustainability definition, which states that needs must be met for both the present and future generations, as well as the necessity of finding creative ways to change institutional structures that indirectly affect individual behaviour [7]. This indicates that action must be taken to alter practices and policies at all levels, from the individual to the global. Since the 1980s, the notions of sustainable development have advanced quickly, with a strong emphasis on the environment, society, and economy as the three pillar dimensions. The concept of sustainable development is not new; it has been applied to the nation's growth in a number of areas, including manufacturing, building, disaster recovery, soil and erosion, ecosystems and biodiversity, and so on.

Since its inception in early 2010, software engineering has acknowledged sustainability as an important concern. The use of software systems started in the new development era, when sustainability ideals were first applied to software development [8]. The majority of software processes and products are created with the intention of making money, whereas the development of current software systems is typically done with no intention of making money. Additionally, software development procedures and products prioritize social benefit over environmental considerations. These development trends are some of the reasons why software engineering needs sustainable development to solve the issue and incorporate all aspects of sustainability as a single, cohesive team [9].

B. Sustainability in Software Engineering

Numerous publications in the form of comprehensive literature reviews on various software engineering subjects, including the relationship between sustainability and software engineering, can be found in the literature. The goal of this study is to present a review of the literature on sustainability in software engineering [10]. It includes studies on sustainable development initiatives, limitations in previous research, methods and methodologies employed, and case studies that are currently available. The inquiry will concentrate on the form of sustainable development that software engineering has created since this field's introduction.

According to the investigation's findings, a number of scholars have developed their own theories on sustainable development in software engineering through a variety of contributions, including framework models, strategy models, requirements engineering techniques, method models, and goal models. A framework for sustainable software engineering was created by [11] and offered as a way to direct the creation of sustainable processes and products. A green strategy model was presented by [12] to help company strategy decision makers. The researcher offered a more comprehensive perspective on software engineering sustainability. Theoretical work by [12], a specialist in the field of corporate organization and quantitative goal modelling methodologies, was employed by [13]. [13] employed the suggested concepts to manage software engineering's environmental sustainability.

The requirements engineering methodologies were first suggested by [14] as a means of achieving sustainability in software engineering. They offered a blueprint for incorporating sustainability into the process of developing software. The concepts started by [14] were most recently carried out by [4] in their requirements engineering for sustainable systems. According to the research, requirements engineering has a significant impact on future hardware and software systems, thus it is important to write requirements carefully in order to guide the development of more sustainable new living systems.

[4] made a contribution to the other paper about requirements engineering as a means of supporting environmental sustainability. The researchers have detailed a set of guidelines about sustainability and the environment in their most recent article. These guidelines are intended to facilitate the creation of ICT systems by integrating environmental informatics knowledge into an appropriate requirement engineering approach. [15], however, have persisted in their attempt to put forth a detailed guideline regarding social sustainability in requirements engineering. The concepts put out by the researcher are founded on the social effects of software systems, which touch on a wide range of topics that are either directly or indirectly related to social sustainability. There aren't any publications that address economic sustainability, though.

The majority of the research' conclusions concentrated on software sustainability creation and management. Too few articles, meanwhile, have been contributed to the examination of software sustainability. None of them have thoroughly examined the metric evaluation for every feature and sub-attribute that has been identified for the purpose of measuring the sustainability standard. As a result, the concept of mandating and rationalizing the basis for software sustainability assessment will be examined in the upcoming work as part of SLR.

2. Software And Sustainability

Software sustainability needs to be comprehended before it can be quantified. The term "sustainability" in contemporary English describes a system's "ability to endure." The Latin root of the word sustainer meant to both survive and uphold, to provide means of support for [something]. This shows that the ability to sustain and longevity as a measure of time are important components at the core of comprehending sustainability. There is a tight relationship between these two ideas. The term "need," which is essential to this definition, refers to both evolution and the acknowledgment of shifting stakeholder requirements in the present as well as the future.

Regarding software, there are at least two different points of view on the subject: software engineering for sustainability (SE4S) and sustainable software. Software endurance, also known as technical sustainability, is the emphasis of the former, whereas the latter addresses issues external to software systems in an effort to promote sustainability in one or more aspects [16]. A five-pronged approach including environmental, economic, social, and technological considerations is advocated for in the Karlskrona Manifesto. Recognizing software sustainability as a new and important subject, it takes into account both viewpoints. These dimensions have the following definitions:

- The economic dimension is concerned with capital, assets, and added value, which includes revenue, capital investment, prosperity, and wealth development.
- In the environmental dimension, we talk about the long-term effects of human activities on the natural world, including things like ecosystems, resources, the weather, trash, pollution, and so on.
- The term "individual dimension" describes how well people are as persons, which encompasses things like mobility, agency, education, independence, self-respect, mental and physical health, and so forth.
- The social component addresses the elements that undermine social trust as well as societal communities, which include person groups and organizations. The ideas that are examined here include democracy, employment, social justice, and equity.
- System upkeep, data integrity, and obsolescence are all parts of the technological dimension. Included in this concept is the notion of the useful life of data, systems, and infrastructure, as well as how these things should adapt to new circumstances.

However, there are interdependencies among these dimensions, and for a system that is being studied, trade-offs may need to be made. Take a car sharing system, for instance, which consists of a fleet of shared private vehicles, an application that runs on both the client and server sides, allowing users to interact and plan rides, and a server that stores all of the necessary data. With respect to the five factors, we can determine the following specifics:

- **Economic:** Users can save money by offering rides in their cars or by sharing rides instead of buying a car, which can add up to savings for the user community. The sustainability of the service is contingent upon its economic viability, meaning that it must generate adequate revenue streams to sustain its operations. A software system's development, maintenance, and operations cost-effectiveness will be impacted by decisions like adopting architectural patterns to prevent technical debt and using open-source components.
- **Environmental:** Energy is needed by both IT systems and automobiles, which has an effect on the environment through pollutants, for example. Additionally, each hardware item has a lifespan that requires it to be made, acquired from someplace, maintained, and finally recycled or disposed of. Sharing systems frequently result in lower overall resource utilization, which lessens their impact on the environment.
- **Individual:** Individual users may gain by having access to personal mobility and from having a stronger sense of responsibility associated with eco-friendly behaviour.
- **Social:** Depending on the system and its workings, a new user community focused on assisting one another in selecting less carbon-intensive forms of transportation may emerge, or social ride-sharing networks may deteriorate as gradually developing human connections are supplanted by routing algorithms.
- **Technical:** Over time, both the cars and the connected IT system require maintenance. Longevity of the system will depend on a number of aspects, including technical debt, the architecture's capacity for evolution, and the lifecycles of supporting technologies.

Relationships between particular dimensions appear immediately; nevertheless, each of these dimensions, as well as the ways in which they intersect, need to be examined in terms of their implications for a long-term system vision and the means by which this can be guaranteed [17]. The scientific community is becoming more and more conscious of the need to shift toward a more all-encompassing understanding of sustainability that takes these various aspects into account. Furthermore, the effects of these five aspects on sustainability appear in three orders of influence, which are as follows:

- Third-order impacts, such improved urban air quality, less problems with downtown parking, etc., result from widespread, long-term system usage. The development, production, installation, and use of the software and hardware components of the car-sharing system all constitute first-order effects.
- A second-order effect occurs when the system's usage leads to different kinds of behaviour or expectations from the first-order system, like users coming together to form communities and reduce their individual environmental impacts. Finally, a third-order effect occurs when users form communities and reduce their individual environmental impacts.

