Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review

DALIA SOBHY, Computer Engineering Department, Arab Academy of Science and Technology and Maritime Transport, Egypt
RAMI BAHSOON, School of Computer Science, University of Birmingham and FRSA, UK
LEANDRO MINKU, School of Computer Science, University of Birmingham, UK
RICK KAZMAN, Information Technology Management, University of Hawaii and SEI/CMU, USA

Context: Evaluating software architectures in uncertain environments raises new challenges, which require continuous approaches. We define continuous evaluation as multiple evaluations of the software architecture that begins at the early stages of the development and is periodically and repeatedly performed throughout the lifetime of the software system. Numerous approaches have been developed for continuous evaluation; to handle dynamics and uncertainties at run-time, over the past years, these approaches are still very few, limited, and lack maturity.
Objective: This review surveys efforts on architecture evaluation and provides a unified terminology and perspective on the subject.
Method: We conducted a systematic literature review to identify and analyse architecture evaluation approaches for uncertainty including continuous and non-continuous, covering work published between 1990-2020. We examined each approach and provided a classification framework for this field. We present an analysis of the results and provide insights regarding open challenges.
Major results and conclusions: The survey reveals that most of the existing architecture evaluation approaches typically lack an explicit linkage between design-time and run-time. Additionally, there is a general lack of systematic approaches on how continuous architecture evaluation can be realised or conducted. To remedy this lack, we present a set of necessary requirements for continuous evaluation and describe some examples.

Additional Key Words and Phrases: Continuous Software Architecture Evaluation, Design-time Software Architecture Evaluation, Run-time Software Architecture Evaluation, Uncertainty.

Reference Format:
Dalia Sobhy, Rami Bahsoon, Leandro Minku, and Rick Kazman. 2021. Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review. 1, 1 (April 2021), 50 pages.

1 INTRODUCTION

Architecture evaluation is a milestone in the decision-making process. It aims at justifying the extent to which architecture design decisions meet a system's quality requirements and their trade-offs, particularly in the face of operational uncertainties and changing requirements. The evaluation can aid in early identification and mitigation of design risks; the exercise is typically done in an effort to save integration, testing and evolution costs [124]. Examples of seminal work include Architecture Tradeoff Analysis Method (ATAM) [85], and Cost Benefit Analysis Method (CBAM) [82].

Authors' addresses: Dalia Sobhy, Computer Engineering Department, Arab Academy of Science and Technology and Maritime Transport, Alexandria, Egypt, dalia.sobhi@aast.edu; Rami Bahsoon, School of Computer Science, University of Birmingham and FRSA, Birmingham, UK, r.bahsoon@cs.bham.ac.uk; Leandro Minku, School of Computer Science, University of Birmingham, Birmingham, UK, l.l.minku@cs.bham.ac.uk; Rick Kazman, Information Technology Management, University of Hawaii and SEI/CMU, Hawaii, USA, kazman@hawaii.edu.

XXXX-XXXX $

Software architectures that operate in dynamic and non-stationary environments (e.g., IoT and cloud applications) require a fundamental shift in the way evaluations are conducted. This is due to unforeseen factors that may affect the evaluation, including (but not limited to), fluctuations in QoS, multi-tenancy, hyper-connectivity, sensor ageing effects, *etc* [71, 109, 130].

Though existing design-time evaluation approaches promise to evaluate flexibility in architectures under uncertainty and their responses in enabling change [19, 82, 85], in contexts of highly dynamic environments these approaches tend to be limited because there may be emerging scenarios where the architect cannot rely solely on design-time evaluation. Such scenarios require a run-time evaluation to inform and calibrate the design-time decisions. In this context, a more continuous approach would benefit the evaluation process. We define *continuous* software architecture evaluation as *multiple evaluations of the software architecture that begins at the early stages of the development and is periodically and repeatedly performed throughout the lifetime of the software system.* Continuous evaluation is performed either continuously or sporadically covering either one feature (e.g. QoS) or multiple features.

There have been many research studies aimed at evaluating software architectures to deal with uncertainty which may implicitly or explicitly adopt continuous approaches (e.g. DevOps [17]). The field has attracted a wide range of researchers and practitioners. However, continuous evaluation has not been viewed as a key area within software architecture research. We still lack a clear vision regarding the elements of a continuous software architecture evaluation approach.

In past years, many research studies have reviewed design-time architecture evaluation methods (e.g. [27, 53, 122]), while some have attempted to review run-time methods without addressing them from the context of continuous architecture evaluation (e.g. [26, 47, 93, 98, 131]). In particular, to date there is no systematic literature review for software architecture evaluation approaches for uncertainty which may implicitly or explicitly adopt continuous approaches. A systematic literature review (SLR) is a methodological mean to aggregate empirical studies, to systematically investigate a research topic, answer specific research questions, and finally determine the gaps and research directions for the research topic [88, 89, 116].

The objective of this study is to (i) provide a basic classification schema which categorises software architecture evaluation approaches under uncertainty; (ii) categorise the current design-time and run-time approaches for evaluating software architectures based on this schema; (iii) determine the necessary guidelines for developing a continuous evaluation approach; (iv) point out current gaps and directions for future research in software architectures for environments characterised by uncertainty, where we consider both design-time and run-time evaluation that take into account the possibility of uncertainties in the environment where the system will operate / is operating. Concretely, we aim to provide answers for the following research questions:

(1) How can the current research on software architecture evaluation under uncertainty be categorised and what are the current state-of-the-art approaches with respect to this categorisation? *The goal is to provide a categorisation of existing architecture evaluation approaches under uncertainty and classify the state-of-the-art approaches under this categorisation.*
(2) What are the actions taken by these architecture evaluation approaches to deal with uncertainty? *The aim of this question is to demonstrate and discuss how the existing approaches deal with uncertainty and whether these actions can contribute to developing more continuous approaches.*
(3) What are the current trends and future directions in software architecture evaluation for uncertainty and their consideration for continuous evaluation? *This question aims to show how researchers and practitioners can benefit from the existing approaches to draw inspiration*

on the essential requirements and address the pitfalls when developing a continuous evaluation approach.

The manuscript is structured as follows: Section 1.1 identifies and explains the necessary concepts to ease the understanding of the review. Section 2 demonstrates the systematic literature review process, Section 3 provides an overview of the included studies from the chronological and distribution perspectives. Section 4 categorises the included studies with respect to a classification framework and presents the limitations of review. The related reviews are discussed in Section 5. New trends and research directions are discussed in Section 6. Finally, Section 7 concludes the work.

1.1 Preliminaries and Basic Concepts

In this section, we list descriptions of the main concepts used in this review to ease the analysis.

1.1.1 Architecture Design Decisions. The foundation of an architecture is in the set of taken [25, 80, 137]. The architects define the possible set of candidate architectures to serve a particular concern and then based on their experience and knowledge they choose the best candidate [35]. For example, in an IoT application, the architect could prefer processing the data in the cloud rather than the fog devices to improve the energy consumption. However, this design decision could have a negative impact on the performance. This motivates the need for software architecture evaluation.

1.1.2 Software Architecture. In the literature, software architecture is defined in many ways. In our work, we use the definition introduced by ISO/IEC/IEEE 42010:2011: "the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution". This definition is complementary to [115, 125] and later ones [16]. In this context, a software architecture represents the abstractions for a software system by defining its structure, behaviour, and key properties [125]. These include software components (i.e. processing and computational elements), connectors (i.e. interaction elements), and their relation to the environmental conditions [16, 115].

1.1.3 Architecture Evaluation. It is a milestone in the decision-making process. Classical approaches to architecture evaluation are generally a human-centric, where architects and various stakeholders (e.g. developers, managers, *etc*) are involved to evaluate the extent to which the architecture design decisions and adopted styles can meet quality attributes of interest and their trade-offs. The exercise also involves analysis of costs and likely added value of the decisions. Classical approaches heavily rely on experts' judgement; they utilise human generated inputs, such as scenarios for evaluating the architecture. Evaluation is conducted at design-time and before the system is built, covering the statics of an architecture (e.g. style, structure and topology) and its dynamics (e.g. likely performance and scalability).

1.1.4 Design-time Architecture Evaluation. It is the process where humans, tools, and methods are used to reason about the architecture of the system-to-be. The evaluation can cover both static aspects of the architecture relating to structure, topology, environment, and style, *etc* and dynamic analysis that relates to behavioural properties of the architecture, such as performance, scalability, *etc*. The evaluation can heavily rely on stakeholders involvement and their estimates. Estimation can be backed up by experts judgement about the domain, historical data and benchmarks that relates to the likely performance of similar systems, or what-if analysis of simulated instances for the projected deployment environments, predicted or eventual load (before the system is deployed).

1.1.5 Run-time Architecture Evaluation. It means the execution of the architecture under study; this can be a typical execution profile or it can be the actual deployed system implementing the architecture for the objectives of profiling, refinements or enrichment. For either cases, architects can collect dynamic, near real or real time information about the performance of QA of interest to inform the evaluation or further tuning of the running system. In other cases, simulated data (e.g. QoS data) are used to capture the dynamic behaviour of architecture decisions under uncertainty at run-time and to use such information to profile and evaluate design decisions, if full deployment was expensive. The evaluation can leverage simulation tools with inputs from the running system to perform anticipatory evaluation of key design decisions and their possible variants based on the run-time contextual requirements.

1.1.6 Continuous Architecture Evaluation. It goes beyond the classical architecture evaluation approaches to include additional run-time information that can assist the evaluation and help in tuning the parameters. Several flavors can implement this category of evaluation: for example, info-symbiotic simulation[1] can be linked to the architecture to simulate how an architecture can behave if implemented and deployed in particular environment. The run-time information can be then fed into the evaluation to tune the parameters. This step can involve a self-adaptive mechanism and can leverage components of the MAPE-K to tune the parameters. As for the actors involved in the evaluation - these can be various stakeholders (architects, developers, *etc*) and automated agents (taking the form of monitoring agents for the environment, analysis, planning and actuating for the observed inputs - these can be automatic and/or interactive *etc*).

We see continuous architecture evaluation to include two activities: *design-time* and *run-time* evaluation. In particular, design-time evaluation can be used to support the necessary initial system design and deployment based on estimations only. After that, run-time evaluation can assist continuous architecture evaluation in monitoring QAs and suggesting re-configuration from a repository of candidate options, some of which their technical viability has been established but requires further profiling and confirmation following continuous monitoring at run-time. The recommendation can utilise learning and suggest a suitable configuration; it can also call for further refinements and/or phasing out of existing reconfiguration. Once the architecture is adopted, it is very expensive to change the architecture or amend its structural design. Would the architecture appear to lag behind optimality, for this case, run-time evaluation may recommend more structural changes to the architecture, which can be very expensive to deal with following deployment, unless the context is aimed as prototyping and learning through prototypes. In other words, the evaluation can be also used to *repeatedly* assess to what extent the architecture options created at design-time, as well as other potential architecture options, perform well at run-time. This enables architects to make informed decisions on potential changes to the architecture, so that its performance remains good over time. In other contexts, evaluations can be intertwined and interleaved between design-time and run-time. Consider, for example, in modern incremental software development (e.g. DevOps), microservices, *etc*, the design of each change to the system when evolving it again is "design-time".

1.1.7 Uncertainty in Architecture Evaluation. A common issue in architecture evaluation is the presence of uncertainty. In architecture evaluation and decision-making, uncertainty is the lack of full knowledge about the outcomes of deploying the architecture options [95]. For instance, the architects may be uncertain about the effect of a proposed software architecture on benefit (e.g. performance, availability, *etc*) and cost. Uncertainty also may arise due to unpredictable situations

[1] a term that is widely used by the dynamic data driven simulation system community (e.g. http://1dddas.org/InfoSymbiotics/DDDAS2020, https://sites.google.com/view/dddas-conf/home)

in dynamic applications, such as IoT. For instance, sensors ageing effects, the varying internet connectivity and mobility of sensors, fluctuations in QoS and so forth [1, 71, 108, 109].

Architecture can experience two sources of uncertainty: *aleatory* and *epistemic* [15, 50, 66]. *Aleatory* conception of uncertainty intends that uncertainty arises from variability in possible realisation of a stochastic event, where unknown and different results could appear every time one runs an experiment under similar conditions. It is also defined as "the inherent variation associated with the physical system or environment under consideration" [111]. This type of uncertainty is more common in run-time. In other words, it is the uncertainties occurring in the later execution environment. For instance, in IoT systems, new types of sensors with new communication behaviour might be introduced, which do not match the workload model assumed for a system. This knowledge will only become available after running the system. *Epistemic* conception of uncertainty denotes the rise of uncertainty due to lack of confidence or missing knowledge to a fact which is either true or false. It is also defined as "uncertainty of the outcome due to the lack of knowledge or information in any phase or activity of the modelling process" [111]. This type of uncertainty is more common in design-time. In particular, this may occur due to the impact of decisions at design-time that are not yet known (e.g. designing new way of communication, without knowing yet how much performance can be improved in a distributed and parallel setup by this decision, which one needs to implement and measure to find out). In some contexts, this type of uncertainty could be partially reduced at design-time.

1.1.8 Quality Attribute. We adopt the definition introduced by the IEEE Standard for Software Quality Metrics [45], where a quality attribute is "a characteristic of software, or a generic term applying to quality factors, quality sub-factors, or metric values". Examples of quality attributes are performance, reliability, energy consumption, availability, security, and so forth.

1.1.9 Stakeholder. We adopt the notion used by ISO/IEC/IEEE 42010:2011: "an individual, team, organization, or classes thereof, having an interest in a system". In this context, stakeholders have a stake in the success of the architecture, and of any systems that are derived from the architecture. So this could include customers, programmers, testers, reusers, architects, integrators, users, managers, *etc.* An architect is just one stakeholder among many, whose needs are less important (and hence lower priority) than the needs of many of the other stakeholders.

2 SYSTEMATIC LITERATURE REVIEW PROCESS

In this section, we will discuss the SLR protocol, how the systematic review process has been carried out, and finally the existing architecture evaluation approaches with respect to criteria and review objectives.

2.1 SLR Protocol

We have followed the systematic literature review guidelines and procedures [116] and the work of [27] to develop our review protocol. In particular, the protocol identifies the objectives of the review, the necessary background, research questions, inclusion and exclusion criteria, search strategy, data extraction and analysis of gathered data. One author has developed the review protocol and then the outcome has been revised by other authors to limit bias. The review objectives, background, and the research questions are discussed in Section 1, whereas other procedures are described below.

2.2 Inclusion and Exclusion Criteria

Initially, we needed to set up a criteria to aid in the search process and filtration of irrelevant studies. We considered English papers published in peer-reviewed journals, conferences, and workshops from 1990 and early 2019. This time frame was chosen because one of the earlier well-known architecture evaluation approaches (e.g. SAAM [83]) was published in 1994. We excluded studies that do not have software architecture evaluation as one of its main contributions. We also excluded editorials, opinion, keynote, abstract, tutorial summary, position paper, panel discussion, or technical reports, panels and poster sessions. Moreover, we found that some studies are duplicated in different versions that appear as books, journal papers, conference and workshop papers. In this context, we included only the latest and most complete version. We provide a summary of the inclusion and exclusion criteria below. Publications are included if they cover all the inclusion criteria in Section 2.2.1, and publications are excluded if they fit any of the exclusion criteria in Section 2.2.2.

2.2.1 Inclusion Criteria.

- Studies published between 1990 and early 2020.
- Studies in the domain of software architecture evaluation. In particular, the study should include a software architecture evaluation method as one of its contributions.
- Studies that discuss architecture evaluation approaches with explicit focus on high-level architecture design (e.g. component level, style, architecture design decisions and tactics), covering design-, run-time and continuous evaluation; we exclude approaches which discuss low-level structural design (e.g. code and class refactoring).
- Studies that report on software architecture evaluation supported by quantitative analysis/models (e.g, using utility theory as part of ATAM; using cost-benefit analysis as part of CBAM, *etc.*)

2.2.2 Exclusion Criteria.

- Studies that do not explicitly consider architecture evaluation. For example, some self-adaptive system studies may make use of architecture evaluation to inform self-adaptation, but may not explicitly refer to this as architecture evaluation. Such studies were excluded.
- Studies that are non-peer reviewed.
- Studies not written in English and not accessible in full-text.

2.3 Search Strategy

The search strategy was performed to identify the studies through the following:

1. Applying an initial search to determine the current systematic reviews and mapping studies, and hence identifying significantly related primary studies.
2. Using the concept of "quasi-gold" standard, as introduced by Zhang and Babar [142], where we performed a manual scan for the most well-known venues of the software architecture and software engineering domains to cross-check the automated search results.
3. Performing several trials using different combinations of keywords derived from the main objectives of the review (i.e. automated search from recognised bibliographical data sources).
4. Performing an additional search to manually check and analyse the related references (snowballing) [140] to ensure that we did not miss any important study and hence guarantee a representative set of studies.

All the prior procedures aided us in defining valid search strings along with other procedures discussed in Section 2.3.1. For the venues, we manually searched the following:

- International Conference on Software Engineering (ICSE).
- International Conference on Software Architecture (ICSA) [2].
- European Conference on Software Architecture (ECSA).

[2]Formerly the Working IEEE/IFIP Conference on Software Architecture (WICSA) and International Conference Series on the Quality of Software Architectures (QoSA).

Our manual search included the title, keywords, and abstract of each publication. After finishing the manual and automatic searches, we checked the differences between the results to guarantee the most appropriate coverage of the domain. We found that all the manual results were a subset of the automatic results (i.e. meeting the "quasi-gold" standard).

2.3.1 Keyword Selection. As mentioned above, we used both automatic and manual search. In the automatic search, we tried several keywords on search engines of electronic bibliographical sources. Manual search is not a practical procedure as it retrieves thousands of results, which is difficult to manually filter. However, we still performed a manual search (to meet the "quasi-gold" standard [142]) to ensure that we used the most suitable search queries.

One of the main challenges identified through our automatic search is a lack of well-defined terminology for the process of continuous architecture evaluation. As an example, some self-* systems can implicitly incorporate some principles that resemble architecture evaluation. To avoid missing any relevant studies, we used some generic keywords in the search query of automatic search (e.g. "run-time", "dynamic", *etc*). This led to retrieving some studies that were actually relevant to our search. We have also performed a manual search for the studies, which could seem to be a run-time architecture evaluation approach. To obtain our search query, we applied the following procedures:

1. Extract the major keywords from the objectives of review and main research topics.
2. Determine and try different spellings, related terms and synonyms for major keywords, if applicable.
3. Use the "advanced" search option to insert the complete search query and filter by date, if the bibliographical source allows for that (Section 2.3.2).
4. Pilot various combinations of search keywords in test queries.
5. Validate the results of (4) with "quasi-gold" standard.

From our pilot testing, we found that the notion of "continuous" architecture evaluation is used in different forms in the context of software architecture and software engineering with other closely-related alternative terms, such as run-time and dynamic. This is because the term "continuous" is not clearly defined. We also incorporated additional keywords which may implicitly refer to continuous evaluation, such as design-time and static (i.e. the state-of-the-art approaches for architecture evaluation). Furthermore, in other contexts, architecture evaluation is interpreted as architecture assessment or architecture analysis. Therefore, we tried to consider these related keywords in our search query and used them in an interchangeable manner.

The search query is composed of five major terms, Continuous **AND** Software **AND** Architecture **AND** Evaluation **AND** Uncertainty. To generate the main search query, we used the alternate keywords listed above. This is performed by connecting these terms through logical OR as follows: (design-time **OR** run-time **OR** design time **OR** runtime **OR** static **OR** dynamic **OR** continuous) **AND** Software **AND** (architecture **OR** architectural) **AND** (evaluation **OR** analysis **OR** assessment) **AND** uncertainty

2.3.2 Bibliographical Sources. The selected databases present the most important and highest impact journals and conference proceedings. They also provided us with the ability to perform expert search with a variety of Boolean operations and limit the search on the Title, Abstract and Keywords fields and time frame, which returned more relevant results as compared to searching all the fields. For instance, this allowed us to use Boolean "OR" to try different spellings and synonyms, and use Boolean "AND" to link the major keywords (e.g. software AND architecture AND evaluation).

The electronic bibliographical sources used include:

Table 1. Summary of Search Results and Included Studies from each database. *Note that the number of included studies listed for each of the databases excludes studies that have already been included by a former database. A total of 48 unique studies have been included.*

Database	Search Results	# Included Studies
IEEE Xplorer	994	11
ACM digital library	2108	8
SpringerLink	999	3
ScienceDirect	524	5
GoogleScholar	1000	7
Other		
Snowballing Process	349	14
Total		48

– IEEE Xplorer (http://ieeexplore.ieee.org/Xplore/)
– ACM digital library (http://portal.acm.org/)
– SpringerLink (http://www.springerlink.com/)
– ScienceDirect (http://www.sciencedirect.com/)
– GoogleScholar (http://scholar.google.com/)

Note that we included Google Scholar as there are some of works in software architecture evaluation (e.g. ATAM), which were not retrieved in the first four databases. we have found that Google retrieves many irrelevant results after the first pages of retrieved results. This is because Google enables retrieval of results that do not match the search query completely. Therefore, we have limited the Google scholar results to 1000. Other works (e.g. [2]) have also limited the Google scholar results to specific number of pages.

2.4 Search Execution

In this stage, we executed the search process in Figure 1, realising the procedures in Section 2.3. Initially, we manually searched in the current systematic reviews and mapping studies (e.g. [11, 27, 53, 98, 122]) to identify significantly related primary studies (13 results). We then performed manual search (17 results) to determine the set of studies to be compared with automatic search list (i.e. "quasi-gold" standard). After that, we searched through all the search engines and bibliographical sources mentioned in Section 2.3.2 using search queries created in Section 2.3.1. All the search engines provided the option to save the results to excel spreadsheets, except for Springer which exports only the first 999 relevant results and ScienceDirect which does not have that option and hence a manual scan was performed. We then filtered the 5,625 primary studies using title, abstract, full-text (when needed), inclusion and exclusion criteria. We also snowballed the primary studies [140], where we scanned the list of references for the primary studies and the citations to add related works (349 results), which were not identified by the bibliographical engines. In the end we included 48 studies. The search results and number of included studies from each database and snowballing process are listed in Table 1.

2.5 Quality Assessment

To assess the quality of the findings, we adopted similar quality criteria to the ones used by [27]. The following criteria show the credibility of an individual study when analyzing the results:

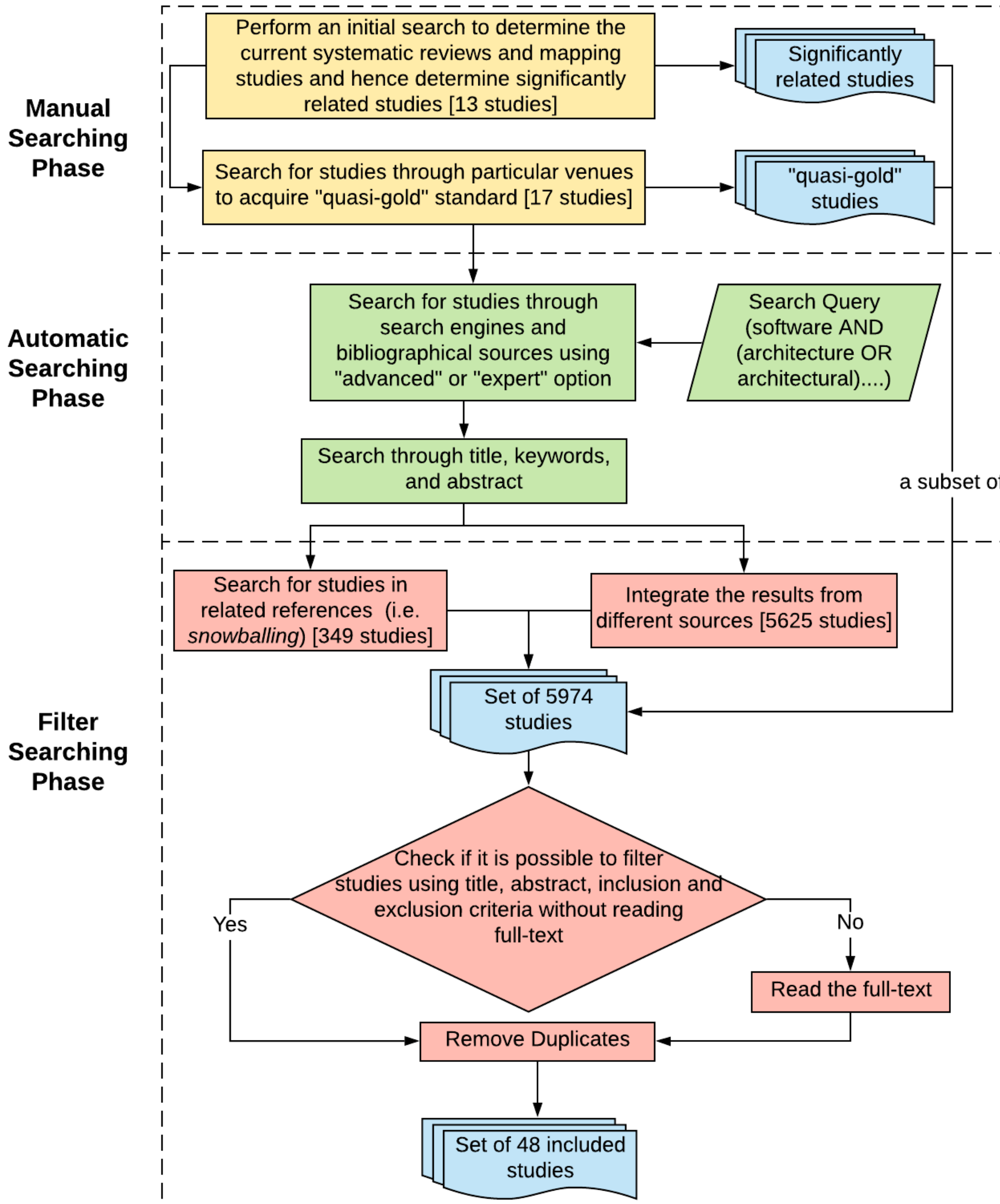

Fig. 1. Search Execution.

Table 2. Data Extraction Criteria.

Extracted Data	Description
Study Identification	Unique ID for the study
Bibliographical references	Author, title, publication type, source and year
Study Type	Book, journal paper, conference paper workshop paper
Study Focus	Main area and study objectives
Strengths and Limitations	Identified strengths and limitations of the approach and its application and its potentials for future directions

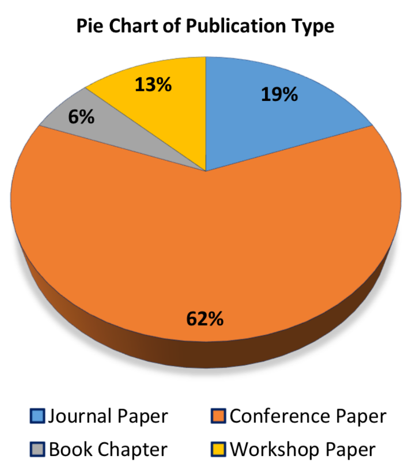

Fig. 2. Distribution of the publication types.

1. The study provides evidence or theoretical reasoning for their experimental evaluation and data analysis rather than relying on non-justified or adhoc statements.
2. The study describes the context in which the research was conducted.
3. The design and implementation of the research is mapped to the study objectives.
4. The study provides full description of their data collection process.

All 48 studies identified in the search described above met the quality assessment criteria.

2.6 Data Extraction process

In this process, we performed a thorough scan for the 48 included papers to extract the relevant data, which were managed by Excel spreadsheets and bibliographical management tool BibTeX. The data extraction for the 48 studies was driven by the form depicted in Table 2 and the classification framework in Section 4.1. For the data analysis, we investigated the extracted data with respect to their relationships. The results of this process is given in the subsequent sections. The list of included studies are presented in Appendix B.

3 OVERVIEW OF THE INCLUDED STUDIES

Here we provide an overview of the included studies with respect to their distribution along publication channels, over the years, and their ranks.

Table 3. Distribution of included studies along with the publication channels.

Publication Channel	**No. of Studies**
IEEE International Conference on Software Engineering (ICSE)	8
International Conference on Software Architecture (ICSA) [3]	7
Software Engineering for Self-Adaptive Systems (SEAMS)	4
Journal of Systems and Software (JSS)	5
Book	3
IEEE Transactions on Software Engineering (TSE)	2
IEEE Internet Computing	1
Software Quality Journal	1
Empirical Software Engineering	1
European Conference of Software Architecture (ECSA)	2
IEEE International Conference on Software Maintenance (ICSM)	1
ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	1
IEEE International Conference on Autonomic Computing (ICAC)	1
ACM/SPEC International Conference on Performance Engineering (ICPE)	1
International Conference on Software Reuse (ICSR)	1
International Conference on Quality of Software Architectures (QoSA)	2
IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)	1
International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)	1
IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)	1
International Workshop on the Economics of Software and Computation (ESC)	1
IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOC)	1
Proceedings of the 3rd international workshop on Software and performance (WOSP)	1
Software and Systems Modeling (Springer)	1
International Workshop on Software Engineering for Embedded Systems (SEES)	1
Total	48

3.1 Distribution of Studies over Publication Channels

Most of the included studies (i.e. 48 studies) were published in the most well-known and prominent journals and conferences. In Table 3, we provide an overview of the included studies with respect to their publication channels and the number of studies per channel. We have checked the included

[3]Formerly the Working IEEE/IFIP Conference on Software Architecture (WICSA) and International Conference Series on the Quality of Software Architectures (QoSA).

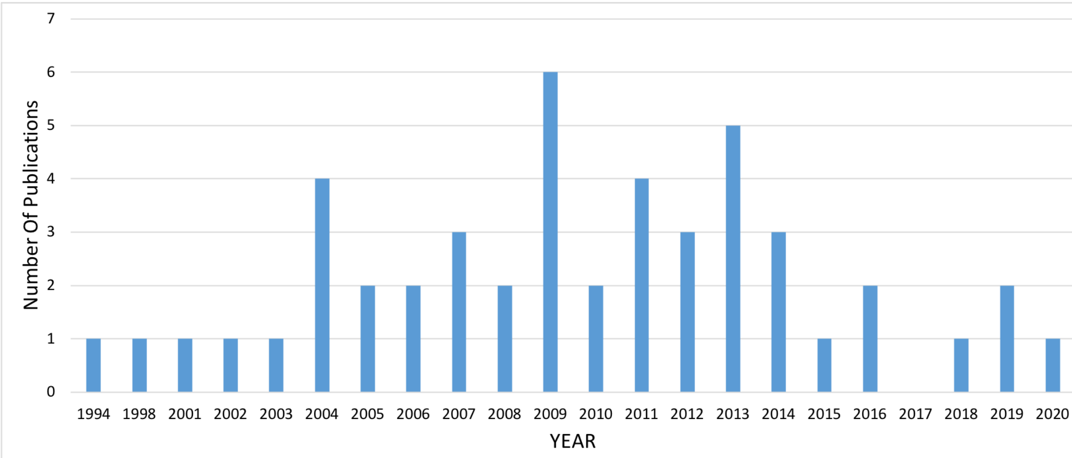

Fig. 3. Distribution of the publication types among the years.

Table 4. An overview of citation rate of included studies.

Cited by	<10	10-50	50-100	>100
Number of Studies (Total = 48)	8	21	5	14

studies against the criteria for quality assessment and confirmed that they indeed fulfil the quality criteria introduced in Section 2.5. We have also plotted the distribution of the included studies related to the publication channel (i.e. conference, journal, *etc*) in Figure 2. From these results, we found that there are a significant number of studies published in conferences (about 62%), followed by a smaller number of studies (19%) in journals. There are limited studies published in workshops (roughly 13%) and books (about 6%). This indicates that architecture evaluation approaches are still presented in conferences, and some of them have matured and published through books and journals.

3.2 Distribution of Included Studies Through the Years

By analysing the studies by year of publication, as depicted in Figure 3, we observe an increasing trend in the domain of software architecture evaluation starting from 2003 till 2013 (with some oscillation). Though it may seem that interest in architecture evaluation has decreased in the past four years, there were recent studies that provided new architecture evaluation approaches, which are included in this survey (e.g., [127, 136]).

3.3 Citation Rate of Included Studies

We list in Table 4 the citation rate for the included studies, which was obtained from Google Scholar[4]. The citation rate is not meant for comparing studies; instead we use it to provide a rough estimate of the quality of papers. In particular, almost five studies were cited by fewer than 10 sources. Two of them were cited in 2004 and 2010 and hence we do not expect that they will be cited further, whereas the others are relatively new. Almost 45% of the studies (21 publications)

[4] http://www.googlescholar.com

Table 5. Featuring the most cited studies above 100 citations.

Rank	Ref	Author(s)	Year	Title
1	[43]	R. Kazman, M. Klein, P. Clements and others	2003	Evaluating software architectures
2	[83]	R. Kazman, L. Bass, G. Abowd, & M. Webb	1994	SAAM: A method for analyzing the properties of software architectures
3	[19]	P. Bengtsson, N. Lassing, J. Bosch, and H. Vliet	2004	Architecture-level modifiability analysis (ALMA)
4	[30]	R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, & G. Tamburrelli	2011	Dynamic QoS management and optimization in service-based systems
5	[58]	I. Epifani, C. Ghezzi, R. Mirandola, & G. Tamburrelli	2009	Model evolution by run-time parameter adaptation
6	[82]	R. Kazman, J. Asundi, & P. Clements	2001	Quantifying the costs and benefits of architectural decisions
7	[139]	G. Williams,U. Smith	2002	PASA: A Method for the Performance Assessment of Software Architectures
8	[18]	P. Bengtsson,J. Bosch	1998	Scenario-based software architecture reengineering
9	[133]	G. Tesauro	2007	Reinforcement learning in autonomic computing: A manifesto and case studies
10	[40]	S. Cheng	2004	Rainbow: cost-effective software architecture-based self-adaptation
11	[5]	T. Al-Naeem, I. Gorton, and M. Babar	2005	A quality-driven systematic approach for architecting distributed software applications
12	[62]	N. Esfahani, E. Kouroshfar, & S. Malek	2011	Taming uncertainty in self-adaptive software
13	[145]	L. Zhu, A. Aurum, I. Gorton, & R. Jeffery	2005	Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process
14	[32]	R. Calinescu & M. Kwiatkowska	2009	Using quantitative analysis to implement autonomic IT systems

were cited by 10-50 other sources, and five studies were cited 50-100 times. Fourteen studies have very high rates with more than 100 citations and the first ranked study was cited almost 1578 times. This shows that the included studies are, in general, highly cited, which signifies their quality and impact. In Table 5, we present the most cited publications. The first study is a book, and the remainder are journal and conference papers.

4 DATA EXTRACTION RESULTS

This section aims to provide answers for the first and second research question: *(1) How can the current research on software architecture evaluation under uncertainty be categorised and what are the current state-of-the-art approaches with respect to this categorisation?*; *(2) What are the actions taken by these architecture evaluation approaches to deal with uncertainty?* Our analysis of research topics addressed in each study and the systematic reviews and surveys found in literature (e.g. [11, 27, 53, 93, 98]) helped us in developing the following classification framework. This classification aided us in filtering, mapping, and understanding the architecture evaluation domain. We also discuss how the included evaluation approaches deal with uncertainty.

4.1 Classification Framework

Next, we will explain in detail the criteria presented in Figure 4.