Divergent opinions exist within the software engineering community over the definition of sustainability, which contributes to its lack of widespread understanding in the profession. A formal definition of software sustainability, however, has been proposed in other contributions; this definition focuses on the longevity of the software system. As a result, the term has been used to describe a desirable quality of software or a first-class non-functional need. One example of a composite non-functional necessity is software sustainability, which is defined as "a measure of a system's extensibility, interoperability, maintainability, portability, reusability, scalability, and usability," according to [5]. The idea of the software system's evolution is directly tied to a number of the metrics. The justification for incorporating usability as a sustainability statistic is its direct correlation with perceived utility as viewed by stakeholders, which consequently links sustainability to the issue of need. Furthermore, a number of the quality criteria indicate the "effort required" to reach a specific result. This implies that sustainability is closely related to other quality criteria including cost and energy efficiency, resource utilization over the course of the software's lifetime, and alignment with the environmental and economic components of sustainability. In the realm of software engineering, a consensus on what sustainability means is still developing, hence further investigation is needed to support or contradict this viewpoint. It is argued that context, such as that suggested by [6], and (social) structure are necessary for the idea of sustainability, in addition to the simultaneous evaluation of multiple interrelated elements of sustainability. Similarly, it is stated that the goal should be to define how various cultures use the terminology in order to have a shared and common understanding, rather than aiming for broad conformity of definitions.

Software sustainability evaluation techniques and reference models for developing such software have dominated the literature on software engineering. Sustainable software, defined as a system that requires less maintenance work to modify or consumes less energy while functioning, saves resources. Several reference models have been presented in the area of software engineering and sustainability to design such software. For example, the GREENSOFT model was put up by [6] as a conceptual reference model for software that includes sustainability standards and measurements, software engineering extensions, and a product life cycle model for software products from birth to death. Software systems are considered throughout the model's lifecycle, from creation to deactivation and disposal, and it incorporates both short-term and long-term sustainability metrics and measures. [18] put forth a development model that incorporates a set of criteria to assess the environmental sustainability of every software engineering step, with the goal of promoting software engineering that is both ecologically sustainable and socially responsible. Although the primary focus of these approaches is environmental sustainability, they also recognize that other sustainability dimensions, like those suggested by [19], several elements need to be taken into account in order to ascertain the impact of software systems on sustainable development. Making use of a generalized sustainability reference meta-model with examples drawn from specific software systems and processes, they offer a methodology to assist software developers in comprehending the interconnectedness of environmental sustainability with individual, social, economic, and technical aspects of sustainability.

Apart from the creation of several reference models for sustainable software, other methods have also been suggested for assessing software sustainability. These methods centre on assessing the lifespan of software systems. In order to model and integrate stakeholders' sustainability issues, [20] put out a structure that may be used as a classification system for sustainability. When creating software systems, this framework may help you consider different design approaches that might affect sustainability. Although it's unknown how far this method can be applied outside the case study that served as the basis for the taxonomy's development, it offers a helpful starting point for investigating its limitations and generalizability. [21] focuses on ensuring a system's economic viability throughout its lifespan, emphasizing sustainability from an economic perspective. They introduce TechSuRe, a method that integrates sustainability considerations into software development by assessing time, cost-benefit, and risk factors. One approach involves evaluating "sustainability risk," which considers nine factors, including market risk, production lifetime, and technology evolution. This evaluation projects the expected financial viability of the technology. [22] further develops this by using scenario analysis, architecture compliance, and metric tracking to assess architecture sustainability in two ways: (i) designing sustainable systems and (ii) enabling software architecture to evolve without compromising its core structure. This multi-perspective method helps to avoid architecture degradation and to track changes in requirements and technology. For various stakeholders, including managers, architects, and programmers, [23] created a library of sustainability principles that covered the software development life-cycle from system design to maintenance. The guidelines include software engineering approaches, strategies, and instruments to improve system lifespan in an economical manner. Both strategies ignore the meaning of various sustainability characteristics in favour of concentrating on the technical and financial aspects of sustainable software systems. Furthermore, elicitation and modelling of sustainability requirements has been the subject of other relevant work on sustainability in the field of requirements engineering and software.

In this paper, we identify the five dimensions that express pertinent issues and offer a framework for the concepts and indicators needed to comprehend the resilience of real-world cyber-physical and socio-technical systems. More precisely, we centre the conversation on two facets of sustainable software design. To achieve this, we narrow our attention to three ideas:

1. **Software sustainability:** The community and the running organization will be concerned about how long the software-intensive system can last.
2. **Software architecture sustainability:** In turn, the longevity of that software system depends on its architecture and their flexibility. This is sometimes referred to as architecture sustainability (i.e., the extent to which the software system's architecture permits ongoing upkeep and development over time without necessitating significant and costly restructuring).
3. **Sustainable Software Architecture: Design Decisions:** There's a growing emphasis on decision-making as this architecture mirrors the basic design decisions that arrange the system and its parts. Because architectural judgments have far-reaching consequences and are expensive to change, architects are fundamentally concerned with the decision's legitimacy. The sustainability of software architecture design decisions has been used to characterize this idea.

3. Software Architectures and Sustainability

Software systems' ability to function over the long term, maintain efficiency, and evolve appropriately in a constantly changing execution environment is strongly influenced by their architectural design. Software architectures are essential to a software system's ability to survive and develop, according to [24]. The ability of an architecture to withstand modifications brought about by changes in the environment, requirements, technologies, business strategies, and goals over the course of software system life cycles is known as architecture sustainability. Nonetheless, architectural drift and erosion are two connected phenomena that undermine the sustainability of any system architecture. When the source code deviates from the intended architecture, architectural erosion frequently occurs. Architectural drift, on the other hand, is defined as a system's code deviating from the underlying architecture. Both issues can occur during the system's evolution and maintenance cycles and are the result of haphazard, unintentional additions, deletions, and modifications to architectural design decisions. Architecture erosion and drift can result from a variety of causes, including a build-up of poor or subpar design choices and issues with communication between the development and design teams. Despite growing in importance over the last fifteen years, as a distinct subfield of software engineering, software architectures is only starting to gain traction in the industry [25]. In spite of software architectures' vital role in software system design, the subject of software architecture sustainability has only lately emerged as a separate discipline of study. Investigating the function of architectural metrics and technical debt in gauging the sustainability of architectural designs has been the main emphasis of this.

One way to make flexible software architectures in advance is to create them in an open source style that can accommodate more changes in the future without drastically changing the system's core, sustainable software architectures at a low cost. Adherence to recognized design principles—such as conceptual integrity and the separation of concerns—as well as avoiding making bad evolution decisions are crucial for this [26]. Templates for designing sustainable architectures can be found in the rise of software reference architectures. These architectures represent the architectural knowledge of structures, elements, and relationships of many successful architectural implementations, though they are limited to a specific domain or family of software systems. Continua for healthcare IT, SOA-related OASIS Standards, and IBM's Service-Oriented Solution Stack (S3), AUTOSAR for the automotive industry, and the most current Industrial Internet Reference Architecture (IIRA) are examples of well-known reference architectures. For instance, standardization, knowledge reuse, interoperability facilitation, and improved communication among interested parties (car manufacturers, suppliers, and other businesses in the electronics, semiconductor, and software industries) are just a few of the major advantages that AUTOSAR has brought about [27]. However, many reference architectures do not sufficiently or clearly consider sustainability. Reference architectures function at a higher degree of abstraction for a collection of systems in particular areas, whereas software architectures make up the design solution for particular systems. Both architectural styles incorporate aspects of reusable knowledge about important design choices that support standardization and the long-term viability of tested solutions. Their durability is one of the most obvious and conspicuous signs of these architectural solutions' effectiveness. This is contingent upon their capacity to identify signs of degradation and their resistance to design deterioration. In order to handle this, AUTOSAR has an update strategy that includes version control and release management for its documentation. This is because each document is updated continuously, incorporating elements from several releases. Figure 1 displays the major AUTOSAR evolution stages and releases.