1. **Quality Evaluation:** Architecture evaluation is typically done as a milestone review that aims at justifying the extent to which the architecture design decisions meet the quality requirements and their trade-offs. The evaluation can aid in early identification and mitigation of design risks. The point of the exercise is to avoid poor decisions, identify a stable architecture and thus save integration, testing and evolution costs that can be attributed to design decisions that are not fit in meeting the changes [124]. We review *Stage of Evaluation*, covering *design-time*, *run-time* and *continuous* along with *Approaches to Evaluation* covering major efforts including *utility-based, scenario-based, parametric-based, search-based, economics-based,* and *learning-based.*
2. **Quality Attributes Considerations:** Our literature review aims to show how the studied software architecture evaluation methods *addressing quality attributes* (i.e. focus on single versus multiple QAs), as well as what are the *supported quality attributes.* Examples of quality attributes are performance, reliability, security, cost, *etc.* Further *monitoring and treatment of quality attributes* is an important aspect to discuss, which could provide the architects and architecture evaluaters with the necessary elements to design a continuous architecture evaluation framework.
3. **Level of Autonomy:** In software architecture evaluation, the level of autonomy is an important aspect while designing a continuous architecture evaluation framework. In this context, we will review how the studies performed the *management of stakeholder input* and *management of trade-offs* between conflicting requirements.
4. **Uncertainty Management:** In this category, we focus on discussing the *sources of uncertainty* and how the literature has *treated uncertainty*.

In Section 4.2 to 4.5, we aim to provide answers for the review's research questions mentioned earlier. We classify the architecture evaluation approaches as design-time and run-time. In each category we further classify and explain the existing architecture evaluation approaches with respect to the framework (answering research question 1). We also discuss the actions taken by these architecture evaluation approaches to deal with uncertainty (answering research question 2). Table 6- 12 provide a summary of the representative contributions with respect to the classification framework.

4.2 Quality Evaluation

4.2.1 Approaches to Evaluation Under Uncertainty. Architecture evaluation methods can take several forms: the methods can be bespoke, providing phases and systematic guidance for architects to evaluate the extent to which the architecture can meet its non-functional goals and trade-offs - e.g. ATAM [85], CBAM [82], *etc.* Additionally, the architects can utilise generic frameworks for quality assessment, which can be used to evaluate any artefact under consideration, where the software architecture can be a beneficiary. Regardless of the type of evaluation used, the architects can adopt one of the below commonly approaches to evaluate architecture design decisions and choices in the presence of uncertainty. The commonly used approaches can be categorised as *utility-based*, *scenario-based*, *parametric-based*, *search-based*, *economics-based*, and *learning-based.*

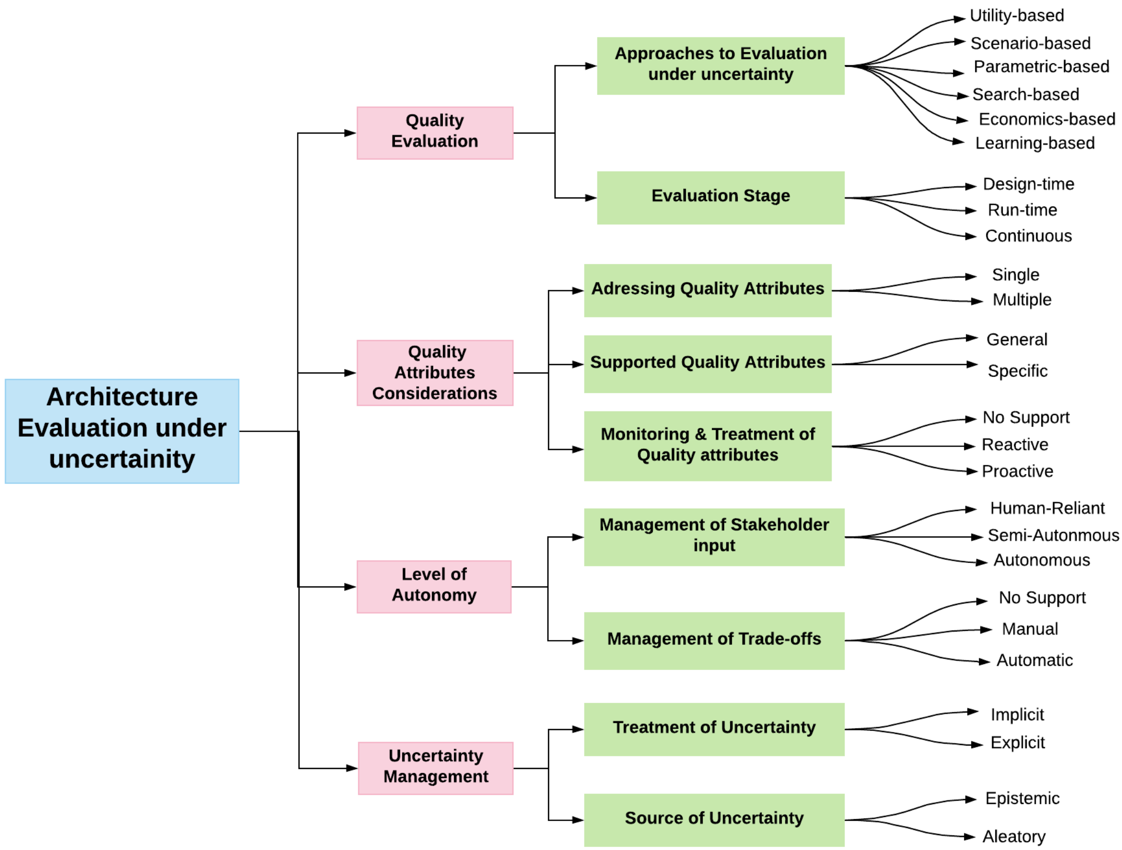

Fig. 4. The proposed classification of architecture evaluation approaches.

1. **Utility-based:** This category focuses on approaches to architecture evaluation methods that adopt utility functions for decision-making when justifying architecture design decisions, adopting a tactic and style among alternative candidates, *etc.* Utility functions are used in two contexts. First, it is a measure of the extent to which the candidate solution satisfy the set of quality attributes in question. Second, it can be used to provide a stakeholder's preferences over a set of quality attributes, which is called a *Weighted Utility function.* Various methods have adopted utility theory to shortlist the candidate architectures operating under uncertainty, such as [63, 95, 113].
   – Osterlind et al. [113] used utility theory to balance quality attributes against each other to obtain the best possible architecture.
   – GuideArch [63] is an architecture framework that explicitly models the uncertainty of architecture decisions using fuzzy logic to rank and determine the optimal architecture decision. However, the use of fuzzy logic cannot be empirically evaluated and adjusted.
   – Letier et al. [95] designed a method, based on GuideArch and CBAM, to deal with uncertainty. Utility theory and Monte Carlo simulation were used to calculate the costs and benefits of candidate architecture decisions under uncertainty. The latter approach made

an assumption that the probability distributions of model attributes are accurate; this may affect its applicability, particularly in dynamic environments.

- The architecture evaluation approach in [96] focuses on middleware and design pattern integration for developing adaptive self-managing architectures at design-time that is able to recover from failures. This approach suffers from the same limitation of design-time approaches: the design-time patterns (i.e. decision) may not be able to handle the changing environmental conditions at run-time. Architecture Software Quality Assurance (aSQA) [42] is an evaluation method that uses metrics to determine the user's satisfaction towards prioritized quality requirements, especially in agile software projects. Despite it focuses on a single point of evaluation to lighten the evaluation process, yet it misses the main aim of evaluation (i.e. assess the impact of architecture decisions on quality attributes).
- Decision-centric software architecture evaluation method (DACAR) [57] assesses the architecture decisions made or to be made independently using utility functions based on stakeholders' beliefs, rather than evaluating the whole architecture. The method could be potentially adopted in agile projects, since the architecture decisions could be evaluated as they appear in the process. However, the approach is not as flexible as scenario-based methods in obtaining the novel paradigms and significant change domains from stakeholders.
- Heaven et al. [75] reported on an approach tailored for self-managed software systems. The approach provides the following features: high-level task planning, architecture configuration and reconfiguration, and component-based control. Their approach uses weighted utility functions to represent quality attributes and determine the total utility of configurations by taking into account reliability and performance concerns.
- Esfani et al. [62] proposed an approach that elicits from stakeholders their beliefs regarding uncertainty with respect to attributes such as network bandwidth. In particular, the stakeholders provide an estimate for the range of uncertainty with respect to the expected level of input variation. The approach also quantifies the uncertainty through profiling by comparing the actual values with estimates from stakeholders and hence provides probability distributions for the variation in data collection. After that the overall uncertainty is computed using fuzzy math.
- Veritas [67] is another utility-based approach which adopts utility functions for the management of run-time test cases to improve the adaptation procedure.
- Cooray et al. [46] proposed a proactive approach, which continuously updates reliability predictions in response to environmental changes. The approach has proved its efficiency in adapting the system before it experiences a significant performance drop. However, the approach does not consider cost and suffers from scalability issues.
- Models@run.time [22, 39, 107] includes built-in mechanism for evaluating the behaviour of software systems through continuous monitoring, planning, and model transformation. However, the effort was not discussed from the architecture evaluation angle. In particular, the authors state that "models of the functional and/or non-functional software behaviour are analysed at run-time, in order to select system configurations that satisfy the requirements" [29]. Models@run.time operates on the assumption that possible run-time configurations have already been evaluated and encoded in the system, where evaluation can be an afterthought through profiling configurations and recommending alternatives. It aims to "reevaluate requirements satisfaction while the system is evolving" [54]. In the spirit of models@run.time, several approaches which are architecture-centric have been discussed in the context of self-adaptive and managed architectures [36, 47, 68, 91, 131].

Examples of these approaches include [7, 30, 40, 46, 58, 69], which formally analyse their architectural models.

- The Rainbow framework [40] uses Markov processes to determine the likely aggregated impact of each strategy on each quality attribute. It requires high human intervention to determine the effects of strategies with respect to quality attributes (i.e. predefined probabilities) [41].
- Epifani et al. [58] proposed a utility-based approach leveraging a Discrete Time Markov Chain approach and Bayesian estimators to provide continuous automatic verification of requirements at run-time and support failure detection and prediction. Their approach does not consider multiple quality attributes, switching cost, and variance in run-time data.
- In [104], Meedeniya et al. proposed a Discrete Markov Chain approach that performs MonteCarlo simulations to predict the reliability of heterogeneous software architectures. The approach also adjusts the number of architecture evaluations with respect to particular performance levels. They then extended the work to deal with different sources of uncertainty, which occur in different software architecture evaluation models [105]. One major concern in this approach is its assumption that all software architectures can be modelled as Markov chains, which may not be true in some contexts due to complexity.
- Ghezzi et al's [69] method is one of the few that complement design-time with run-time analysis. At design-time, the approach integrates goal-refinement methodologies with Discrete Markov Time Chains to determine all possible execution paths for the goal. At run-time, it exploits utility functions to measure the utility of paths, based on assumptions. For example, the utility for a 5ms response time is 1 and so forth. Given these assumptions, a hill climbing algorithm is used to select the optimal goal. We will discuss [7, 30, 46] in learning-based section.
- Other utility-based approaches are found in Table 6.

Summary and Reflection: *Generally, the major problems of the prior approaches are: (i) the high reliance of stakeholders for utility estimations, which is subject to their experience; (ii) the utility functions are hard to define; (iii) there is complexity and uncertainty in the quantification of utility values. This motivated the need to integrate learning techniques to learn over time and hence improve the analysis (discussed in the learning-based section).*

2. **Scenario-based:** The foundation of most architecture evaluation approaches rests on scenarios [27, 53, 122]. These approaches use quantitative evaluation to determine the fitness of operational quality attributes. They elicit from stakeholders the utilities of architecture decisions and their effect on quality attributes of interest. Some of the scenario-based approaches have been validated and used in industry over the past decades [53].
   - *Software Architecture Analysis Method (SAAM) [83]:* is the first well-known architecture evaluation method that aimed to reify quality attributes via a set of scenarios as a means to evaluate architecture design decisions under concern and identify risks in an architecture. It assesses the extent to which the architecture satisfies the quality goals. It was originally used for assessing modifiability, but it has been applied for other quality attributes, such as portability and extensibility. SAAM takes as input: business goals, software architecture description, and quality requirements that illustrate the interaction between stakeholders and the system being analysed. It then maps between scenarios and architecture components to assess anticipated changes to the system. This mapping can also be employed to estimate the amount of effort needed to handle these changes. The SAAM does not explicitly

deal with trade-offs between quality attributes. The lack of trade-offs management has contributed to the evolution of Architectural Trade-off Analysis Method.

- *Architectural Trade-off Analysis Method (ATAM) [43]:* is the most popular architecture evaluation method. It is an evolved version of SAAM. Unlike SAAM, the ATAM focuses on a comprehensive evaluation of quality attributes rather than just concentrating on modifiability, portability, and extensibility. ATAM is a generic design-time architecture evaluation method that uses scenarios to assess the value of architecture design decisions. Specifically, it aims to reveal the degree to which an architecture will meet its quality requirements (e.g. availability, security, usability, and modifiability), and the interaction between those goals through trade-off analysis.
- *Cost-Benefit analysis method (CBAM) [82]:* is an architecture evaluation method that extends ATAM to provide cost/benefit analysis of architecture design decisions. The CBAM was created to "develop a process that helps a designer choose amongst architectural options, during both initial design and its subsequent periods of upgrade, while being constrained to finite resources" [9]. Although CBAM uses cost/benefit information to value the architecture design decisions and to justify their selection, this method is unable to dynamically profile the added value of architecture decisions, which is essential for applications operating in uncertain environments (such as IoT). It only deals with uncertainty through set of scenarios, similar to ATAM.
- *Scenario-Based Architecture Re-engineering (SBAR) [18]:* is another scenario-based architecture evaluation method that uses different techniques to assess the quality attributes of interest and implicitly deal with uncertainty: scenarios, simulation, and mathematical modeling. For example, if a quality attribute is concerned with development and design-time properties, such as maintainability and reusability, scenario-based techniques can be best utilized. Scenario-based analysis can be still used for behavioral and run-time properties, such as performance and fault-tolerance, simulation and/or mathematical models can better provide meaningful insights and can complement scenario-based ones. A major concern in SBAR is its use of impractical assumptions. For instance, to address the reusability concern, the architect has to define all the scenarios related to the reuse of parts of the architecture, which is not feasible.
- *Architecture-Level Modifiability Analysis (ALMA) [19]:* Unlike ATAM and CBAM, ALMA focuses on a single quality attribute, and hence it does not consider trade-offs. It utilizes probabilities to determine the likelihood of the impact of scenarios at the software architecture level with respect to modifiability concern (e.g. maintenance cost prediction and risk assessment).
- *Systematic Quantitative Analysis of Scenarios' Heuristics (SQUASH)* [79]: is a systematic quantitative method for scenario-based value, risk, and cost analysis. The method focuses on evaluating the relative benefits of proposed scenarios in early stages of architecting. The method extends some steps from CBAM by providing extensive evaluations of the internal structure of the scenarios to predict the quality attributes of architecture decisions. In this context, the approach relies more on stakeholders than CBAM and hence it may not be easy to apply in practical settings.
- *Analytic Principles and Tools for the Improvement of Architectures (APTIA) [84]:* is an architecture improvement method that combines existing architecture evaluation methods (such as ATAM, CBAM, *etc.*) through: "quality attribute models, design principles in the form of tactics, scenario-based quality attribute elicitation and analysis, and explicit elicitation of the costs and benefits of architecture decisions from stakeholders" [84] as well as the use of architecture documentation templates. It also adds new steps to the analysis. Particularly, it

identifies design decisions linked to the analysis rather than stating their future problems. It was able to aid the team of architects to propose architecture design decisions for a complex system and in a short period of time.

– *Architectural Tradeoff Method using Implied Scenario (ATMIS) [64]:* is an extension of ATAM through the adoption of Implied Scenarios for security testing [3]. The main aim of this approach is to apply trade-off analysis between security and any other quality attribute through the use of implied scenarios.
– Further, there is another scenario-based method which is different from the commonly used scenario-based architecture evaluation methods. The method is named Performance Assessment of Software Architecture (PASA) [139]. In PASA, the architect uses the architecture specification to form performance models. The generated models are then utilised to assess whether the performance objectives are met. ATAM uses scenarios to determine, prioritise and refine the key quality attributes by constructing a utility tree, where each leaf in tree represents a scenario. PASA instead employs scenarios in the form UML and sequence diagrams to demonstrate how the software architecture will achieve the performance objectives.
– Finally, Yang et al. [141] proposed a utility-based approach that extends the scenario-based approaches (e.g. ATAM, CBAM) and profiles the run-time information to better manage the QA trade-offs. It aims to improve decision-making and handle the uncertainty which may be better managed at run-time. In particular, their approach determines the potential QA trade-off points, designs the adaptive architecture decisions, and finally deploys their system on a middleware platform to collect run-time information. Though the latter approach is one of the few attempts to extend scenario-based approaches at run-time, it lacks the ability to learn over time and hence cannot forecast the future potentials of architecture decisions.

Summary and Reflection: *Scenario-based evaluation approaches can be described as best-effort, where the evaluators' expertise, choice of stakeholders,* etc., *are all factors that influence the evaluation. In particular, these approaches heavily rely on human inputs and expert judgement. These processes can thus suffer from subjectivity, bias and can never be complete. As for the their effectiveness for evaluating for uncertainties, these methods advocate the use of exploratory, growth and stress and the like of scenarios that can test for the likelihood of an issue (e.g., sudden spike in load; downtime in part of the network; hostile attack, etc) to be confronted by the architecture along its response and quality trade-offs affected and the soundness of the architecture design decision and choices in responding to these issues. The choice of these scenarios can be critical input to the evaluation process and its conclusion on the extent to which the architecture can be resilient to uncertainty. Henceforth, the soundness of the evaluation for uncertainty can be influenced by human expertise, judgement and their skills and experience in identifying of uncertainty revealing scenarios to steer the evaluation exercise.*

3. **Parametric-based:** The previous scenario-based approaches used simplistic mathematical models and relied heavily on stakeholders for the elicitation of scenarios and on expert evaluators for the impact of these scenarios on quality attributes. Here, we will discuss approaches that assess architecture decisions using parametric models - parameterised mathematical models with parameters identified and supplied that can aid decision-making. Stakeholders often provide values for these parameters (i.e. design-time and interactive approaches) or can be provided or calibrated at run-time through observing relevant concerns of the parameterised functions.
   – Analytic Hierarchy Process (AHP) [123] is a mathematical modelling tool used in dealing with complex decision-making. AHP has been used in two contexts for architecture evaluation: managing trade-offs and determining the relative importance of scenarios and

decisions. Zhu et al. [145] adopted AHP to explicitly determine the trade-offs being made and the relative size of these trade-offs. It has been used with CBAM to determine the relative importance of scenarios through pair-wise comparisons [94]. It relies on eliciting the benefits and costs from stakeholders, and hence suffers from the same limitations of scenario-based approaches.