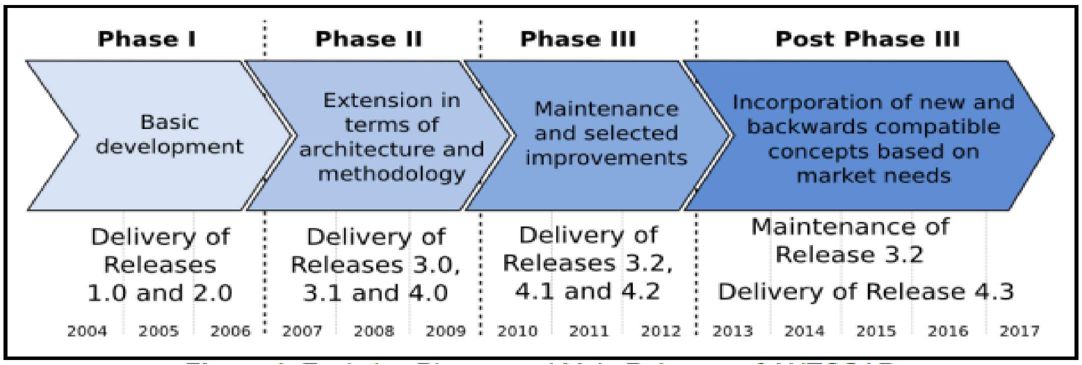
Phase I
Phase II
Phase III
Post Phase III
Basic development
Extension in terms of architecture and methodology
Maintenance and selected improvements
Incorporation of new and backwards compatible concepts based on market needs
Delivery of Releases 1.0 and 2.0
Delivery of Releases 3.0, 3.1 and 4.0
Delivery of Releases 3.2, 4.1 and 4.2
Maintenance of Release 3.2
Delivery of Release 4.3
2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017

Figure 1. AUTOSAR Evolution and Key Releases [28]

Figure 2 illustrates how new versions are continuously delivered by other reference architectures as well. Even while improvements and extensions are included in new editions, this has led to a notable rise of documentation. This is comparable to IIRA.

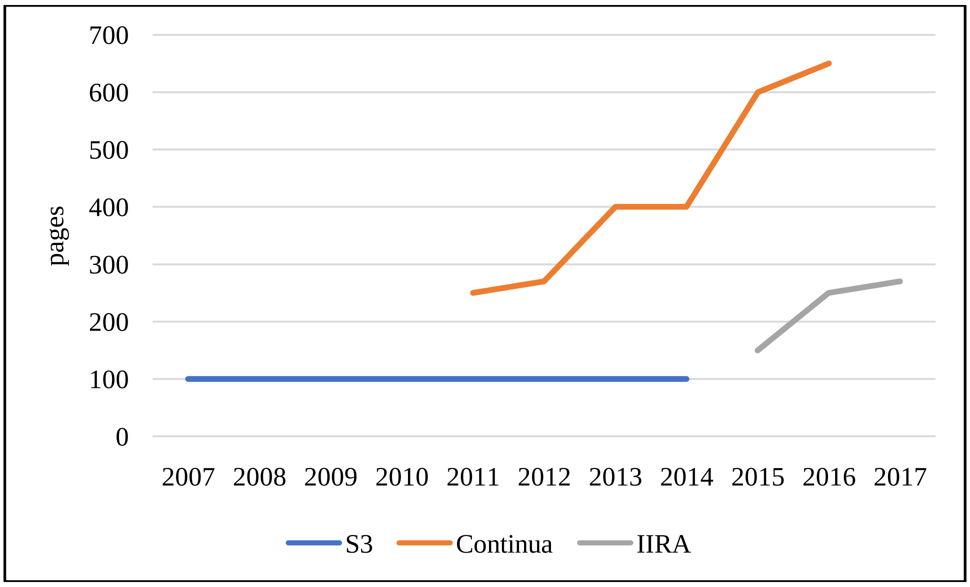

Figure 2. Revisions to the Reference Architectures [29]

Despite the many reference designs proposed for various application areas, many of them have either not been adopted or have not been maintained. For example, the Service Oriented Paradigm was the primary focus of [30] examination of sixteen reference designs for systems development. The findings revealed that thirteen of them were completely inactive, with no website, initiatives, or related papers to be found. Having said that, reference designs should be flexible enough to integrate extra design choices that stick around for when new challenges arise and these architectures need to be adjusted to fit new needs, such smart vehicle features. In a larger sense, some of the main elements that contribute to the sustainability of reference designs include:

- For an architecture to be considered long-lasting, it must:
- Be consistent with other technologies in the target domains (e.g., communication protocols, domain standards, and architectural styles);
- Get updates and releases often;
- Adapt to new requirements without affecting previous decisions;
- Have a community around it, which can be strengthened through collaboration with businesses, research facilities, and academic institutions.

4. Software Architecture Decisions And Sustainability

One facet of the umbrella term "sustainability debt" is the practice of tracking and reporting the ways in which technical debt impacts sustainability as a result of software design choices. This idea was discussed in [31], and it brings attention to the fact that previous design choices have an unintended effect on all five aspects of sustainability, including the monetary problem of long-term expenses. Thus, design choices have a big impact on how long systems and their architecture last. The quality of optimal design decisions alone, however, is not enough to ensure the sustainability of designs; other factors that must be considered include the economic, individual, social, and technical aspects of capturing those decisions, such as a lack of incentive or motivation, inadequate tools, architectural knowledge (AK) capture is not without its difficulties, such as the time and energy it takes, the fact that it interrupts the design flow, stakeholders' lack of comprehension, and the difficulty of deciding what data is relevant and valuable to capture (especially in Agile projects with minimal documentation). The massive and ongoing effort required to efficiently store massive volumes of codified information is believed to be the root source of the many challenges associated with gathering AK. This data may be difficult to manage and put to good use. It is therefore necessary to compile a lasting set of design choices that are easy to maintain and have practical applications. For instance, in order to minimize the effort required for AK documentation, [32] suggested use configurable AK templates to record a small number of important architectural design decisions. Sustainable decision-making, then, requires both lightweight, minimalistic approaches to reduce paperwork and emphasize key decisions and timeless, strategic knowledge to lengthen the life of systems and their designs. To circumvent this problem and enable designers to make greener architectural design choices, [33] published a meta-model and principles for creating flexible Architectural Knowledge Management (AKM) systems.

Their method introduces a flexible and configurable meta-model that addresses the rigidity of earlier approaches. It incorporates AKM tools for maintaining a minimal set of AK, configurable entities to extend and customize new AK, and metrics to assess the sustainability of design decisions. The method links quality attributes like timeliness, changeability, complexity, and cost to these metrics, ensuring the sustainability of the AK [34]. The authors of [33] propose updating the meta-model from [33] to include additional AK-related features by enhancing the sustainability model and its extensions.