- ArchDesigner [5] is an architecture framework that first adopts AHP to elicit from stakeholders their preferred architecture decisions. It then uses Integer programming to determine the optimal architecture decision, which satisfies conflicting stakeholder quality goals subject to project constraints, such as cost and time.
- LiVASAE [86] (a lightweight value-based architecture evaluation technique) attempts to measure the level of uncertainty using AHP and also provides three simplified evaluation procedures as compared to the CBAM. All these approaches rely on stakeholders for evaluating the candidate architecture decisions as well as their benefits and costs.
- Other approaches include [40, 58, 103–105] (mentioned in utility-based approaches section), can also satisfy the parametric-based evaluation, as they use Markov Chains to determine the QoS of architectures.

Summary and Reflection: *Though all the prior approaches provide some management for uncertainties, they suffer from the same concerns: the high reliance on stakeholders for the elicitation of the relative importance (i.e. rank) of architecture decisions and their impact on quality attributes.*

4. **Search-based:** This category focuses on showing how search-based techniques have been used to complement architecture evaluation (but not related to work on search-based techniques in software architecture unless the work is evaluation-related). Search-based software engineering is "the application of metaheuristic search techniques, such as genetic algorithms, simulated annealing and tabu search" to the analysis [73]. In software architecture, it is used to solve complex problems in terms of searching for the most suitable (i.e. optimal) candidate architecture choice [73]. In this context, it is sometimes called search-based optimisation [74].
   - Evolutionary Algorithms are generally adopted for decision-making in software systems [7]. For example, ArcheOpetrix [6] is a tool that exploits evolutionary algorithms for multi-objective optimization of an embedded system's architecture.
   - Grunske et al. [70] proposed a method to automate the trade-off management process using an evolutionary algorithm. The aim of the approach was to rank design decisions (architecture refactorings) by taking into consideration competing quality goals. However, this was an initial attempt without a complete evaluation (i.e. it has not been applied on architecture evaluation methods).
   - As aforementioned in utility-based section, Ghezzi et al's [69] method uses a hill climbing algorithm to select the optimal goal, which could also be seen as a search-based technique.
   - Among the notable excluded work is [6], as the work does not explicitly or implicitly address *uncertainties* in architecture evaluation though they have covered some phases of design-time and run-time evaluation. However, we have included their subsequent work [102] as it addresses uncertainty in architecture evaluation decision-making. In particular, Meedeniya et al. [102] proposed a Robust ArcheOpterix framework that can determine the uncertain information related to system parameters and hence search for the most optimal and robust candidate architecture. The framework provides the architect with the flexibility to choose the most suitable optimisation algorithm from the following list [101, 118]: Multi-Objective Genetic Algorithm (MOGA), Non-dominated Sorting Genetic Algorithm (NSGA-II), Pareto Ant Colony Algorithm (P-ACO), Simulated Annealing (SA),

Hill Climbing, Bayesian Heuristic for Component Deployment optimization (BHCDO), Random Search Algorithm, and Brute-Force Algorithms. The used software architecture evaluation model is based on their previous work [104].

- PerOpteryx [28] is an automated tool based on Palladio framework [120] for selecting the optimal candidate architecture. This approach performs evaluation at design-time. So still run-time monitoring is important to complement the design-time decisions. However, we see potentials in extending PerOpteryx tool with run-time analysis to develop the continuous evaluation framework.
- Other approaches include [5, 62, 63, 67, 95, 103, 127] (mentioned in other sections), can also satisfy the search-based evaluation, as they adopt some search-based algorithms for the analysis.

Summary and Reflection: *Search-based techniques, which are fundamentally optimisation-based, have been used to evaluate architecture design decisions and choices. These techniques often rely on the assumption that fitness functions guide the search. These techniques suffer from the following limitations: stopping criteria for the search is often difficult to confirm with confidence and solutions tend to provide "good enough" optima. Additionally, as much of the work on architecture evaluation are scenario-based, mapping the concerns of the scenarios into search-based objective functions along their constraints can be complex to abstract if one would be seeking a search that would reflect on these scenarios. Nevertheless, search-based techniques can be specifically useful if one would use the search and evolutionary techniques to generate new styles and architecture configurations that could better meet the requirements of interest.*

5. **Economics-based:** This category presents approaches that inform architecture evaluation using economics and finance inspired methods; these approaches quantitatively evaluate the worthiness, short- and long-term benefits, option, risks and costs of the architecture design decisions. Though these approaches can be essentially utility-based and/or parametric, we are discussing the economics-driven approaches that were utilised in steering these efforts. In most cases, economics-based approaches have been used to evaluate the architectures at design-time.
   - Traditional cost-benefit analysis methods have been used to evaluate software. For instance, Cellini et al. [34] computed the net benefit of a software through the deduction of total costs from total benefits. These attributes have been obtained from software architects through a group of questions (e.g. "what is the state of the world in the absence of the program ?"). CBAM [82] is a utility-based architecture evaluation method that uses cost-benefit to analyse the impact of architecture decisions on quality attributes of interest. This approach partially capture uncertainties which motivated the need to integrate some finance-inspired approaches into the software engineering field. Boehm [23, 24] was among the first to introduce economics and finance theories to evaluate software design decisions. Examples of these approaches: Net Present Value (NPV) [56, 97], Modern Portfolio Theory (MPT) [100], and Real Options Analysis (ROA) [8] (which will be discussed afterwards).
   - Recently, the approach in [136] proposes an architecture evaluation approach inspired by CBAM [82] for run-time decision-making in self-adaptive systems that considers benefits and costs of decisions. The approach adopts a weighted utility measure of the qualities that the adaptation decisions can provide to the stakeholders. Although this approach seems to provide continuous evaluation, it requires additional elements, such as online machine learning techniques, and extra experimental evaluation for applicability and efficiency.

   Real Options Analysis and Modern Portfolio theory have been used to inform that analysis of software architecture in the presence of uncertainty. Though they have been used in various software engineering and design domains, such as [14, 60, 128, 132], to evaluate low design decisions (e.g. modularity in design) using economics-based thinking; they were not concerned with architecture evaluation. There are other few works (e.g. [12, 13, 112, 136]) which initiated the use of economics-based techniques in architecture evaluation. In this context, in Table 10, we outlined the software architecture evaluation-related approaches (e.g. [12, 13, 112, 136]) as they operate on widely used architecture frameworks such as ATAM and CBAM, obtained from our search results and satisfy our inclusion/exclusion criteria.

– *Net Present Value (NPV) [56, 97]:* is a popular approach used to value software. It values the software project by eliciting the probability of investing in an established discount rate or interest. A positive NPV indicates that its financially beneficial to invest (i.e. deploy this architecture decision) and negative NPV is the opposite. It has been originally used in [49, 65].
– *Modern Portfolio Theory (MPT) [100]:* was first introduced by the Nobel prize winner Markowitz in 1950s. MPT aims to improve the decision-making process by allocating capital to a portfolio of diverse investment assets. MPT handles uncertainty through the distribution of capital among assets to minimize risk and maximize the returns. In particular, it provides a weighted combination (i.e. portfolio) of the assets, where the weight denotes the investor's share of capital in each asset. In this context, MPT seeks to demonstrate the rewards of having a diversified portfolio of assets. MPT is well-known in finance domain and has been also introduced in software engineering domain as a means to deal with uncertainties. In software architecture [112], it has been adopted with CBAM to determine which portfolio of architecture decisions will deliver value by considering sustainability dimensions. Although this approach explicitly deals with uncertainty, yet it provides a short-term value. It does not embed flexibility as real options analysis.
– *Real Options Analysis (ROA) [8]:* provides an analysis paradigm that emphasizes the value-generating power of flexibility under uncertainty. An option is the right, but not the obligation, to make an investment decision in accordance to given circumstances for a particular duration into the future, ending with an expiration date [134]. Real options are typically used for real assets (non-financial), such as a property or a new product design. ROA treats uncertainty as an option which may provide future opportunities to the project, which could be exercised when it provides a high option value. On the contrary, MPT specifically deals with financial assets and considers uncertainty as a risk that should be minimized. Real Options analysis has been used in software architecture in [12, 13, 114]. Bahsoon et al. [12] used real options analysis along with CBAM to measure the architecture's stability. They then used their method to value scalability in distributed architectures [13].

Summary and Reflection: *NPV has been discouraged, because it ignores the value of the flexibility under uncertainty [14, 59, 129]. Modern Portfolio Theory provides some treatments for uncertainty, but for short-term evaluation. On the contrary, Real Options analysis methods could be used as a way to manage uncertainty on the long-term. Further, in software architecture evaluation, few methods embedded finance-inspired techniques to their analysis. However, we see great potentials for including these techniques to the evaluation especially in high dynamic and unpredictable environments.*

6. **Learning-based:** We define learning-based architecture evaluation methods as methods which adopt machine learning techniques to improve the evaluation. In most cases, learning-based approaches have been used to evaluate the architectures at run-time. "The effectiveness of model-based reasoning about the properties of a software system depends on the accuracy of the models used in the analysis" [29]. For example, some models may become obsolete

due to evolution in the software architecture. The same applies to the use of utilities for evaluation and decision-making. Therefore, machine learning could be adopted to better enhance the evaluation through profiling the observations of the system properties over time, as in the following studies [31, 61, 87, 106, 133].

– In the context of using reinforcement learning techniques, Tesauro et al. [133] integrated queuing policies with reinforcement learning, forming a hybrid approach to enhance the dynamic resource-allocation decision-making process in data centers. The approach suffers from scalability and performance overhead. A reinforcement learning online planning technique was used by Kim et al. [87] to improve a robot's operation with respect to changes in the environment, by dynamically discovering the appropriate adaptation plans. However, it does not continuously evaluate the cost-effectiveness of architecture decisions over time. These approaches [87, 133] tend to be domain-specific. Further, Calinescu et al. [31] proposed initial attempts for the use of Bayesian learning and ageing coefficients to update the model parameters, where the ageing coefficients may be a useful element for a continuous evaluation approach. Because it may then allow the architect to tune the sensitivity of approach to present/past observations. Though their work had potential, it was still work-in-progress (i.e. initial evaluation for the approach has been performed and hence it requires further analysis).
– FUSION [61] is another learning-based approach that adopts a machine learning algorithm named Model Trees Learning (MTL) to tune the adaptation logic towards unpredictable triggers, rather than using static analytical models. It also uses utility functions to determine the benefit of models in question. The major benefit of FUSION is its ability to learn over time and improve the adaptation actions due to the promising learning accuracy. However, FUSION has the following limitations: (i) it is specifically tailored to feature modelling; and (ii) it only detects goal violations, i.e. constraints, but does not have the ability to check if the current architecture option is getting worse.
– In [127], a run-time architecture evaluation approach has been proposed, which is suited for systems that exhibit uncertainty and dynamism in their operation. The method uses machine learning and cost-benefit analysis at run-time to continuously profile the architecture decisions made, to assess their added value. This approach is considered as a *reactive* approach, as it ignores the future potentials of architecture decisions. This approach is considered as one of the few attempts which explicitly evaluates software architectures at run-time.
– Moreno et al. [106] proposed a proactive latency-aware adaptation approach that constructs most of the Markov Decision Processes offline through stochastic dynamic programming. Their method focuses on optimizing the latency of adaptation action based on forecasts, without considering the cost of architecture decisions and multiple stakeholder concerns.

Summary and Reflection: *The use of machine learning in architecture evaluation can be challenging. First, formulating the evaluation as a learning problem requires data that relate to historical observations along with data evaluation for recency, decay, relevance, etc. Second, the problem with any study involving machine learning is that the results may not generalise to other data sets, therefore, the methods should be tested on various data sets with different input parameters. Further, comparative studies should be provided to confirm the validity of the model. Accuracy and error metrics should also be adopted to determine how far are the forecast values from the actual ones. The selected measures should be unbiased towards under or over estimations. Additionally, the software architecture community can benefit from guidance on the type of learners that can be best suited for the evaluation of software architectures under uncertainty, yet such guidance is lacking and bridging efforts are still needed.*

4.2.2 Stage of Evaluation. The evaluation could occur at *design-time* and/or *run-time*. Design-time evaluation occurs before system deployment, where the stakeholders are more involved in reasoning the system under study, whereas the run-time evaluation approaches use run-time and/or simulated data (e.g. QoS) to capture the dynamic behaviour of architecture decisions under uncertainty and use such information to profile or evaluate design decisions either during the prototyping stage or post-deployment.

1. **Design-time Evaluation:** The design-time evaluation of software architectures aims at eliciting a proper specification of the problem, which is the first step on the path of analysing architecture decisions for suitability.

– Documented efforts on systematic design-time architecture evaluation approaches are best linked to the seminal work of [19, 82, 83, 85]. These approaches focus on identifying design decisions that best fit the quality requirements of interest and their trade-offs using scenarios (i.e. *scenario-based* approaches).

– Other examples of design-time approaches are treated as *utility-based* (e.g. [18, 19, 63, 79, 82, 84, 95, 139, 145]), *parametric-based* (e.g. [5, 18, 84, 103, 145]), *search-based* (e.g. [5, 28, 63, 95, 102, 103]), and *economics-based* (e.g. [12, 13, 112, 114]). Since learning-based approaches require run-time analysis, therefore, we have not found methods which are learning-based. Table 10 summarises the included studies related to design-time architecture evaluation approaches with respect to the proposed classification.

Summary and Reflection: *Design-time evaluation has received significant attention over the years and the subject is a relatively mature area. However, as we can see from the various discussed methods, the evaluation is essentially human-reliant and the treatment for uncertainty has been left to the evaluators; this can include their choice for the scenarios to steer the evaluation, the adopted models, stakeholders involved, etc. The process can then suffer from subjectivity, bias and can never be complete. Therefore, a systematic design-time evaluation approach that explicitly deals with uncertainty rather than either relying on ad hoc evaluation or implicit mitigation of uncertainty is necessary.*

2. **Run-time Evaluation:** By run-time evaluation, we refer to approaches that use run-time and/or simulated data (e.g. QoS data) to capture the dynamic behaviour of architecture decisions under uncertainty and to use such information to profile and evaluate design decisions. Table 11 summarises the run-time architecture evaluation methods studied.

   – In software architecture evaluation, utility functions are commonly used to select the optimal architecture option. This approach has also been adopted to determine the stakeholder's preferences towards quality attributes of interest. Therefore, it is utilized as a way to model trade-offs between quality attributes. Utility functions have been used at run-time (i.e. *utility-based*) for self-adaptive and self-managed systems, such as [38, 40, 46, 61, 62, 67, 69, 75, 136, 141].

- Other run-time evaluation approaches apply some machine learning techniques (i.e. *learning-based*) to improve the decision-making process through profiling the observations of the system properties over time, as in the following studies [31, 61, 87, 133].
- [141] is one of the few attempts to extend *scenario-based* approaches at run-time. As mentioned earlier, this approach lacks the ability to learn over time and hence cannot forecast the future potentials of architecture decisions.
- To the best of our knowledge, there are no economics-based approaches that evaluate architectures at run-time.

Summary and Reflection: *As far as we know, the majority of run-time evaluation approaches rely on models for the analysis, which may be subject to scalability and complexity concerns. For that, these approaches have adopted some machine learning algorithms, such as Reinforcement learning, to update their models at run-time. Despite their potential, these approaches suffer from the following limitations: (i) they assume that the quality data about architecture decisions is available at every timestep, which may not be true in non-stationary environments such as IoT; and (ii) they lack the capability for checking whether the current architecture decision is getting worse. However, the proposed method in [127] has provided some techniques to handle the above concerns but still requires further investigation and more techniques are needed to enhance the evaluation. The most important component in a continuous evaluation approach is the run-time approach to be included. Some of the above approaches (e.g. [55, 127]) seem to provide important elements for a run-time approach in terms of providing learning techniques. These techniques could aid the architect in predicting the impact of architecture decisions on quality attributes under different scenarios of interest. On the contrary, few of the approaches were explicitly used in the context of software architecture evaluation (e.g. [127]).*

3. **Continuous Evaluation:** We define continuous evaluation as multiple evaluations of the software architecture that begins at the early stages of the development and is periodically and repeatedly performed throughout the lifetime of the software system.
   - Continuous Performance Assessment of Software Architecture (CPASA) [117] is one the few explicit attempts for continuous evaluation. It is an extension of PASA, with an explicit focus on deployment in agile development process. It provides an interactive system that aids the architect in the automatic assessment of performance attributes through modelling of architecture decisions. They define "continuous" assessment as the production of continuous performance evaluation tests. Despite the attempts in PASA and CPASA to handle cost-benefit trade-offs, (i) the evaluation was incomplete; (ii) they are not using any run-time information to refine their architecture decisions; and (iii) it lacks run-time monitoring and forecasting of the performance of architecture decisions. In such cases architecture is, at best, a modelling tool, which may (or may not) be applicable in dynamic environments. Therefore, these approaches are still design-time evaluation approaches.
   - Further, the approaches proposed in [69], [136] and [127] could seem to provide some initial attempts for continuous evaluation, but they suffer from the concerns mentioned in (ii) and (iii).

Summary and Reflection: *Continuous architecture evaluation approach starts at design-time and continues to operate at run-time, with design-time architecture evaluation being at its earlier stages. Continuous evaluation shall provide built-in support to deal with operational uncertainties and dynamicity, starting from design-time by predicting run-time behaviour and while calibrating its evaluation at run-time and post deployment. Continuous evaluation can leverage machine learning to provide predictive and proactive diagnostic capabilities; however, such improvement requires data that can relate to the architecture design decisions, quality attributes performance, that might not be always available or easy to extract from operational and maintenance logs. In the absence of real-time data, the evaluation can, for example, benefit from info-symbiotic[a] simulations and digital twins capabilities to improve the prospect of the evaluation in dealing with uncertainties.*

[a] a term that is widely used by the dynamic data driven simulation system community (e.g. http://1dddas.org/InfoSymbiotics/DDDAS2020, https://sites.google.com/view/dddas-conf/home)

4.3 Quality Attribute Considerations

4.3.1 Addressing quality attributes: There are some evaluation methods which focus on a *single* or *multiple* quality attributes. Based on the results, we have found that most of the software architecture evaluation studies' have addressed multiple quality attributes, e.g. modifiability with portability and extensibility [83], stability with cost [12]. Other examples of studies are found in Table 7- 9.

4.3.2 Supported Quality Attributes: we categorised the quality attributes supported into: *general* and *specific*. For *general*, we consider the literature that discusses the support of any quality attribute, such as performance, availability, reliability, *etc.* For instance, some studies propose generic methods (62% of included studies) that can be generic enough and applicable to various quality attributes. However, there are others that focus on *specific* quality attributes (e.g. performance only, i.e. any QA) – 36% of included studies, whereas others focus on cost only – 49% of included studies. Based on our review, for the approaches that evaluate the software architecture at design-time, some studies (e.g. [43, 70, 79, 82, 84, 145]) accept generic quality attributes, whereas others focus on specific quality attribute (e.g. [12, 18, 19, 79, 82–84, 96, 104, 145]). One remarkable investigation is that very few run-time architecture evaluation approaches consider costs through the evaluation process (e.g.[40, 61, 127, 136, 141]), as well as most of the run-time approaches evaluate with respect to specific quality attributes (e.g. performance and energy consumption [32], and reliability and performance [30, 31, 75]). As for the few continuous approaches, their proposed techniques could be applied for generic quality attributes. Other examples of studies are found in Table 7- 9. The way existing evaluation methods consider cost and value is not done in isolation but in alignment with the qualities under consideration and their trade-offs. Our review holds examples from mainstream architecture evaluation methods (e.g. [5, 34, 79, 82, 84, 139, 145]). Other examples are shown in Table 7- 9.

4.3.3 Monitoring and treatment of quality attributes: This criterion is relevant to run-time and continuous evaluation approaches where quality attribute values are either determined through run-time monitoring or through prediction. Similarly for the treatment, there are two types [72, 93, 121]: *reactive* and *proactive*. A reactive approach triggers a switch after experiencing a drop in performance, a goal violation, *etc.* A proactive approach switches architecture options without experiencing a drop in performance; instead it is based on predictions that a significant change in performance may occur in the near future. Based on our investigation, most of the approaches used reactive monitoring and treatment of quality attributes (e.g. [30, 31, 61, 62, 67]), whereas very few approaches embedded proactivity to their architecture evaluation method (e.g [46, 69]). Other examples of studies are found in Table 7- 9.

Summary and Reflection: *Quality Attributes continue to be the driver for architecture evaluation to test the architecture fitness with respect to the considered attributes. Considering multiple quality attributes and their simultaneous effect on the architecture is still a challenging task, if the evaluation would consider uncertainties that relate to the provision and support of these attributes. Research has also to look at how the evaluation can consider multiple source of uncertainties that can relate to the simultaneous provision of these attributes. Research can benefit from search-based and evolutionary computing to provide the basis for automatic refinements of architecture in supporting quality attributes and embracing for various sources of uncertainties. The challenge, however, is to construct sound fitness functions and stopping criteria for managing the search. The support can goes beyond the classical monitoring and reactive interventions to provide a holistic approach for proactive and preventive diagnostic of software architecture, while having multiple qualities and their corresponding source of uncertainties, as first class citizen in the evaluation.*

4.4 Level of Autonomy

4.4.1 Management of Stakeholder Involvement in Evaluation: This category has been further categorised to *human-reliant*, *semi-autonomous*, and *autonomous*. We have to distinguish between: (i) Human-reliant (i.e. totally dependent on stakeholders for evaluating the behaviour of candidate architecture options); (ii) Semi-autonomous process for architecture evaluation, with human in the loop (e.g. stakeholders and architects in the loop for interactive evaluation); (iii) Autonomous (i.e. the evaluation is performed autonomously without human intervention). To further clarify those categories, we consider the case of architecture evaluation in self-adaptive Systems (SAS): there are *human-reliant* architecture design decisions (such as whether to introduce a self adaptation mechanism), *semi-autonomous* (such as human in the loop participation in self-adaptive systems [33]), and *autonomous* architecture design decisions (such as the SAS adapting and deploying components to different servers at run-time). Another example of the use of autonomous architecture design decisions is the incorporation of intelligent and learning mechanisms, evolutionary computations, *etc*, to assist in the automatic evaluation of decisions. Continuous architecture evaluation can monitor QAs and suggest re-configuration from a repository of candidate options, some of which their technical viability has been established but requires further profiling and confirmation. The evaluation process can then learn and suggest a suitable configuration; it can also call for further refinements and/or phasing out of reconfiguration.

For classical design-time architecture evaluation approaches (e.g. scenario-based), most of them tend to fully involve the stakeholders to their analysis, e.g. ATAM, CBAM, ATMIS, *etc*. Other design-time approaches (e.g. utility-based, economics-based and search-based) are semi-autonomous, such as [5, 12, 70, 95, 105, 113], in the context of requiring some inputs (e.g. utilities, users' satisfaction towards quality attributes, QoS constraints, *etc*) for evaluation from the architect. Since that run-time architecture evaluation approaches occur at run-time (e.g. learning-based), most of these approaches are autonomous (e.g. [31, 32, 40, 87, 133]), whereas few of them require some human involvement (e.g. [30, 75, 141]). Other studies are depicted in Table 7- 9.

4.4.2 Management of Trade-offs: A common problem in selecting an optimal architecture decision is the management of trade-offs [21]. For example, an architecture decision concerning a sensor could provide high response time but with low energy efficiency. So one objective could be to select an architecture decision that can satisfy both quality attributes. There are two types of trade-off management: *manual* and *automatic*. *Manual* management denotes the adoption of tools or techniques that require human-intervention, whereas *automatic* indicates the use of parametric models that automatically select and/or shortlist trade-off candidates. Some of the design-time architecture evaluation approaches (e.g. ATAM, CBAM, ATMIS, and APTIA) handle trade-offs manually through the analysis of trade-off points elicited from stakeholders or do not consider it at all (e.g. SAAM, SBAR, SQUASH, and ALMA). As for the run-time architecture evaluation approaches, some run-time approaches provide automatic management of trade-offs, such as [30, 32, 40, 62, 87, 127], whereas one noticeable investigation is that many approaches have no support for trade-off management, such as [31, 67, 69, 75, 141]. Other studies are shown in Table 7-9.

Summary and Reflection: *Providing semi- or fully-autonomous and automated techniques for trade-off management is crucial in a continuous evaluation framework. In particular, research shall look at how the evaluation can support continuous and seamless management for various quality trade-offs and their corresponding uncertainties. In line with what we discussed in the quality attribute considerations section, the seamless management may need to consider simultaneous qualities, their inference, risks contributions and aversions. Additionally, the autonomous evaluation can operate at various views (e.g. 4+1 views [92]) of the architecture, where the evaluation can then converge to seamless negotiation of the various views for conflicts, reconcile these views while considering the various uncertainties within the architecture and across the views - the ultimate objective is to provide holistic seamless evaluation of the architecture.*

4.5 Uncertainty Management

4.5.1 Source of Uncertainty: As aforementioned in Section 1.1, architecture can experience two sources of uncertainty: *aleatory* and *epistemic* [15, 50, 66]. To summarise: *aleatory* conception of uncertainty intends that uncertainty arises from variability in possible realisation of a stochastic event, where unknown and different results could appear every time one runs an experiment under similar conditions; *epistemic* conception of uncertainty denotes the rise of uncertainty due to lack of confidence or missing knowledge to a fact which is either true or false. We analysed the works based on the sources of uncertainty it addresses.

- We found that most of the design-time architecture evaluation approaches address epistemic uncertainty (e.g. [18, 19, 43, 79, 82, 83]).
- Aleatory uncertainty is encountered in most of the run-time architecture evaluation approaches (e.g. [32, 40, 64, 87, 96, 133]).
- On the contrary, very few design-time (e.g. [64, 79, 104, 105]), run-time (e.g. [30, 40, 58, 61, 62, 136]), and continuous (e.g. [127]) approaches experience both epistemic and aleatory uncertainties.

4.5.2 Treatment of Uncertainty: In the research literature there are approaches that deal with *explicit* or *implicit* uncertainty. *Explicit* approaches are those that consider uncertainty to be a main focus whereas other methods which do not mention uncertainty, but their tools and techniques could be used to handle uncertainties (i.e. *implicit*). Next, we will summarise how the studied architecture evaluation approaches dealt with uncertainty.

- Uncertainties and risks, linked to the deployment, are implicitly discussed and mitigated through envisioning a set of scenarios, taking the form of use case, growth, and exploratory scenarios [43, 85, 90] as defined by the ATAM (a design-time architecture evaluation approach). A use case scenario reveals how stakeholders envision the system usage. A growth scenario illustrates planned and foreseen refinements to the architecture, whereas an exploratory scenario helps to probe the extent to which the architecture can adapt to future changes (e.g. functionality upgrades, new quality attribute requirements). Hence, the evaluation and its conclusions are highly dependent on the choice of these scenarios. The ATAM defines

Table 6. Summary of Contributions for the included studies with Respect to Approaches to Evaluation.

Stage	Approaches to Evaluation					
	Utility-based	Scenario-based	Parametric-based	Search-based	Economics-based	Learning-based
Design-time	[18, 19, 63, 79, 82, 84, 95, 139, 145] [5, 42, 57, 64, 70, 94, 96, 113] [86, 102–105]	[18, 19, 79, 82–84, 139, 145] [63, 95, 112, 114]	[5, 18, 84, 103, 145]	[5, 28, 63, 95, 102, 103] [102, 103, 105]	[12, 13, 112, 114]	-
Run-time	[30–32, 40, 62, 75, 87, 133, 141] [46, 58, 61, 67, 69, 106]	[141]	[40, 58]	[62, 67]	-	[61, 87, 106, 133]
Continuous	[69, 117, 127, 136]	[136]	[69]	[69, 127]	[127, 136]	[127]

and records the risks that may threaten the achievement of quality attribute goals. These include architecture decisions leading to subsequent problems in some quality attributes *(risks)*, architecture decisions where a slight alteration results in significant impact on quality attribute responses *(sensitivity points)*, and the simultaneous effect of a single decision on multiple quality attributes *(trade-off points)* [81]. ATAM focuses on the risks and benefits of architecture decisions and does not explicitly consider cost. CBAM extends ATAM by considering cost-benefit trade-offs.

– To summarise, for the design-time scenario-based architecture evaluation approaches, ATAM, CBAM, ATMIS, SQUASH and APTIA partially capture uncertainty through scenarios (as mentioned in the previous point), despite that they do not conduct evaluation at run-time. However, they suffer from the same drawbacks of design-time evaluation (i.e. high reliance on stakeholders). ATMIS is also specifically tailored for security. ALMA is similar to ATAM and CBAM, in the context of taking more utility-based perspective for the evaluation. It aids in performing architecture evaluation more systematically than SBAR. Scenario-based approaches do not provide explicit management for uncertainties, and include manual tools/techniques which may not be effective at run-time.
– For the other architecture evaluation approaches, some approaches provided explicit management of uncertainty through the use of probability distributions (e.g. [103, 104]), Fuzzy math (e.g. [62, 63]), Monte Carlo simulation (e.g. [95]), Modern Portfolio Theory (e.g.[112]), Real Options Analysis (e.g. [12, 13, 114]), ageing coefficients (e.g. [31, 127]), AHP consistency rate [86], utility theory (e.g. [41]), *etc.*

Summary and Reflection: *The treatment for uncertainties, its sources and management has been discussed in earlier sections in relation to qualities attribute management, trade offs, autonomy and covering various stages, techniques (e.g., utility-based, economics-based, evolutionary, search-based, etc) and various methods for evaluation(e.g., design and runtime). This is because the discussion and treatments for uncertainties is orthogonal to all the above and cannot be discussed in isolation of the solution domains. Interested reader can refer to the relevant summary and reflection sections. However, the software architecture evaluation community may need to develop common language and knowledge for eliciting architecture uncertainties at various levels and provide guidance from mitigating their consequence on the software architecture. The community may also identify various techniques for managing the uncertainties, covering various contexts, application domain, etc.*

4.6 Limitations of the Review

Though this review was developed following the typical systematic literature review methodology [88, 89, 116], there are some limitations that require clarification:

– The main threats to validity in this SLR is the selection bias when including the studies and extracting the data. To resolve that in terms of determining the relevant studies a research protocol (Section 2) was conducted. We applied this protocol to set out the objectives of the review, the necessary background, the research questions, inclusion and exclusion criteria,

Table 7. Summary of Contributions for Design-time Architecture Evaluation approaches with other categories.

	Category		**Representative Contributions**
Design-time	Addressing QA	Single	[19, 104, 139]
		Multiple	[18, 28, 43, 79, 82, 83] [70, 84, 86, 94, 102, 145] [5, 12, 28, 95, 113, 117] [13, 57, 103, 105, 112, 114]
	Supported QA	General	[28, 43, 79, 82, 84, 145] [63, 70, 86, 94, 113] [12, 57, 95, 105, 112, 117]
		Specific (Cost)	[5, 79, 82, 84, 139, 145] [42, 70, 86, 94, 117] [12, 13, 63, 95, 113] [112, 114]
		Specific (Other QA)	[18, 19, 64, 83, 96, 102] [12, 13, 103, 104, 139]
	Management of stakeholder input	Full	[43, 79, 82, 83, 86, 139] [18, 19, 84, 96, 145] [57, 64, 94, 117]
		Semi-Autonomous	[5, 12, 28, 70, 95, 105, 113] [13, 42, 63, 102–104, 112, 114]
	Management of Trade-offs	Manual	[43, 82, 84, 86, 145] [12, 13, 64, 94, 117, 139] [112, 114]
		Automatic	[5, 28, 42, 63, 70, 95, 102, 103, 113]
		No Support	[18, 19, 57, 79, 83, 96, 104, 105]
	Treatment of Uncertainty	Implicit	[18, 19, 43, 82, 83] [5, 70, 79, 84, 96] [57, 117, 139]
		Explicit	[86, 94, 95, 102, 113, 145] [12, 13, 28, 42, 64, 114] [63, 103–105, 112]
	Source of Uncertainty	Epistemic	[18, 19, 43, 79, 82, 83, 102, 139] [5, 64, 84, 86, 94, 117, 145, 145] [12, 13, 57, 63, 95, 113] [42, 70, 105, 112, 114] [18, 19, 28, 43, 82, 83, 104]
		Aleatory	[86, 94, 95, 113, 145] [12, 13, 42, 64, 102, 114] [63, 103–105, 112]

search strategy, data extraction and analysis of gathered data. The SLR protocol was arranged by one author and then revised by other authors to verify and evaluate the research questions and whether the search queries map to the review objectives and research questions. They also checked the relevance between data to be extracted and research questions.

Table 8. Summary of Contributions for Run-time Architecture Evaluation approaches with other categories.

	Category		Representative Contributions
Run-time	Addressing QA	Single	[58, 67, 133]
		Multiple	[32, 40, 46, 75, 87, 106, 141] [30, 31, 61, 62, 69]
	Supported QA	General	[40, 46, 61, 62, 69, 87, 106, 141]
		Specific (Cost)	[40, 61, 141]
		Specific (Other QA)	[30, 31, 58, 67, 75, 133]
	Management of stakeholder input	Semi-Autonomous	[30, 58, 62, 67, 69, 75, 141]
		Autonomous	[31, 32, 40, 46, 61, 87, 106, 133]
	Management of Trade-offs	Automatic	[30, 32, 40, 46, 61, 62, 87]
		No Support	[31, 58, 67, 69, 75, 106, 133, 141]
	Treatment of Uncertainty	Implicit	[32, 46, 58, 75, 141]
		Explicit	[31, 40, 62, 87, 133] [30, 61, 67, 69, 106]
	Source of Uncertainty	Epistemic	[18, 19, 43, 79, 82, 83, 114] [5, 64, 84, 86, 94, 112, 145] [12, 13, 42, 57, 63, 70, 95, 113]
		Aleatory	[32, 40, 64, 87, 96, 133] [58, 61, 62, 67, 69, 75, 106, 136, 141]
	Monitoring and Treatment of QAs	Reactive	[30, 32, 40, 75, 87, 133, 141] [31, 61, 62, 67]
		Proactive	[46, 58, 69, 106]

Table 9. Summary of Contributions for Continuous Architecture Evaluation approaches with other categories.