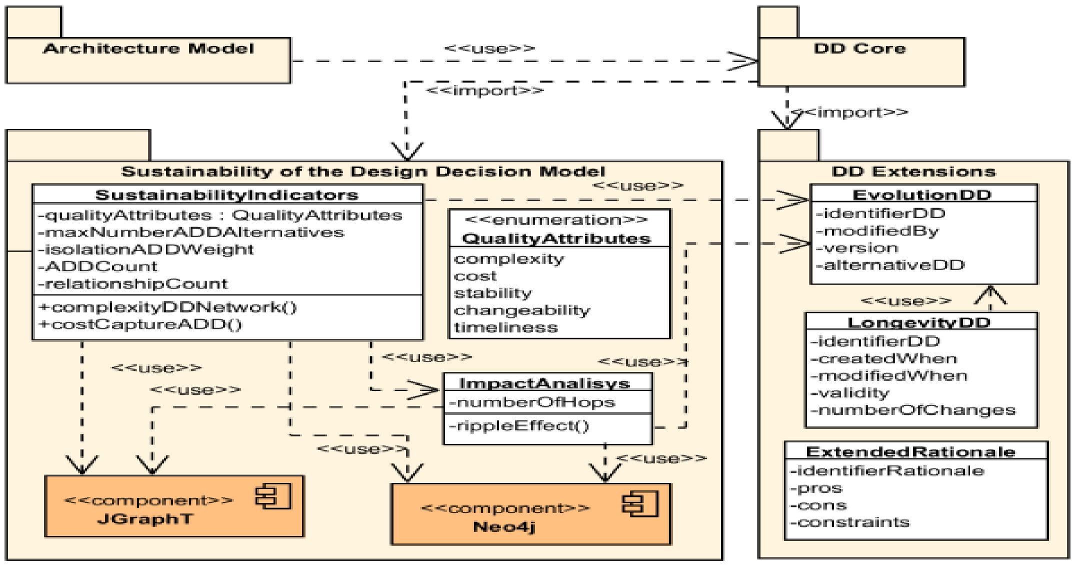

Figure 3. Enhanced Meta-Model for Long-Term Decisions and Sustainability Estimators in AK [35]

On the other side, better decision networks may be built with fewer AK items recorded, which means less effort and maintenance expenses. Some of the quality parameters prescribed by the Sustainability of the Design Decision Model package include the number of options and edges. One way to gauge a decision network's complexity is by looking at these characteristics. Ripple effect measurements are among the additional evolution-related metrics that are used to evaluate the effects of modifications and identify the decisions that are changed most frequently. Two new components, J-GraphT and Neo4j, were added to the Sustainability of the Design Decision Model package to visualize decision networks and assess sustainability based on network complexity. Similarly, [33] employed three architectural templates—short (seven items), medium (ten items), and long (fourteen items)—to evaluate the time, effort, and quality involved in documenting key architectural design decisions. Sixty-four volunteers in all were split into eleven groups at random during the experiment: two groups had five people and nine groups had six. Over a four-week timeframe, each group of three software architects—junior, senior, and cognitive—was tasked with documenting the major design choices made for a particular system. As part of our research approach, we conducted an exploratory case study to determine the potential outcomes and collaboration dynamics among stakeholders while using different AK templates to document architectural design choices. We omitted test and control groups since our goal was to estimate the participants' average effort in recording the AK using each template and the number of alternative alternatives they gathered. We were really evaluating a separate variable here. Since fewer options were easier to manage throughout evolution cycles, participants were more productive when given fewer to choose from, and the results showed that medium and short templates captured the AK better than long ones [36].
Groups using the shorter template spent less time overall than those using the medium or long templates, as seen in Figure 4. However, because group G1 collected a significantly higher number of decisions than groups G7 and G10, their team members spent nearly three times as much using the identical template. In comparison to groups that used the short template, those that used the medium template yielded the predicted results; nevertheless, it took longer for group G4 to record the same number of choices compared to groups G3, G6, and G11. The team members' inflated claims in several of the individual tests we administered to confirm the results are the root cause of this cognitive dissonance. Last but not least, group G5 performs abnormally well in comparison to G2 because its members worked less hard to record more decisions than G2 did. This discrepancy may have resulted from a test taker providing an inaccurate response.

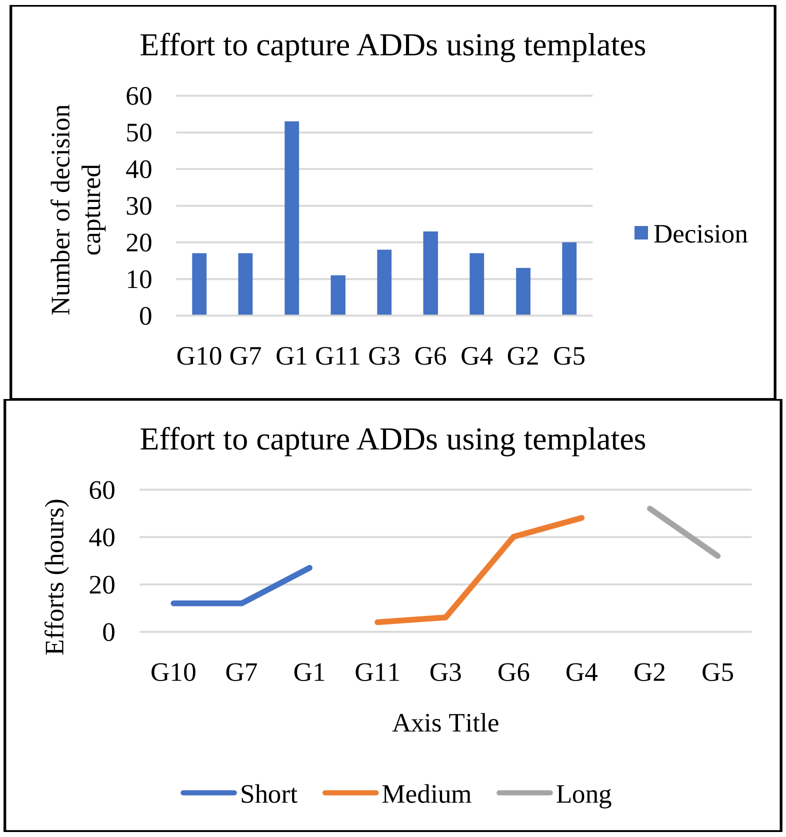

Figure 4. Using three distinct templates to record architectural design decisions for effort [37]

We evaluated the capturing effort by counting how hard each group worked to capture the different numbers of possible options. According to the findings, it took less work for groups to capture two to four design possibilities than for those to capture more than four. Consequently, an increase in the number of options for decisions led to an exponential rise in the amount of time needed to decide, consider, and record more options, which might have a significant effect on agile projects. Additionally, when comparing judgments with two or four possibilities to those with more than four, groups G2 and G10 had lower values for capturing effort. The overall capturing effort is reduced since fewer choices with more than four possibilities are captured compared to those with two to four, which means that the total number of decision points caught is less.