	Category		Representative Contributions
Continuous	Addressing QA	Single	-
		Multiple	[69, 117, 127, 136]
	Supported QA	General	[69, 117, 127, 136]
		Specific (Cost)	[117, 127, 136]
	Management of stakeholder input	Human-Reliant	[117]
		Semi-Autonomous	[69]
		Autonomous	[127, 136]
	Management of Trade-offs	Manual	[117]
		Automatic	[127, 136]
		No Support	[69]
	Treatment of Uncertainty	Implicit	[117]
		Explicit	[69, 127, 136]
	Source of Uncertainty	Epistemic	[117, 127, 136]
		Aleatory	[69, 127, 136]
	Monitoring and Treatment of QAs	No treatment	[117]
		Reactive	[127, 136]
		Proactive	[69]

- Several junior and senior researchers (with up to 15-30 years of experience in architecture evaluation) assessed and reviewed the SLR. They provided feedback which reduced the bias of the formalisation of the protocol, due to the selection of search keywords. There is still a risk of missing some related studies. This could occur in cases where software architecture evaluation keywords are not standardized and clearly identified. For instance, continuous evaluation is defined under different terms, such as continuous, run-time, dynamic, *etc.* Therefore, we made an agreement with each other about the definitions of unclear keywords. In some cases it was difficult to elaborate how the authors of reviewed studies interpreted terms such as continuous or run-time or dynamic (Section 2.3.1). In this context, we tried our best to include all the related terms that imply continuity. However, we cannot guarantee completeness.
- We also used a data extraction form to select information for answering research questions hence improving the consistency of data extraction (Section 2.6). To ensure that the findings and results were credible, we conducted a quality assessment on related studies (Section 2.5).
- The limited number of included studies might open a question about the completeness and coverage of the review, as compared to other SLRs (e.g. [7]). But the objective of this review was to focus on a specific goal, i.e. the state-of-the-art in software architecture evaluation approaches for uncertainty and to what extent continuous software architecture evaluation approaches are used. This results in a narrowed scope for the review. This is analogous to the case of [98] that conducted a review focusing on methods that handle multiple quality attributes in architecture-based self-adaptive systems (54 included studies), and [99] that studied the variability in quality attributes of service-based software systems (48 included studies). The narrow scope of SLRs explains the limited number of search results and included studies. We believe that the relevant studies to the research topic were indeed included. Further, the quality of conferences, journals, and books of the included studies ensures the significance of the analysis.
- In our search execution, some relevant studies may have not been shown in the search results of the bibliographical sources. This may be due to the fact that automated searches depend on the quality of the search engine. However, the selected bibliographical sources are considered the largest and most significant sources for conducting SLRs and the most used ones in software architecture and software engineering [27, 98]. We also performed manual and automated searches through the most popular venues for software architecture and software engineering [98]. Consequently, we are confident that the included studies are the most relevant and important ones and others are unlikely to be missed.
- We applied our search on meta-data (i.e. abstract, title, and keywords) only and some studies might have used architecture evaluation as a part of their proposed work without mentioning that explicitly in abstract, title, and keywords. Since the authors identify the meta-data of their studies, therefore, our included studies depend on the quality of the bibliographical digital sources in classifying and indexing studies.
- One of the main threats to validity is the validation of the classification framework. In this context, the development of the classification framework was guided by a method for building taxonomies [110], where we have taken conceptual to empirical approach informed by the SLR to capture the concepts of software architecture evaluation under uncertainty. The process was iterative. We then applied *subjective* and *objective* evaluation to validate our classification framework. Subjective evaluation of the process of building the classification framework was inspired by [110]. In particular, our team members had several interactive sessions (~4 meetings) first to discuss the initial build-up of the classification framework. Subsequent iterations and refinements were informed by three working and feedback sessions

with team members (each taking an average of 2.5 hrs, one senior member with more than 30 years of experience in academic and industrial software architecture research and considered to be one of the founders of the field of architecture evaluation, a second senior member with more than 20 years experience in software architecture research and practice, and another two with 5-6 years experience in software architecture and computational intelligence in software engineering, covering uncertainties). Our team also consulted two external collaborators with expertise in the area of the software architecture for additional feedback. The following criteria, inspired by [110], informed our refinements and iterations: checks for the extent to which the classification framework is *concise* (i.e. with limited number of dimensions and limited number of characteristics for each dimension), robustness (i.e. with sufficient dimensions and characteristics to determine software architecture evaluation approaches under uncertainty), comprehensive (i.e. to categorize all known dimensions of architecture evaluation approaches under uncertainty within the software architecture domain), extensible (i.e. to allow the inclusion of additional dimensions and new characteristics within a dimension when new types of architecture evaluation approaches under uncertainty appear) and explanatory (i.e. by providing useful discussion of the architecture evaluation approaches under uncertainty to facilitate the understanding of how to evaluate software architectures under uncertainty). As for the objective evaluation inspired by [110], we ensured that every category (e.g. Quality Evaluation, Quality Attributes Considerations, Level of Autonomy, and Uncertainty Management) is unique and not repeated. All characteristics of architecture evaluation under uncertainty have been examined and no new characteristics are needed for addition.

– Our review focuses on architecture evaluation in the presence of uncertainty. In particular, the focus of the survey is on how existing architecture evaluation methods and commonly used approaches can provide ways for mitigating for uncertainties. For example, architecture evaluation can take several forms: the methods can be bespoke, providing phases and systematic guidance for architects to evaluate for the extent to which the architecture can meet its non-functional goals and trade-offs - e.g. ATAM, CBAM, *etc.* These methods can provide support for mitigating uncertainties. As an example, the use of exploratory and stress scenarios in ATAM is a way to anticipate likely or extreme cases and to design the architecture in a way that it can withstand these changes. Additionally, architecture evaluation can also focus on one concern (e.g., performance, security), where the analysis can utilise low level design models (e.g. state charts) and model-based analysis to analyse the system for specific qualities. Though these approaches are often regarded to be design-level evaluation with restricted focus on specific qualities (e.g. performance, security, reliability, liveliness, *etc*), the feedback gathered from their low level design analysis can help the architects to refine the software architecture under evaluation (e.g. ATMIS [64], performance modelling approaches [76–78, 135], *etc*). Analysis using model-based approaches can help the architects to reach more robust architectures against qualities of interest (e.g. security or performance) through continuous refinements that can better cater for uncertainties. For example, the architect can use performance models [76–78, 135] to inform refinements of the architecture that can better cope with uncertainties. Model-based analysis are design-level analysis. This analysis is often focused on the analysis on one or more sets of qualities using model-based modelling, analysis and tooling. Though this analysis operates on lower level of abstraction of that the architecture, the feedback of their analysis can help software architects and designers to evaluate software architectures for uncertainties and to suggest refinements that can better mitigate for uncertainties. These methods were not specifically discussed as either (i) methods for architecture evaluation, nor (ii) methods for evaluating and mitigating for uncertainties.

Nevertheless, we acknowledge their complementary role, if the architect would wish to utilise their use. Henceforth, model-based analysis is not the core objective of our survey due to their wide use of versatile and context-dependent use.

- Since the self-adaptive and self-managed domain is large, we did our best to include studies which show architecture evaluation as part of their approach. In particular, we added studies from a list of 5974 papers (the output of the search process in Figure 1) through search databases and a snowballing process, in addition to some manual search. However we may have missed some works unintentionally.
- Furthermore, a common threat to validity is the fact that there are some criteria—such as dealing with uncertainty and management of trade-offs—where the paper's authors do not explicitly mention whether they are addressed or not. In this context, we attempted to infer these criteria. Similarly, a common concern in the run-time approaches is that, in most cases, "the proactiveness or reactiveness of the approaches are not explicitly discussed and it can only be inferred from the adaptation strategies" [98]. Accordingly, we made our best effort to infer the reactiveness and proactiveness of the examined approaches.
- Other approaches, such as self-healing works, were excluded. For example, self-healing refers to the process of automatic recovery from failure. However, our SLR is concerned with the extent to which the architecture design decisions, tactics, and architecture choices tend to meet the quality requirements of the systems and their trade offs. As for uncertainty, it refers to the evaluation of these decisions in situations where it is difficult to predict the performance of these qualities due to dynamism in the system's operations and/or adequate understanding of the application domain. Though self-healing is not among the objectives of the paper, it can represent a specific scenario for the evaluation, where the architects can evaluate the extent to which the architecture design decisions can realise self-recovery for faults under uncertainty.
- Some continuous approaches were excluded from the list of studies. As an example, for [17], the focus has been primarily on development, whereas [146] focused on continuous testing and their relevance to the inclusion criteria is weak. Nevertheless, these types of approaches have motivated us to review and introduce continuous software architecture evaluation to the software architecture community.

5 RELATED REVIEWS

In the area of design-time architecture evaluation, there are many studies, such as [11, 27, 53, 122]. For instance, Dobrica et al. [53] focused on surveying the most popular methods, such as ATAM [85], CBAM [82], and ALMA [19]. Babar et al. [11] provided a framework for classifying design-time software architecture evaluation methods and a comparative analysis for the scenario-based approaches in specific in [10]. Roy et al. [122] extended the previous reviews and considered most of the design-time evaluation methods at that time. The authors in [27] systematically reviewed and classified architecture evaluation methods from the architecture evolution perspective.

Other surveys focused on run-time methods, such as self-adaptive systems [47, 93, 98], self-managed systems [26], and models@run-time [131]. From [26, 47, 93, 98, 131], we found that none of the studies explicitly demonstrated the use of run-time architecture evaluation principles. And none of the works have examined continuous software architecture evaluation. This is surprising because some research studies implicitly provide the elements for a continuous evaluation approach. Our survey bridges this gap by rethinking architecture evaluation and providing classifications that can do the following: (i) help architects to conduct the evaluation in continuous settings by determining the elements of a continuous evaluation approach; (ii) help in identifying common approaches for this type of evaluation; (iii) identify common concerns for systems that can benefit from this type of evaluation; (iv) point out the strengths and weaknesses of these types of approaches.

6 DISCUSSION AND RECOMMENDATIONS FOR FUTURE RESEARCH

Based on the SLR, it is clear that the area of software architecture evaluation has received substantial attention in recent years. Nevertheless, the results demonstrate some observations which could lead to future research. In particular, this SLR has identified several gaps in relation to architecture evaluation for uncertainty with respect to decisions which relate to designing dynamic and complex systems, such as IoT, cloud, and volunteer computing. In this context, this section aims to address the third question: RQ3: What are the current trends and future directions in software architecture evaluation for uncertainty and their consideration for continuous evaluation? *This question aims to show how we can benefit from the existing approaches to draw inspiration from the requirements and address the pitfalls when developing a continuous evaluation approach.* In Section 6.1, we present the architecture evaluation research area maturation stages and classification. We then highlight the important objectives that should be accomplished by the research community to advance this research area (Section 6.2 and 6.3). This is inline with the summary and reflection sections shown in Section 4.

6.1 Research Area Maturation

In this systematic review, we aim to investigate the extent to which architecture evaluation for uncertainty and the consideration for continuous evaluation have matured as a discipline. For this purpose, we examine the included studies with respect to the Redwine-Riddle model [119]. The latter provides six stages for technology (research area) maturity. These stages are [119]:

1. *Basic Research:* investigating the ideas and concepts; and providing a clear articulation of problem's scope.
2. *Concept Formulation:* presenting a comprehensive evaluation of solution approach through seminal paper or a demonstration system.
3. *Development and Extension:* preliminary using the ideas and extending the general approach to a broader solution.
4. *Internal Enhancement and Exploration:* extending the general approach to solve real problems in other research areas.
5. *External Enhancement and Exploration:* creating a broader group and involving them in decision-making to provide a substantial evidence of value and applicability.
6. *Popularization:* showing production-quality, providing supported versions, as well as marketing and commercializing the technology.

Initially, one author has classified the 48 included studies against Redwine-Riddle model, and the outcome was revised independently by other authors. Discussions and agreements were carried out in cases of discrepancies between the authors' categorizations. Figure 5 shows the results of classification. It is clear that almost 80% of the studies are still in early maturity stages (Basic Research and Concept Formulation), whereas almost 20% have been extended to broader problem domains and applied in practice. Among those approaches that are already adopted by industry, none of them are deployed at run-time; they only focus on design-time evaluation. In particular, maturity has only been proven for design-time approaches, such as ATAM and CBAM. This explains why 4% (2) of approaches are still in the popularization stage. In Appendix B, we tabulate the studies with respect to domain maturity level.

We have seen some examples of continuous evaluation that are either implicit, partial, or explicit, such as CPASA and DevOps. However, these research efforts have not demonstrated and

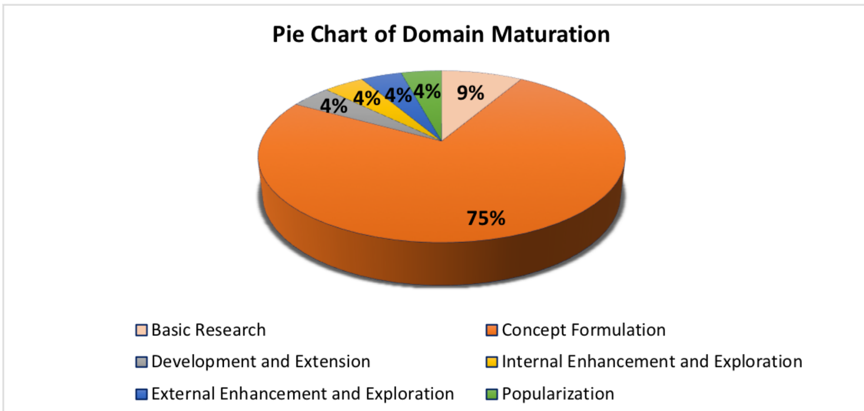

Fig. 5. Distribution of the included studies over the domain maturity classification model (The maturity distribution are shown in percentage).

documented how to adapt those practices in the evaluation of software architectures for uncertainty. Therefore, to mature the architecture evaluation research area with continuous evaluation approaches, we need a set of guidelines, tools, systematic procedures, acceptance from (and case studies with) real-world organizations, and shared benchmarks across companies for best practices.

6.2 Leveraging Existing Approaches To Develop A Continuous Evaluation Framework

Having done this SLR, we observe that elements from different approaches could be combined to develop a continuous software architecture evaluation framework. We briefly present our views on potential ones that seem worthwhile to be further explored below:

- **Architecture capabilities that can better cope and respond to uncertainty:** examples of these capabilities are architecture design diversification [51], tactics for meeting non-functional requirements, *etc*. Consider diversification as one of the capabilities that could enrich the architecture to cater for uncertainty and provide means for reliability and continuously meeting the behavioural requirements. Such capability require the software architecture community to leverage findings on design diversity in software engineering to develop fundamentals for software architecture diversity for uncertainties, covering styles, decisions, tactics, *etc*, which is inline of our earlier work - [126, 127], as well as rethinking architecture evaluation to consider dynamism and uncertainty. In this context, a systematic design-time evaluation approach that can deal with these capabilities and handle uncertainties is necessary. This is an important foundation of a continuous evaluation framework. Some initial works have discussed these potentials [126], but it still requires further investigations.
- **The use of economics-based approaches in architecture evaluation:** based on the existing approaches, we infer that there is a lack of well-documented, real-world examples for economics-based approaches in the context of design-time evaluations (Table 6 and 10:) In particular, these approaches ([12, 13, 114]) have not been used to deal with cost-benefit trade-offs in dynamic environments, such as IoT. Further, they have not been explored from the perspective of forecasting the long-term value of architecture decisions to determine whether the complex design decisions, such as diversity in design [51], can handle uncertainties that can be attributed to dynamic changes in the environment. As mentioned previously, the CBAM [82] is a scenario-based design-time evaluation method, which determines the

influence of architecture decisions on the cost-benefit trade-offs. The CBAM provides an implicit mitigation for uncertainty through different types of scenarios. However, this type of evaluation approach would not be suitable for the emerging technologies and paradigms, such as IoT and cloud-based systems. We believe that economics-based approaches, such as real options analysis [8] and modern portfolio theory [100], could be combined with CBAM to support the analysis. Real Options Analysis is one of the few design-time techniques that can embed flexibility under uncertainty. Therefore, it can aid the architect in predicting the impact of architecture decisions on quality attributes of interest. It can also shortlist the candidate options for deployment at run-time and thus reduce unnecessary costs. This is still very much a research area that requires further investigation in the context of design-time evaluation, as an initial stage for continuous evaluation.

- **New methods for continuous architecture evaluation that interleave and intertwine design and run-time architecture evaluation:** we found that most of the architecture evaluation approaches focus on design-time (about 60% of the approaches) and less on run-time (about 40% of the approaches). Evaluation approaches also tend to focus on development (i.e. mostly human-centric activities) and lack a consolidated approach that integrates design-time and run-time considerations. On the contrary, in the context of architecting and evaluating dynamic and complex systems, a more continuous approach that starts at the early stages of development and continues to evaluate the architecture options during the lifetime of the system at run-time is necessary to cope with operational uncertainties, such as high fluctuations in QoS, sensor ageing effects, *etc.*

6.3 Finding The Necessary Ingredients For Developing A Continuous Evaluation Framework

Modern software system environments, such as IoT, cloud, volunteer computing, and microservices, are a challenge for existing software architecture evaluation methods. Such systems are largely data-driven, characterised by their dynamism, unpredictability in operation, hyper-connectivity, and scalability. Properties, such as performance, delayed delivery, and scalability, are acknowledged to pose great risk and are difficult to evaluate at design-time only. Therefore, a run-time evaluation approach is necessary to complement design-time analysis. This run-time stage should be able to handle different sources of uncertainty and evaluate complex design-time decisions. In this regard, we need to determine the necessary ingredients for this run-time stage. We briefly present our views on potential directions for run-time stage that seem worthwhile to be further explored below:

- **Analysing the cost as a quality concern when developing a continuous evaluation approach:** one interesting observation is that just 25% of run-time approaches address cost as a concern (Table 11 and 6). Since the management of cost-benefit trade-offs is essential in dynamic environments [71], cost will highly influence the value of architecture decisions. When evaluating software architectures, there would be some conflicting QoS goals. Therefore, when designing a continuous evaluation approach, one could benefit from the literature with respect to multi-objective optimisation under uncertainty, such as the use of Pareto-Optimal in [44, 70], Genetic Algorithms in [138], Fuzzy Logic in [144], *etc.*
- **The need to incorporate change detection tests to the evaluation:** based on the results of our review (Table 11), most of the run-time approaches handle uncertainty either by checking goal violations or providing some probabilistic estimations. However, in contexts of highly dynamic environments such as [4, 37, 71, 108, 109], this is not sufficient. Even if the currently deployed architecture decision is not violating any goal, this does not mean that it has good performance. For example, in some cases, an architecture decision is meeting its

quality constraints but it is providing poor performance. In this context, a change detection test is a necessary component in a continuous evaluation framework to determine significant drifts in the architecture decisions. This type of test can provide the architect with the flexibility of adjusting the sensitivity to changes. Therefore, determining the type of test and its efficiency could be a potential future direction. In [127], one type of change detection test was used, however, we see potentials of exploring other change detection tests [52] to handle different forms of uncertainty.