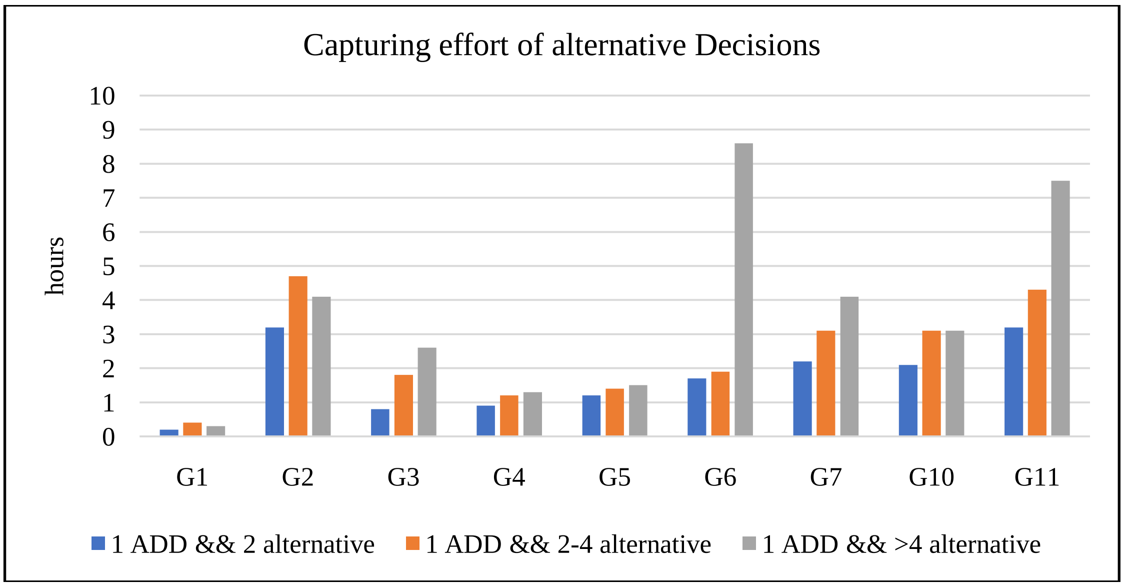

Figure 5. Counting the number of alternate decisions made by each group [38]

The entire amount of work required to capture decisions with varying numbers of alternative decisions is displayed in Figure 6. The findings show that the amount of time required for decision-making and evaluation activities grows in direct proportion to the number of choices examined.

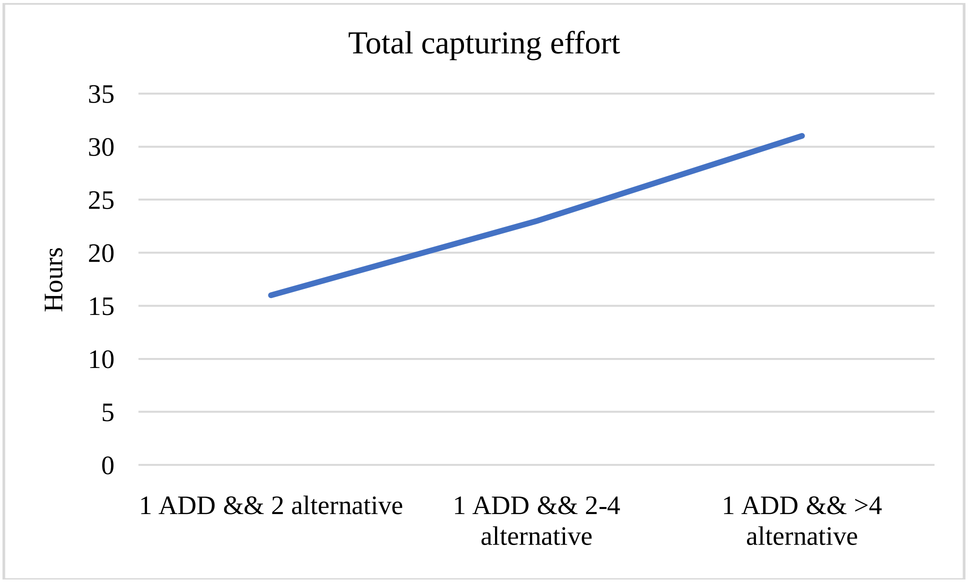

Figure 6. Total amount of time used to record judgments with various numbers of options [39]

Groups also used Enterprise architectural design patterns that were suitable for the task at hand, according to the results, indicating that a large number of the choices were well-founded in recognized knowledge and good technical practice.

Software architecture results from multiple decisions, but these design choices are often easily forgotten, increasing the expenses of upkeep and evolution as well as contributing to design erosion. It is often known that accumulating, disseminating, and repurposing AK can help reduce the loss of architectural knowledge. Even though the discipline of architectural knowledge management (AK) dates back to the early 1990s and has produced a variety of models, techniques, and research tools, the expense of gathering pertinent knowledge has prevented AK management from being widely used. Including decision-viewpoints and their justifications as premium components in architectural descriptions is essential to architectural sustainability. Understanding architectural design decisions is essential to the system's evolution since it allows one to gauge how sustainable the AK is during architectural modifications and estimates the amount of maintenance and documentation work that will be required when new requirements lead to new decisions.

5. Sustainability Estimation Metrics

It is necessary to evaluate a system's prospective quality loss using relevant measurements and indicators that can detect a decline in quality over the course of evolution cycles. Software developers have kept technical debt under control in order to lower the remediation costs associated with technical debt management, since the various dimensions of this debt have emerged as a quality indication of inadequate design decisions and coding techniques. Therefore, in order to quantify software sustainability in both design and code, we need to identify the origins and sources of the debt as well as those "hot-spots" in long-lived systems. Many methods have looked into modularity measurements as a technical debt indicator, however when the debt is not paid back, many other quality criteria are frequently impacted. In this section, we won't bore you with a laundry list of all the metrics that can be used for sustainability or other quality indicators; instead, we'll focus on highlighting the ones that can be chosen depending on quality goals to estimate technical sustainability from different angles and levels of abstraction.

Code and architecture metrics: The most widely used methods for finding "code smells" and improving system maintainability are software metrics, which assess system quality by looking at flaws in source code. Nevertheless, a wide range of code metrics are available to estimate various kinds of errors and poor programming habits that impact crucial quality aspects like complexity, maintainability, and reusability, among others. It is common practice to integrate metrics used to evaluate the complexity, coupling, cohesion, and dependencies across modules in order to estimate technical debt ratios and provide relevant indicators of code quality. More than forty criteria, including those related to evolution problems, were proposed by [40] as a means of estimating the software sustainability of software architecture. [14] subsequent research [2013] reduced the number of measures to twelve. In his work, [41] developed the idea of a "architectural bad smell" to describe design flaws like as anti-patterns, code odours, architectural incompatibilities, and bugs. Architectural refactoring is the consequence of a series of design issues, and the authors identify four of these smells. One of the major problems for system maintainability, according to the open-source systems research by Le et al., is to analyse the development of architectural degradation.

Architecture knowledge metrics: As a relatively new area of study, evaluating the long-term viability of architectural expertise and practice has only yielded a handful of measures. One possible architectural choice measure is the quantity of design issues addressed and the number of solutions investigated for each problem, as suggested in [42]. [33] provided an enhanced taxonomy of quality attributes and metrics for sustaining and improving design decisions.

Decision network complexity, they argue, may help estimate sustainability, with the amount of traceability links and design decision granularity playing a role. They also suggest assessing the work required to capture AK items, like the study's findings, in order to find out how much AK has to be collected so that the set of design choices can withstand evolution and maintenance cycles. In the end, the impact of redesign decisions is evaluated using metrics for change-proneness, instability, and ripple effect. The ability of a software system to endure the domino effect is one way that stability is defined in this context. [43] A design pattern's stability may be evaluated using instability and change-proneness metrics; classes' resistance to changes is a good indicator of how stable they are when involved in a change. This is a promising and unexplored area of research, as seen by these early attempts to predict AK's sustainability.