- **The need for ageing parameters for data analysis:** most of the existing run-time methods rely on historical data or online data to perform the evaluation, but they do not consider the age of data. Therefore, embedding some ageing parameters to emphasise the relative importance of older versus more recent data could potentially improve the analysis [31]. Further investigations, related to the use of these parameters and how the architect could tune these parameters to enhance the evaluation are required.
- **The need for new proactive approaches for continuous architecture evaluation:** from the run-time perspective (Table 11 and 6), it is clear that most of the current approaches (e.g. [31, 61, 143], *etc*) tend to be reactive when simplistic learning, partial or incomplete knowledge is used. Thus they may suggest incorrect decisions due to unexpected future environment changes and recommend unnecessary switches due to the lack of future knowledge about the candidate architecture decisions. This in turn may affect the architecture's stability and overall behaviour. To bridge the gap, further proactive approaches are necessary to improve the continuous evaluation process.
- **Embedding machine learning and forecasting techniques to the continuous evaluation framework:** our analysis shows that just 25% of the run-time approaches embed machine learning principles in the decision-making process. Using machine learning approaches in decision-making has shown great improvements to the decision-making (e.g. [20]). Therefore, another important element when developing a continuous architecture evaluation framework is leveraging machine learning techniques. There are methods (e.g. [58, 117]) that explicitly mention continuous architecting and assessment, and others that implicitly adopt it (e.g. [17]). These approaches can benefit from further investigations in terms of how continuous evaluation could dynamically track and forecast architecture decisions and automatically manage cost-benefit trade-offs.
- **Consider scalability when designing a continuous evaluation framework:** the literature depicts that there are some approaches (e.g. [46, 58, 69, 106]) that are proactive in terms of failure prediction and recommending alternatives. These approaches may, however, experience scalability problems. Moreover, these approaches assume that the impact of architecture decisions on QoS is available at run-time, which is not always the case for uncertain environments such as IoT. To this end, novel solutions are required to determine how QoS monitoring challenges could be handled.

7 CONCLUSION

Continuous evaluation has been discussed under different labels, such as run-time, dynamic, continuous, *etc*, along with assessment and analysis. The common characteristic among these efforts is that they start at design-time (even if they do not mention that explicitly) and continue to evaluate architecture decisions during the life-time of system by observing environmental conditions. In this review we have attempted to unify these efforts. We performed a systematic literature review to examine existing architecture evaluation methods that deal with uncertainty either design-time or run-time. We also provided guidelines for the necessary elements to develop and conduct a continuous architecture evaluation approach. We both automatically and manually searched well-known venues for software architecture and engineering, other related systematic reviews and mapping studies, and significant bibliographical data sources. In addition we applied a snowballing process to collect our primary studies.

The results of our investigation are the following: (a) design-time architecture evaluation approaches garnered more attention than run-time ones, though the latter are increasingly important to handle the dynamism and increasing complexity in software systems; (b) there is a lack of examples on demonstrating how continuous evaluation approaches can realised and conducted; (c) few methods focus on managing trade-offs between benefits and costs at run-time; (d) few methods focus on adopting machine learning techniques to the evaluation; (e) most of the run-time approaches tend to be reactive (and may recommend unnecessary switches and hence increase deployment costs).

In summary, based on our main findings listed in Tables 10, 11, 6, and 7, 8, we suggest the following opportunities for future work in this area: (i) employ economics-based approaches (i.e. forecasting the long-term value of complex architecture decisions); (ii) adopt economics-based principles in the design-time evaluation approach (the initial stage of a continuous evaluation approach) because it embeds flexibility under uncertainty; (iii) perform additional research in analysing the use of machine learning techniques to improve architecture evaluation at run-time (the ongoing stage in a continuous evaluation approach); (iv) investigate the development of proactivity in the architecture evaluation process; (v) explore how tuning the input parameters for the continuous evaluation (e.g. sensitivity to changes, monitoring intervals, the relative importance of present/past data) could affect the evaluation and what are the most suitable parameters to improve the decision-making; (vi) analyze the use of continuous architecture evaluation in dynamic environments, such as IoT and cloud systems.

A THE LIST OF INCLUDED STUDIES WITH RESPECT TO CLASSIFICATION FRAMEWORK

In this appendix, we tabulate the list of included studies with respect to classification framework in Table 10-12.

B THE INCLUDED STUDIES AND THEIR MATURITY LEVEL

In this appendix, we first tabulate the studies with respect to domain maturity level in Table 13 and then provide a list of included studies in the systematic literature review in Table 14, 15, and 16.

REFERENCES

[1] 2017. (July 2017). https://techbeacon.com/5-challenges-performance-engineering-iot-apps?amp

[2] Amritanshu Agrawal, Tim Menzies, Leandro L Minku, Markus Wagner, and Zhe Yu. 2020. Better software analytics via" DUO": Data mining algorithms using/used-by optimizers. *Empirical Software Engineering* 25, 3 (2020), 2099–2136.

[3] Sarah Al-Azzani and Rami Bahsoon. 2010. Using implied scenarios in security testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*. ACM, 15–21.

[4] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2018. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing* 11, 2 (2018), 430–447.

[5] Tariq Al-Naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, and Boualem Benatallah. 2005. A quality-driven systematic approach for architecting distributed software applications. In *Proceedings of the 27th international conference on Software engineering*. ACM, 244–253.

[6] Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. 2009. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*. IEEE, 61–71.

[7] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. 2013. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering* 39, 5 (2013), 658–683.

Table 10. Representative Contributions for Design-time Architecture Evaluation.

Study	Approaches to Evaluation	Addressing QA	Supported QA	Management of stakeholder input	Management of trade-offs	Treatment of uncertainty	Source of uncertainty
[83]	Scenario-based	Multiple	Specific (Modifiability, Portability, Extensibility)	Human-Reliant	No Support	Implicit	Epistemic
[43]	Utility-based Scenario-based	Multiple	General	Human-Reliant	Manual	Implicit	Epistemic
[82]	Utility-based Scenario-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Implicit	Epistemic
[18]	Utility-based Scenario-based Parametric-based	Multiple	Specific (Performance, Fault-tolerance, Maintainability, Reusability)	Human-Reliant	No Support	Implicit	Epistemic
[19]	Utility-based Scenario-based	Single	Specific (Modifiability)	Human-Reliant	No Support	Implicit	Epistemic
[79]	Utility-based Scenario-based	Multiple	General + Specific (Cost)	Human-Reliant	No Support	Implicit	Epistemic + Aleatory
[84]	Utility-based Scenario-based Parametric-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Implicit	Epistemic
[139]	Utility-based Scenario-based	Single	Specific (Performance+Cost)	Human-Reliant	Manual	Implicit	Epistemic
[145]	Utility-based Scenario-based Parametric-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Explicit	Epistemic
[5]	Utility-based Parametric-based Search-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Implicit	Epistemic
[70]	Utility-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Implicit	Epistemic
[86]	Utility-based Parametric-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Explicit	Epistemic
[96]	Utility-based	Multiple	Specific (Dependability, Reliability and Maintainability)	Human-Reliant	No Support	Implicit	Aleatory
[94]	Utility-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Explicit	Epistemic
[42]	Utility-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Explicit	Epistemic
[64]	Utility-based	Multiple	General +	Human-Reliant	Manual	Explicit	Epistemic + Aleatory
[113]	Utility-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Explicit	Epistemic
[57]	Utility-based	Multiple	General	Human-Reliant	No Support	Implicit	Epistemic
[63]	Utility-based Scenario-based Search-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Explicit	Epistemic
[95]	Utility-based Scenario-based Search-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Automatic	Explicit	Epistemic
[12]	Economics-based	Multiple	Specific (Stability) + Specific (Cost)	Semi-Autonomous	Manual	Explicit	Epistemic
[13]	Economics-based	Multiple	Specific (Scalability) + Specific (Cost)	Semi-Autonomous	Manual	Explicit	Epistemic
[114]	Economics-based Scenario-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Manual	Explicit	Epistemic
[112]	Economics-based Scenario-based	Multiple	General + Specific (Cost)	Semi-Autonomous	Manual	Explicit	Epistemic
[104]	Utility-based Parametric-based	Single	Specific (Reliability)	Semi-Autonomous	No Support	Explicit	Epistemic + Aleatory
[103]	Utility-based Parametric-based Search-based	Multiple	Specific (Reliability)	Semi-Autonomous	Automatic	Explicit	Aleatory
[102]	Utility-based Parametric-based Search-based	Multiple	Specific (Reliability + Performance)	Semi-Autonomous	Automatic	Explicit	Epistemic
[28]	Utility-based Search-based	Multiple	Generic	Semi-Autonomous	Automatic	Explicit	Epistemic
[105]	Utility-based Parametric-based	Multiple	Generic	Semi-Autonomous	No Support	Explicit	Epistemic + Aleatory

Table 11. Representative Contributions for Run-time Architecture Evaluation.

Study	Approaches to Evaluation	Addressing QA	Supported QA	Management of stakeholder input	Management of trade-offs	Treatment of uncertainty	Source of uncertainty	Monitoring & Treatment of QAs
[40]	Utility-based Parametric-based	Multiple	General + Specific (Cost)	Autonomous	Automatic	Explicit	Epistemic + Aleatory	Reactive
[133]	Utility-based Learning-based	Single	Specific (Performance)	Autonomous	No Support	Explicit	Aleatory	Reactive
[32]	Utility-based	Multiple	Specific (Performance, Energy Consumption)	Autonomous	Automatic	Implicit	Aleatory	Reactive
[87]	Utility-based Learning-based	Multiple	General	Autonomous	Automatic	Explicit	Aleatory	Reactive
[75]	Utility-based	Multiple	Specific (Reliability and Performance)	Semi-Autonomous	No Support	Implicit	Aleatory	Reactive
[141]	Utility-based Scenario-based	Multiple	General + Specific (Cost)	Semi-Autonomous	No Support	Implicit	Aleatory	Reactive
[31]	Utility-based	Multiple	Specific (Reliability and Performance)	Autonomous	No Support	Explicit	Aleatory	Reactive
[30]	Utility-based	Multiple	Specific (Reliability and Performance)	Semi-Autonomous	Automatic	Explicit	Epistemic + Aleatory	Reactive
[62]	Utility-based Search-based	Multiple	General	Semi-Autonomous	Automatic	Explicit	Epistemic + Aleatory	Reactive
[61]	Utility-based Learning-based	Multiple	General + Specific (Cost)	Autonomous	Automatic	Explicit + Aleatory	Epistemic	Reactive
[67]	Utility-based Search-based	Single	Specific (Energy Consumption)	Semi-Autonomous	No Support	Explicit	Aleatory	Reactive
[58]	Utility-based Parametric-based	Single	Specific (Reliability)	Semi-Autonomous	No Support	Implicit	Epistemic + Aleatory	Proactive
[69]	Utility-based	Multiple	General	Semi-Autonomous	No Support	Explicit	Aleatory	Proactive
[46]	Utility-based	Multiple	Specific (Reliability and Efficiency)	Autonomous	Automatic	Implicit	Aleatory	Proactive
[106]	Utility-based Learning-based	Multiple	Specific (Performance)	Autonomous	No Support	Explicit	Aleatory	Proactive

Table 12. Representative Contributions for Continuous Architecture Evaluation.

Study	Approaches to Evaluation	Addressing QA	Supported QA	Management of stakeholder input	Management of trade-offs	Treatment of uncertainty	Source of uncertainty	Monitoring & Treatment of QAs
[117]	Utility-based	Multiple	General + Specific (Cost)	Human-Reliant	Manual	Implicit	Epistemic	No treatment
[69]	Utility-based Parametric-based Search-based	Multiple	General	Semi-Autonomous	No Support	Explicit	Aleatory	Proactive
[136]	Utility-based Economics-based	Multiple	General + Specific (Cost)	Autonomous	Automatic	Explicit	Epistemic + Aleatory	Reactive
[127]	Utility-based Search-based Scenario-based Learning-based Economics-based	Multiple	General + Specific (Cost)	Autonomous	Automatic	Explicit	Epistemic + Aleatory	Reactive

Table 13. Studies with respect to Domain maturation level.

Domain Maturation Level	Studies	# of Studies
Basic Research	[28, 83, 113, 117]	4
Concept Formulation	[5, 12, 40, 70, 79, 84, 86, 104, 105, 114, 133, 139, 145] [13, 30–32, 46, 58, 61, 62, 75, 87, 94, 96, 141] [48, 67, 69, 95, 102, 103, 106, 112, 136]	35
Development and Extension	[18, 57]	2
Internal Enhancement	[63, 64]	2
External Enhancement	[19, 42]	2
Popularization	[43, 82]	2
Total		48

Table 14. Studies included in the review.

Study	Ref	Author(s)	Year	Title
S1	[83]	R. Kazman, L. Bass, G. Abowd, & M. Webb	1994	SAAM: A method for analyzing the properties of software architectures
S2	[18]	P. Bengtsson & J. Bosch	1998	Scenario-based software architecture reengineering
S3	[82]	R. Kazman, J. Asundi, & P. Clements	2001	Quantifying the costs and benefits of architectural decisions
S4	[139]	L. Williams & C. Smith	2002	PASASM: A Method for the Performance Assessment of Software Architectures
S5	[43]	R. Kazman, M. Klein, P. Clements & others	2003	Evaluating software architectures
S6	[19]	P. Bengtsson, N. Lassing, J. Bosch, & H. Vliet	2004	Architecture-level modifiability analysis (ALMA)
S7	[12]	R. Bahsoon & W. Emmerich	2004	Evaluating architectural stability with real options theory
S8	[40]	S. Cheng	2004	Rainbow: cost-effective software architecture-based self-adaptation
S9	[79]	M. Ionita, P. America, D. Hammer, H. Obbink & J. Trienekens	2004	A Scenario-Driven Approach for Value, Risk, and Cost Analysis in System Architecting for Innovation
S10	[145]	L. Zhu, A. Aurum, I. Gorton, & R. Jeffery	2005	Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process
S11	[5]	T. Al-Naeem, I. Gorton, M. Babar, F. Rabhi & B. Benatallah	2005	A quality-driven systematic approach for architecting distributed software applications
S12	[84]	R. Kazman, L. Bass & M. Klein	2006	The essential components of software architecture design and analysis
S13	[70]	L. Grunske	2006	Identifying good architectural design alternatives with multi-objective optimization strategies
S14	[133]	G. Tesauro	2007	Reinforcement learning in autonomic computing: A manifesto and case studies
S15	[114]	I. Ozkaya, R. Kazman & M. Klein	2007	Quality-attribute based economic valuation of architectural patterns
S16	[86]	C. Kim, D. Lee, I. Ko & J. Baik	2007	A Lightweight Value-based Software Architecture Evaluation
S17	[13]	R. Bahsoon & W. Emmerich	2008	An economics-driven approach for valuing scalability in distributed architectures
S18	[96]	Y. Liu, M. Babar & I. Gorton	2008	Middleware Architecture Evaluation for Dependable Self-managing Systems

[8] Martha Amram, Nalin Kulatilaka, et al. 1998. Real Options:: Managing Strategic Investment in an Uncertain World. *OUP Catalogue* (1998).
[9] Jayatirtha Asundi, Rick Kazman, and Mark Klein. 2001. *Using economic considerations to choose among architecture design alternatives.* Technical Report. DTIC Document.

Table 15. Studies included in the review (Continued).

Study#	Ref	Author(s)	Year	Title
S19	[75]	W. Heaven, D. Sykes, J. Magee & J. Kramer	2009	A case study in goal-driven architectural adaptation
S20	[87]	D. Kim & S. Park	2009	Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software
S21	[141]	J. Yang, G. Huang, W. Zhu, X. Cui & H. Mei	2009	Quality attribute tradeoff through adaptive architectures at runtime
S22	[94]	J. Lee, S. Kang & C. Kim	2009	Software architecture evaluation methods based on cost benefit analysis and quantitative decision making
S23	[58]	I. Epifani, C. Ghezzi, R. Mirandola & G. Tamburrelli	2009	Model evolution by run-time parameter adaptation
S24	[32]	R. Calinescu & M. Kwiatkowska	2009	Using quantitative analysis to implement autonomic IT systems
S25	[42]	He. Christensen, K. Hansen & B. LindstrÃÿm	2011	Lightweight and continuous architectural software quality assurance using the asqa technique
S26	[117]	R. Pooley & A. Abdullatif	2010	Cpasa: continuous performance assessment of software architecture
S27	[64]	F. Faniyi, R. Bahsoon, A. Evans & R. Kazman	2011	Evaluating security properties of architectures in unpredictable environments: A case for cloud
S28	[62]	N. Esfahani, E. Kouroshfar & S. Malek	2011	Taming uncertainty in self-adaptive software
S29	[31]	R. Calinescu, K. Johnson & Y. Rafiq	2011	Using observation ageing to improve Markovian model learning in QoS engineering
S30	[30]	R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola & G. Tamburrelli	2011	Dynamic QoS management and optimization in service-based systems
S31	[104]	I. Meedeniya, I. Moser, A. Aleti, & L. Grunske	2011	Architecture-based reliability evaluation under uncertainty
S32	[103]	I. Meedeniya, I. Moser, A. Aleti, & L. Grunske	2011	Architecture-driven reliability optimization with uncertain model parameters

[10] Muhammad Ali Babar and Ian Gorton. 2004. Comparison of scenario-based software architecture evaluation methods. In *Software Engineering Conference, 2004. 11th Asia-Pacific.* IEEE, 600–607.
[11] Muhammad Ali Babar, Liming Zhu, and Ross Jeffery. 2004. A framework for classifying and comparing software architecture evaluation methods. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian.* IEEE, 309–318.
[12] Rami Bahsoon and Wolfgang Emmerich. 2004. Evaluating architectural stability with real options theory. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on.* IEEE, 443–447.
[13] Rami Bahsoon and Wolfgang Emmerich. 2008. An economics-driven approach for valuing scalability in distributed architectures. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on.* IEEE, 9–18.