Aggregated metric sets: When estimating the sustainability of the system architecture, a combination of measures frequently produces more meaningful results in terms of more precise quality indicators. In a recent study, [42], for example, categorized software measures that could affect the estimation of design sustainability, ranging from modularization to volatility. The authors [44] provide novel measures (i.e., Architectural Smell Density, BDCC: Bi-Directional Component Coupling, and ASC: Architectural Smell

Coverage) and suggest different combinations of metrics to evaluate technological sustainability and identify when architecture begins to corrode or decay. They state that to assess the dispersed parasitic functionality as a maintenance concern that affects modifiability and reusability, it is suitable to combine Component-level Interlacing Between Concerns (CIBC) metrics with Concern Diffusion over Architectural Components (CDAC). This paves the way for the correlation between olfactory perception and building characteristics. However, not all tools provide fine-grained indications of technical debt and other quality qualities, thus it might be challenging to find the correct mix of measures.

In Table 1, we overview metrics for estimating and understanding system sustainability at various abstraction levels, which can be used individually or combined to align attributes with smells. The categorization is based on previous research that outlines the basic work on architectural level criteria [44,45,46]. If you want to estimate technical sustainability based on a given quality trait, you may use the metrics in Table 1 to figure out which ones to combine. We do not, however, offer precise correlations that show which tangible measures are used to quantify each unique quality trait. It should be noted that this table does not address discrepancies between various measures.

Table 1. Overview of software metrics for architectural and knowledge abstraction-level system sustainability estimation [47]

Architecture level metrics			
Category	**Smells**	**Metrics**	**Quality Attributes**
Maintenance	unclear and underutilized interfaces, depending on how large or tiny a module's functionality is; smells related to functional delegation	Index of Module Interaction API function use index, attribute concealing factor, Module Size Boundedness Index, Module Size Uniformity Index	Complexity, Modularity, Analysability, Effectiveness, Understandability
	Odors that impact redundant functioning and component coupling	Finding clones, Linking up with an item, coherence ratio of interaction, Quality of Modularization	Reusability, Complexity, Modifiability, Modularity
	Smells when one component implements an excessive number of worries, or when numerous components realize the same worry; Determine which components have an appropriate proportion of techniques.	Concern spreading about architectural elements interlacing between issues at the component level, number of issues with each part, Properly Scaled Methods Index	Reusability, Modifiability, Understandability, Modularity
	Determine whether components have cyclic dependencies, cross-layer relationships, and an excessive number of dependencies.	Layer Organization Index, API Function Usage Index, Cyclic Dependency Index, Dependency of cumulative components, overly intricate structure	Modularity, Understandability, Changeability, Modifiability
Maintenance	Other pervasive odors that might harm any area of the building	Coverage of architectural odors, Density of architectural scent	Cost
Evolution	Components that change too often	number of components that a change affects, Rigidity, the ripple effect, separation from the main sequence, Index of Module Interaction Stability	Stability, Evolvability
	Probability of components evolving in tandem	Component of a bidirectional coupling	Complexity, Evolvability
Architecture Knowledge Level Metrics			

Maintenance	An excessive number of choices and trail connections	EdgeCount and NodeCount	Stability and Complexity
	There are too many Architecture Knowledge (AK) pieces and options for decisions.	AK capturing effort cost	Price
Evolution	A shift affects a lot of choices.	Instability, change proneness, and ripple impact	Variability, Consistency,
	Too many changes and outdated choices	Volatility of decisions	Accuracy

5. Sustainability In Academia

Software is becoming more important to modern society. With the shift towards data-intensive, large-scale computational science and engineering, software is now fundamental to the expansion of human knowledge. Software's growing relevance in research has led to requests for its classification as a first-class scientific tool. However, as seen in [48] (where 59% of respondents deemed software to be essential), software is relevant to research, 56% of researchers generated code and 21% had no software development expertise. Thus, this raises major concerns about the software's quality, the research output's legitimacy and validity, and the sustainability of vital codebases for research communities.

Metrics that measure software architectural sustainability reveal several challenges. Expertise in developing software artifacts is necessary for making appropriate use of these indicators, comprehending them, and identifying and fixing sustainability issues. Developing, deploying, and sustaining reusable software presents several challenges, yet scientific progress relies on accessible and high-quality software at all levels. The significance of creating and updating software to back up an increasing amount of research has been emphasized by academics at all levels, from undergraduates to senior faculty and research leaders. However, there is growing recognition that the education and research pipeline lacks software development skills.

In October 2016, a Knowledge Exchange Workshop on Research Software Sustainability brought together major European players in the research and higher education infrastructure and service provisioning industries. From this gathering, five initiatives were developed to make research software more sustainable:

a. Researchers should be aware that software is crucial to their work.
b. The financing it gets and the research it allows should determine the worth of research software as a research object.
c. With their influence, funders should push for software sustainability.
d. The research community needs to be equipped with skills related to software sustainability.
e. Organizations, whether central or distributed, should be established to serve as hubs for software sustainability expertise.

Some of these issues have prompted many attempts to find solutions. By providing postdoctoral and PhD researchers with computer science foundations, Software Carpentry aims to fill the skills gap. Originally hosted by the US National Laboratories, Software Carpentry has expanded into a worldwide volunteer initiative since its 1998 inception. The program's two-day practical courses educate participants on techniques commonly used in the software industry, including version control, program design, and task automation. In a similar vein, Data Carpentry was founded in 2014 [49] to address the increasing need for researchers to improve their skills in data analysis and management. Data Carpentry follows the same two-day course structure as its parent organization but is specifically tailored to research data. These programs have been widely requested and embraced (400 workshops for more than 12,000 academics were held between January 2013 and July 2015, for instance), and they have significantly improved the chronic shortage of skills in the research community. Nevertheless, they are efforts to solve a problem that, looking back, should have been tackled earlier in the research and practitioner career path. Therefore, we need to consider training and education, especially for first-year students majoring in computer science and engineering.

Attracting software talent in academia is already a challenge, and the "hotchpotch" of solutions put up to meet the needs of different departments (such as HR and finance), university cultures, and funding limitations only makes things worse. It is challenging to find and keep such important employees since software specialists are often linked to professions that are not indicative of their work and lack a clear career path. Furthermore, funding assessment committees have a skewed preference for hiring

"traditional" academics rather than software specialists. As a result, there aren't enough of these specialists in high positions, which would help to increase role recognition and bring about institutional and cultural change. Therefore, these Research Software Engineers (RSEs) need a recognized career path in order to recover expenses and have their work evaluated using metrics relevant to their position (rather than merely publications). In addition, this would assist convince review panels that hiring software specialists is a legitimate practice and increase knowledge and recognition for the profession. Enhancing the sustainability and repeatability of the software supporting contemporary research while expanding access to essential skills would be possible with a supported recruitment and retention procedure.
The Policy team looks at software-related community concerns and launches awareness-building and problem-solving efforts. The Software team provides direct assistance in evaluating and enhancing research software and procedures. The Software team regularly hosts an "Open Call for Projects" to prioritize submissions, focusing on their value to the broader research community. This is done because of the scope of prospective work in the field of software enhancement. The evaluations are presented in the form of a report including observations and recommendations based on the subject's past performance. The evaluations, structured like software reviews, assess the program, its development, operational infrastructure, and management processes. Figure 7 shows the locations where the Software team supported Open Call projects from May 2013 to October 2015. Nineteen projects from the Institute's Open Call's first five cycles are included in this research.

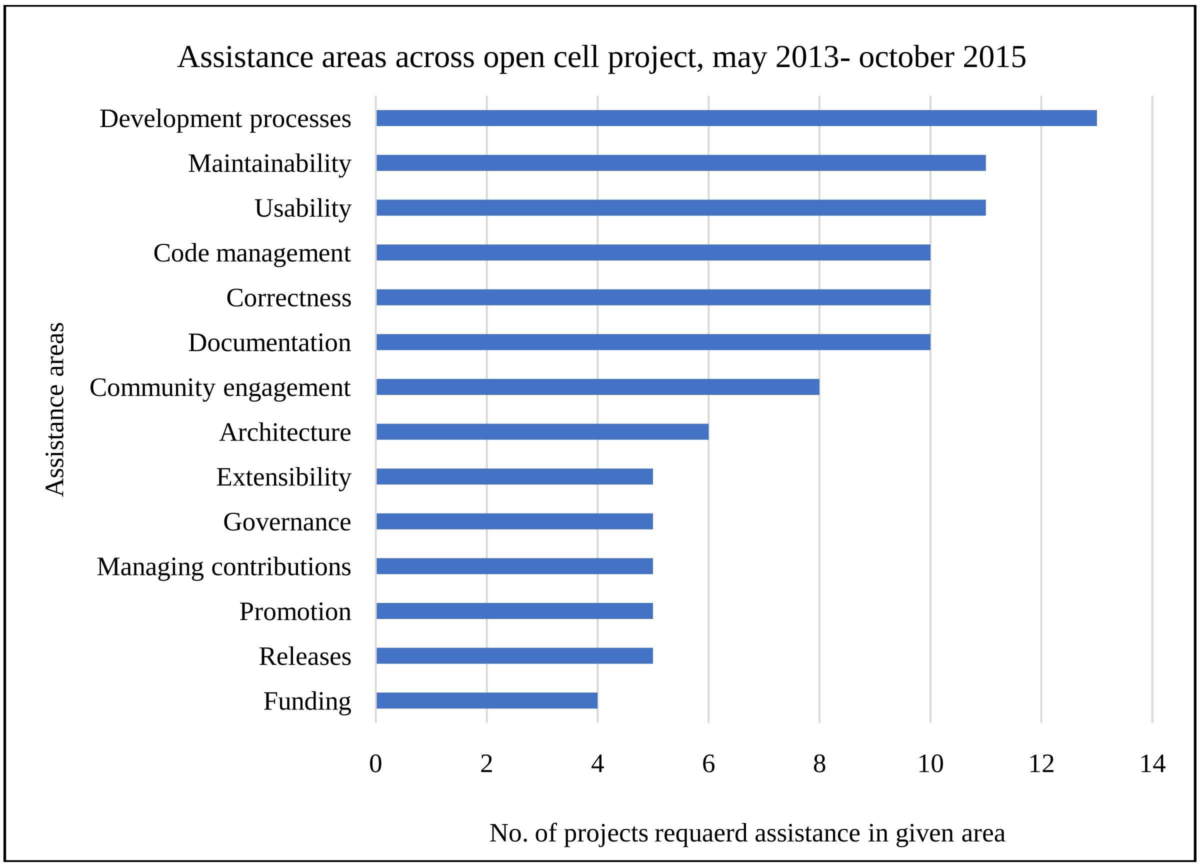

Figure 7. Areas where the Institute Software team provided support for open-call projects between May 2013 and October 2015 [50]

This study lends credence to the previously established cultural and technological problems, most notably the urgent need to fill the knowledge gap in research software creation, deployment, and maintenance. These findings are typical of the problems that the research community frequently highlights. As was previously said, there is a connection between architectural concerns and other issues including system evolution (which is connected to several of these domains), code maintainability, and modifiability (including extensibility). When focusing solely on software architecture, independent of processes and infrastructure, it ranks among the top five areas needing improvement and is a concern in over 25% of projects.

6. CONCLUSION

Sustainable software practices are vital given the growing use of software in both daily applications and cutting-edge research. The present study has investigated several aspects of software sustainability, with a particular emphasis on software design choices, sustainability measurements, and the dynamic function of software in scholarly investigations. Software architecture choices have a significant effect on sustainability. As was said, design decisions affect software systems' overall sustainability in addition to its lifespan and functionality. The idea of "sustainability debt" draws attention to the negative effects that previous choices might have on sustainability in terms of social, technological, and economic aspects. Effectively managing and capturing architectural knowledge (AK) with the use of tools and meta-models may help reduce this debt and promote more environmentally friendly decision-making procedures. Metrics are essential for controlling technical debt, recognizing it, and determining the sustainability of software. These methods provide important insights into the quality and sustainability of software systems, ranging from code metrics that assess complexity and maintainability to architectural knowledge metrics that assess decision networks. By directing developers and researchers toward more sustainable methods, the integration of diverse indicators aids in the analysis and resolution of possible sustainability challenges. A paradigm change has occurred as a result of the incorporation of software into research, making it an essential part of contemporary scientific investigation. Even with these developments, issues like poor software skill recognition and a lack of formal training in software development still exist. In order to solve these issues, programs like the Open Call for Projects and institutions like the Software Sustainability Institute are essential. They draw attention to the need of improved software processes, skill development, and professional recognition for software developers. In conclusion, attaining software sustainability requires a multifaceted strategy that includes careful consideration of architectural choices, reliable measurements, and a constant drive to enhance software processes. Given the continued importance of software in both study and daily life, taking care of these issues will guarantee that software systems continue to be dependable, maintainable, and able to assist long-term objectives. For software to continue to be a significant resource for innovation and discovery in the future and to be sustainable, researchers, developers, and institutions must work together.

REFERENCES

[1] Argawal, S., Nath, A., & Chowdhury, D. (2012). Sustainable Approaches and Good Practices in Green Software Engineering. International Journal of Research and Reviews in Computer Science (IJRRCS), Vol. 3(1).

[2] Calero, C., & Bertoa, M. F. (2013). Sustainability and Quality: icing on the cake. Journal of Green Engineering.

[3] Durdik, Z., Klatt, B., Koziolek, H., Krogmann, K., Stammel, J., & Weiss, R. (2012). Sustainability guidelines for long-living software systems. 2012 28th IEEE International Conference on Software Maintenance (ICSM), 517–526. doi:10.1109/ICSM.2012.6405316

[4] Johann, T., Dick, M., Kern, E., & Naumann, S. (2011). Sustainable Development , Sustainable Software , and Sustainable Software Engineering. International Symposium on Humanities, Science and Engineering Research.

[5] Johann, T. (2013). Position Paper: The Social Dimension of Sustainability in Requirements Engineering University of Hamburg University of Hamburg. Journal of Systems and Software.

[6] Kitchenham, B. (2004). Procedures for Performing Systematic Reviews. NICTA Technical Report (TR/SE-0401, 1–28.

[7] Kitchenham, B. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering (pp. 1–53).

[8] Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering – A systematic literature review. Information and Software Technology, 51(1), 7–15.

[9] Allen, C. Aragon, C. Becker, J. Carver, A. Chis, B. Combemale, M. Croucher, K. Crowston, D. Garijo, A. Gehani, C. Goble, R. Haines, R. Hirschfeld, J. Howison, K. Huff, C. Jay, D. S. Katz, C. Kirchner, K. Kuksenok, R. Lämmel, O. Nierstrasz, M. Turk, R. van Nieuwpoort, M. Vaughn, J. J. Vinju, "Engineering Academic Software (Dagstuhl Perspectives Workshop 16252)," Dagstuhl Manifestos, vol. 6, no. 1, pp. 1–20, 2017.

[10] Ameller, C. Farré, X. Franch, D. Valerio and A. Cassarino, "Towards continuous software release planning," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, pp. 402-406, 2017.

[11] Becker, R. Chitchyan, L, Duboc, S. Easterbrook, B. Penzenstadler, N. Seyf, and C. C. Venters. 2015, May. Sustainability design and software: the Karlskrona manifesto. In IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE) (Vol. 2, pp. 467-476), 2015.

[12] Betz, C. Becker, R. Chitchyan, L. Duboc, S.M. Easterbrook, B. Penzenstadler, N. Seyff, C. C. Venters, Sustainability Debt: A Metaphor to Support Sustainability Design Decisions, Fourth International Workshop on Requirements Engineering for Sustainable Systems, RE4SuSy, CEUR Workshop Proceedings, pp. 45-54, 2015.

[13] Becker, R. Chitchyan, L. Duboc, S. Easterbrook, M. Mahaux, B. Penzenstadler, G. RodriguezNavas, C. Salinesi, N. Seyff, C. C. Venters, C. Calero, S. Akinli Kocak, S. Betz. The Karlskrona manifesto for sustainability design. 2014. Available at: http://sustainabilitydesign.org/.

[14] Becker, R. Chitchyan, L, Duboc, S. Easterbrook, B. Penzenstadler, N. Seyf, and C. C. Venters. 2015, May. Sustainability design and software: the Karlskrona manifesto. In IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE) (Vol. 2, pp. 467-476), 2015.

[15] Brett, M. Croucher, R. Haines, S. Hettrick, J. Hetherington, M. Stillwell, C. Wyatt. Research Software Engineers: State of the Nation Report 2017.

[16] Brown, S. Sentance, T. Crick and S. Humphreys. Restart: The Resurgence of Computer Science in UK Schools. ACM Transactions on Computer Science Education, 14(2), 1–22, 2014

[17] Calero, M. A. Moraga, and M. F. Bertoa. "Towards a software product sustainability model," WSSSPE1: First workshop on sustainable software for science: practice and experiences, SC'13, 17 Denver, CO, USA, 2013.

[18] Carrillo, R. Capilla, O. Zimmermann, and U. Zdun. Guidelines and metrics for configurable and sustainable architectural knowledge modelling. In Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSA'15, pp. 1–5. ACM, 2015.

[19] Carrillo, "A Sustainable for Architectural Design Decisions Management". Doctoral Thesis in Systems Engineering and Services for the Information Society, Technical University of Madrid (UPM), Madrid, Spain, 2017.

[20] Capilla, A. Jansen, A. Tang, P. Avgeriou, M.A. Babar, 10 years of software architecture knowledge management: Practice and future. Journal of Systems and Software 116(6), 191-205, 2016.

[21] Capilla, E.Y. Nakagawa, U. Zdun, C. Carrillo, Toward Architecture Knowledge Sustainability: Extending System Longevity. IEEE Software 34(2), 108-111, 2017.

[22] V. Cerf. A Brittle and Fragile Future. Communications of the ACM, 60(7), 2017

[23] Crick, B. A. Hall and S. Ishtiaq. Reproducibility in Research: Systems, Infrastructure, Culture. Journal of Open Research Software 5(1), 2017.

[24] Chitchyan, C. Becker, S. Betz, L. Duboc, B. Penzenstadler, N. Seyff, and C. C. Venters, "Sustainability design in requirements engineering: State of practice," in Proceedings of the 38th International Conference on Software Engineering Companion, ser. ICSE-SEIS '16, pp. 533–542, 2016.

[25] Crouch, N. Chue Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, M. Parsons. "The Software Sustainability Institute: Changing Research Software Attitudes and Practices," Computing in Science & Engineering, vol.15, no.6, pp.74,80, 2014.

[26] Garcia, I. Ivkovic and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 486-496, 2013.

[27] Groher and R. Weinreich, "An interview study on sustainability concerns in software development projects," in Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA '17, 2017.

[28] Kasurinen, M. Palacin-Silva, and E. Vanhala, "What concerns game developers?: A study on game development processes, sustainability and metrics," in Proceedings of the 8th Workshop on Emerging Trends in Software Metrics, WETSoM '17, pp. 15–21, 2017.

[29] Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A.Shapochka, A Case Study in Locating the Architectural Roots of Technical Debt, ICSE'15, IEEE CS, 179-188, 2015.

[30] Koziolek, D. Domis, T. Goldschmidt, P. Vorst, R. J. Weiss, Morphosis: A lightweight method facilitating sustainable software architectures. In: 10th Joint Working Conference on Software Architecture (WICSA 2012) 6th European Conference on Software Architecture (ECSA 2012), pp. 253-257, 2012.

[31] Koziolek. Sustainability evaluation of software architectures: A systematic review. In CM SIGSOFT Symp.—ISARCS on Quality of Software Architectures—QoSA and Architecting Critical Systems—ISARCS, QoSA'115, 2015.

[32] Le, C. Carrillo, R. Capilla, N. Medvidovic. Relating Architectural Decay and Sustainability of Software Systems, 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, IEEE pp. 178-181, 2016.
[33] Le. Architectural-based speculative analysis to predict bugs in a software system. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, 2016.
[34] Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In Proc. Mining Software Repositories, 2015.
[35] Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In 10th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA'14, 2014.
[36] Mahmoud and I. Ahmad, "A green model for sustainable software engineering," Intl. Journal of Software Engineering and Its Applications, vol. 7(4), pp. 55–74, 2013.
[37] Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE '16, pp. 237–248, 2016.
[38] Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells, 12th Working IEEE/IFIP Conference on Software Architecture, WICSA IEEE CS, pp. 51-60, 2015.
[39] Nakagawa, M. Guessi, D. Feitosa, F. Oquendo, J. C. Maldonado, Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures, 11th Working IEEE/IFIP Conference on Software Architecture, WICSA, pp. 143-153, 2014.
[40] Philippe, N. Chue Hong and S. Hettrick. Preliminary analysis of a survey of UK Research Software Engineers. 4th Workshop on Sustainable Software for Science: Practice and Experience, 2016.
[41] Ramsey, On not defining sustainability, Journal of Agricultural and Environmental Ethics, vol. 28(6), 1075–1087, 2015.
[42] Roher and D. Richardson, "A proposed recommender system for eliciting software sustainability requirements," in Workshop USER, pp. 16–19, 2013.
[43] Roher and D. Richardson, "Sustainability requirement patterns," IEEE Third International Workshop in Requirements Patterns (RePa), 2013 IEEE CS, pp. 8–11, 2013.
[44] Sherman, I. Hadar, Identifying the need for a sustainable architecture maintenance process. In: 5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2012), pp. 132-134, 2012.
[45] Taivalsaari and T. Mikkonen, "A Roadmap to the Programmable World: Software Challenges in the IoT Era," in IEEE Software, vol. 34, no. 1, pp. 72-80, Jan.-Feb. 2017.
[46] Zimmermann, Metrics for Architectural Synthesis and Evaluation - Requirements and Compilation by Viewpoint. An Industrial Experience Report. 2nd IEEE/ACM International Workshop on Software Architecture and Metrics, SAM 2015, IEEE DL, 8-14, 2015.
[47] Zimmermann, L. Wegmann, H. Koziolek, T. Goldschmidt, Architectural Decision Guidance Across Projects - Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge. WICSA, pp. 85-94, 2015.
[48] Voas, and R. Kuhn, What happened to software metrics? Computer, 50(5), 88-98, 2017.
[49] Sehestedt, C. Cheng, E. Bouwers, Towards quantitative metrics for architecture models. In: 11th IEEE/IFIP Conference on Software Architecture (WICSA 2014), pp. 51-54, 2014.
[50] Bosch, Continuous Software Engineering. Springer, 2014.