Table 16. Studies included in the review (Continued).

Study#	Ref	Author(s)	Year	Title
S33	[102]	I. Meedeniya, A. Aleti, I. Avazpour & A. Amin	2012	Robust ArcheOpterix: Architecture Optimization of Embedded Systems Embedded Systems under uncertainty
S34	[113]	M. Osterlind, P. Johnson, K. Karnati, R. Lagerstrom & M. Valja	2013	Enterprise architecture evaluation using utility theory
S35	[63]	N. Esfahani, S. Malek & K. Razavi	2013	GuideArch: guiding the exploration of architectural solution space under uncertainty
S36	[46]	D. Cooray, E. Kouroshfar, S. Malek & R. Roshandel	2013	Proactive self-adaptation for improving embedded, the reliability of mission-critical, and mobile software
S37	[61]	N. Esfahani, A. Elkhodary & S. Malek	2013	A learning-based framework for engineering feature-oriented self-adaptive software systems
S38	[69]	C. Ghezzi & A. Sharifloo	2013	Dealing with non-functional requirements for adaptive systems via dynamic software product-lines
S39	[67]	E. Fredericks, B. DeVries & B. Cheng	2014	Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty
S40	[95]	E. Letier, D. Stefan & E. Barr	2014	Uncertainty, risk, and information value in software requirements and architecture
S41	[105]	I. Meedeniya, A. Aleti, & L. Grunske	2014	Evaluating probabilistic models with uncertain model parameters
S42	[57]	V. Eloranta, U. Heesch, P. Avgeriou, N. Harrison & K. Koskimies	2015	Lightweight Evaluation of Software Architecture Decisions
S43	[112]	B. Ojameruaye, R. Bahsoon & L. Duboc	2016	Sustainability debt: a portfolio-based approach for evaluating sustainability requirements in architectures
S44	[106]	G. Moreno, J. Camara, D. Garlan & B. Schmerl	2016	Efficient decision-making under uncertainty for proactive self-adaptation
S45	[136]	V. Donckt, M. Jeroen, D. Weyns, M. Iftikhar & R. Singh	2018	Cost-Benefit Analysis at Runtime for Self-Adaptive Systems Applied to an Internet of Things Application
S46	[48]	M. De Sanctis, R. Spalazzese, & C. Trubiani	2019	Qos-based formation of software architectures in the internet of things
S47	[28]	A. Busch, D. Fuchß, & A. Koziolek	2019	Peropteryx: Automated improvement of software architectures
S48	[127]	D. Sobhy, L. Minku, R. Bahsoon T. Chen & R. Kazman	2020	Run-time evaluation of architectures: A case study of diversification in IoT

[14] Carliss Young Baldwin and Kim B Clark. 2000. *Design rules: The power of modularity*. Vol. 1. MIT press.
[15] Linden J Ball, Balder Onarheim, and Bo T Christensen. 2010. Design requirements, epistemic uncertainty and solution development strategies in software design. *Design Studies* 31, 6 (2010), 567–589.
[16] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software architecture in practice*. Addison-Wesley Professional.
[17] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.
[18] PerOlof Bengtsson and Jan Bosch. 1998. Scenario-based software architecture reengineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE, 308–317.
[19] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. 2004. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software* 69, 1 (2004), 129–147.
[20] Amel Bennaceur, Valérie Issarny, Daniel Sykes, Falk Howar, Malte Isberner, Bernhard Steffen, Richard Johansson, and Alessandro Moschitti. 2012. Machine learning for emergent middleware. In *International Workshop on Eternal Systems*. Springer, 16–29.
[21] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, et al. 2005. Software quality attributes and trade-offs. *Blekinge Institute of Technology* (2005).
[22] Gordon Blair, Nelly Bencomo, and Robert B France. 2009. Models@ run. time. *Computer* 42, 10 (2009).
[23] Barry W Boehm et al. 1981. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ).
[24] Barry W Boehm and Kevin J Sullivan. 2000. Software economics: a roadmap. In *Proceedings of the conference on The future of Software engineering*. ACM, 319–343.
[25] Jan Bosch. 2004. Software architecture: The next step. In *European Workshop on Software Architecture*. Springer, 194–199.
[26] Jeremy S Bradbury, James R Cordy, Juergen Dingel, and Michel Wermelinger. 2004. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM, 28–33.
[27] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. 2012. A systematic review of software architecture evolution research. *Information and Software Technology* 54, 1 (2012), 16–40.
[28] Axel Busch, Dominik Fuchß, and Anne Koziolek. 2019. Peropteryx: Automated improvement of software architectures. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 162–165.
[29] Radu Calinescu. 2013. Emerging techniques for the engineering of self-adaptive high-integrity software. In *Assurances for Self-Adaptive Systems*. Springer, 297–310.
[30] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. 2011. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering* 37, 3 (2011), 387–409.
[31] Radu Calinescu, Kenneth Johnson, and Yasmin Rafiq. 2011. Using observation ageing to improve Markovian model learning in QoS engineering. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*. ACM, 505–510.
[32] Radu Calinescu and Marta Kwiatkowska. 2009. Using quantitative analysis to implement autonomic IT systems. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 100–110.
[33] Javier Cámara, Gabriel Moreno, and David Garlan. 2015. Reasoning about human participation in self-adaptive systems. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 146–156.
[34] Stephanie Riegg Cellini and James Edwin Kee. 2015. Cost-effectiveness and cost-benefit analysis. *Handbook of practical program evaluation* 4 (2015).
[35] Humberto Cervantes and Rick Kazman. 2016. *Designing software architectures: a practical approach*. Addison-Wesley Professional.
[36] Franck Chauvel, Nicolas Ferry, Brice Morin, Alessandro Rossini, and Arnor Solberg. 2013. Models@ Runtime to Support the Iterative and Continuous Design of Autonomic Reasoners.. In *MoDELS@ Run. time*. 26–38.
[37] Lianping Chen. 2018. Microservices: architecting for continuous delivery and DevOps. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 39–397.
[38] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. 2018. FEMOSAA: Feature-Guided and Knee-Driven Multi-Objective Optimization for Self-Adaptive Software. *ACM Trans. Softw. Eng. Methodol.* 27, 2, Article 5 (June 2018), 50 pages. https://doi.org/10.1145/3204459
[39] Betty HC Cheng, Kerstin I Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A Müller, Patrizio Pelliccione, Anna Perini, Nauman A Qureshi, Bernhard Rumpe, et al. 2014. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*. Springer, 101–136.
[40] Shang-Wen Cheng. 2008. *Rainbow: cost-effective software architecture-based self-adaptation*. ProQuest.

[41] Shang-Wen Cheng. 2008. *Rainbow: Cost-effective Software Architecture-based Self-adaptation*. Ph.D. Dissertation. Pittsburgh, PA, USA. Advisor(s) Garlan, David. AAI3305807.

[42] Henrik Bærbak Christensen, Klaus Marius Hansen, and Bo Lindstrøm. 2010. Lightweight and continuous architectural software quality assurance using the asqa technique. In *European Conference on Software Architecture*. Springer, 118–132.

[43] Paul Clements, Rick Kazman, Mark Klein, et al. 2003. *Evaluating software architectures*. Tsinghua University Press Beijing.

[44] David W Coit, Tongdan Jin, and Naruemon Wattanapongsakorn. 2004. System optimization with component reliability estimation uncertainty: a multi-criteria approach. *IEEE transactions on reliability* 53, 3 (2004), 369–380.

[45] Software Engineering Standards Committee et al. 1998. *IEEE Standard for a software quality metrics methodology, Std. 1061-1998*. Technical Report. Technical Report.

[46] Deshan Cooray, Ehsan Kouroshfar, Sam Malek, and Roshanak Roshandel. 2013. Proactive self-adaptation for improving the reliability of mission-critical, embedded, and mobile software. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1714–1735.

[47] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.

[48] Martina De Sanctis, Romina Spalazzese, and Catia Trubiani. 2019. Qos-based formation of software architectures in the internet of things. In *European Conference on Software Architecture*. Springer, 178–194.

[49] Mark Denne and Jane Cleland-Huang. 2004. The incremental funding method: Data-driven software development. *IEEE software* 21, 3 (2004), 39–47.

[50] Armen Der Kiureghian and Ove Ditlevsen. 2009. Aleatory or epistemic? Does it matter? *Structural Safety* 31, 2 (2009), 105–112.

[51] Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. 1998. Diversity against accidental and deliberate faults. In *csda*. IEEE, 171.

[52] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10, 4 (2015), 12–25.

[53] Liliana Dobrica and Eila Niemela. 2002. A survey on software architecture analysis methods. *IEEE Transactions on software Engineering* 28, 7 (2002), 638–653.

[54] Donia El Kateb, François Fouquet, Grégory Nain, Jorge Augusto Meira, Michel Ackerman, and Yves Le Traon. 2014. Generic cloud platform multi-objective optimization leveraging models@ run.time. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 343–350.

[55] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. FUSION: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 7–16.

[56] Salah E Elmaghraby, Willy S Herroelen, et al. 1990. The scheduling of activities to maximize the net present value of projects. *European Journal of Operational Research* 49, 1 (1990), 35–49.

[57] Veli-Pekka Eloranta, Uwe van Heesch, Paris Avgeriou, Neil Harrison, and Kai Koskimies. 2015. Lightweight Evaluation of Software Architecture Decisions. In *Relating System Quality and Software Architecture*. Elsevier, 157–179.

[58] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. 2009. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 111–121.

[59] Hakan Erdogmus. 2002. A Real Options Perspective of Software Reuse. In *International Workshop on Reuse Economics âĂIJRedirecting Reuse EconomicsâĂİ Tuesday*.

[60] Hakan Erdogmus and Jennifer Vandergraaf. 1999. Quantitative approaches for assessing the value of COTS-centric development. (1999).

[61] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. 2013. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE transactions on software engineering* 39, 11 (2013), 1467–1493.

[62] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. 2011. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 234–244.

[63] Naeem Esfahani, Sam Malek, and Kaveh Razavi. 2013. GuideArch: guiding the exploration of architectural solution space under uncertainty. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 43–52.

[64] Funmilade Faniyi, Rami Bahsoon, Andy Evans, and Rick Kazman. 2011. Evaluating security properties of architectures in unpredictable environments: A case for cloud. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. IEEE, 127–136.

[65] John Favaro. 1996. A comparison of approaches to reuse investment analysis. In *Software Reuse, 1996., Proceedings Fourth International Conference on*. IEEE, 136–145.
[66] Craig R Fox and Gülden Ülkümen. 2011. Distinguishing two dimensions of uncertainty. *Perspectives on thinking, judging, and decision making* (2011), 21–35.
[67] Erik M Fredericks, Byron DeVries, and Betty HC Cheng. 2014. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 17–26.
[68] David Garlan and Bradley Schmerl. 2004. Using architectural models at runtime: Research challenges. In *European Workshop on Software Architecture*. Springer, 200–205.
[69] Carlo Ghezzi and Amir Molzam Sharifloo. 2013. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In *Software Engineering for Self-Adaptive Systems II*. Springer, 191–213.
[70] Lars Grunske. 2006. Identifying good architectural design alternatives with multi-objective optimization strategies. In *Proceedings of the 28th international conference on Software engineering*. ACM, 849–852.
[71] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660.
[72] Marcus Handte, Gregor Schiele, Verena Matjuntke, Christian Becker, and Pedro José Marrón. 2012. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7, 1 (2012), 10.
[73] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.
[74] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.
[75] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. 2009. A case study in goal-driven architectural adaptation. In *Software Engineering for Self-Adaptive Systems*. Springer, 109–127.
[76] Robert Heinrich. 2016. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *ACM SIGMETRICS Performance Evaluation Review* 43, 4 (2016), 13–22.
[77] Robert Heinrich, Christian Zirkelbach, and Reiner Jung. 2017. Architectural Runtime Modeling and Visualization for Quality-Aware DevOps in Cloud Applications. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 199–201.
[78] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. 2016. Model-based self-aware performance and resource management using the descartes modeling language. *IEEE Transactions on Software Engineering* 43, 5 (2016), 432–452.
[79] Mugurel T Ionita, Pierre America, Dieter K Hammer, Henk Obbink, and Jos JM Trienekens. 2004. A scenario-driven approach for value, risk, and cost analysis in system architecting for innovation. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*. IEEE, 277–280.
[80] Anton Jansen and Jan Bosch. 2005. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. IEEE, 109–120.
[81] Lawrence G Jones and Anthony J Lattanze. 2001. *Using the architecture tradeoff analysis method to evaluate a wargame simulation system: A case study*. Technical Report. DTIC Document.
[82] Rick Kazman, Jai Asundi, and Mark Klein. 2001. Quantifying the costs and benefits of architectural decisions. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 297–306.
[83] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. 1994. SAAM: A method for analyzing the properties of software architectures. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*. IEEE, 81–90.
[84] Rick Kazman, Len Bass, and Mark Klein. 2006. The essential components of software architecture design and analysis. *Journal of Systems and Software* 79, 8 (2006), 1207–1216.
[85] Rick Kazman, Mark Klein, and Paul Clements. 2000. *ATAM: Method for architecture evaluation*. Technical Report. DTIC Document.
[86] Chang-Ki Kim, Dan-Hyung Lee, In-Young Ko, and Jongmoon Baik. 2007. A lightweight value-based software architecture evaluation. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, Vol. 2. IEEE, 646–649.
[87] Dongsun Kim and Sooyong Park. 2009. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 76–85.
[88] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[89] Barbara Kitchenham, Rialette Pretorius, David Budgen, O Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen Linkman. 2010. Systematic literature reviews in software engineering–a tertiary study. *Information and Software Technology* 52, 8 (2010), 792–805.

[90] Heiko Koziolek. 2011. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM, 3–12.

[91] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*. IEEE Computer Society, 259–268.

[92] Philippe B Kruchten. 1995. The 4+ 1 view model of architecture. *IEEE software* 12, 6 (1995), 42–50.

[93] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17 (2015), 184–206.

[94] Jihyun Lee, Sungwon Kang, and Chang-Ki Kim. 2009. Software architecture evaluation methods based on cost benefit analysis and quantitative decision making. *Empirical Software Engineering* 14, 4 (2009), 453–475.

[95] Emmanuel Letier, David Stefan, and Earl T Barr. 2014. Uncertainty, risk, and information value in software requirements and architecture. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 883–894.

[96] Yan Liu, Muhammad Ali Babar, and Ian Gorton. 2008. Middleware architecture evaluation for dependable self-managing systems. In *International Conference on the Quality of Software Architectures*. Springer, 189–204.

[97] Timothy A Luehrman. 1998. Strategy as a portfolio of real options. *Harvard business review* 76 (1998), 89–101.

[98] Sara Mahdavi-Hezavehi, Vinicius HS Durelli, Danny Weyns, and Paris Avgeriou. 2017. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology* 90 (2017), 1–26.

[99] Sara Mahdavi-Hezavehi, Matthias Galster, and Paris Avgeriou. 2013. Variability in quality attributes of service-based software systems: A systematic literature review. *Information and Software Technology* 55, 2 (2013), 320–343.

[100] Harry Markowitz. 1959. Portfolio Selection, Cowles Foundation Monograph No. 16. *John Wiley, New York. S. Moss (1981). An Economic theory of Business Strategy, Halstead Press, New York. TH Naylor (1966). The theory of the firm: a comparison of marginal analysis and linear programming. Southern Economic Journal (January)* 32 (1959), 263–74.

[101] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26, 6 (2004), 369–395.

[102] Indika Meedeniya, Aldeida Aleti, Iman Avazpour, and Ayman Amin. 2012. Robust archeopterix: Architecture optimization of embedded systems under uncertainty. In *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*. IEEE, 23–29.

[103] Indika Meedeniya, Aldeida Aleti, and Lars Grunske. 2012. Architecture-driven reliability optimization with uncertain model parameters. *Journal of Systems and Software* 85, 10 (2012), 2340–2355.

[104] Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. 2011. Architecture-based reliability evaluation under uncertainty. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM, 85–94.

[105] Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. 2014. Evaluating probabilistic models with uncertain model parameters. *Software & Systems Modeling* 13, 4 (2014), 1395–1415.

[106] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 147–156.

[107] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. 2009. Models@ run. time to support dynamic adaptation. *Computer* 42, 10 (2009).

[108] Klara Nahrstedt, Hongyang Li, Phuong Nguyen, Siting Chang, and Long Vu. 2016. Internet of mobile things: Mobility-driven challenges, designs and implementations. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE, 25–36.

[109] Nanjangud Narendra and Prasant Misra. 2016. Research Challenges in the Internet of Mobile Things. (March 2016). https://iot.ieee.org/newsletter/march-2016/research-challenges-in-the-internet-of-mobile-things.html

[110] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. 2013. A method for taxonomy development and its application in information systems. *European Journal of Information Systems* 22, 3 (2013), 336–359.

[111] William L Oberkampf, Jon C Helton, Cliff A Joslyn, Steven F Wojtkiewicz, and Scott Ferson. 2004. Challenge problems: uncertainty in system response given uncertain parameters. *Reliability Engineering & System Safety* 85, 1-3 (2004), 11–19.

[112] Bendra Ojameruaye, Rami Bahsoon, and Leticia Duboc. 2016. Sustainability debt: a portfolio-based approach for evaluating sustainability requirements in architectures. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 543–552.

[113] Magnus Osterlind, Pontus Johnson, Kiran Karnati, Robert Lagerstrom, and Margus Valja. 2013. Enterprise architecture evaluation using utility theory. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops*. IEEE, 347–351.

[114] Ipek Ozkaya, Rick Kazman, and Mark Klein. 2007. Quality-attribute based economic valuation of architectural patterns. In *Economics of Software and Computation, 2007. ESC'07. First International Workshop on the*. IEEE, 5–5.

[115] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (1992), 40–52.

[116] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[117] RJ Pooley and AAL Abdullatif. 2010. Cpasa: continuous performance assessment of software architecture. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*. IEEE, 79–87.

[118] Aurora Ramirez, José Raúl Romero, and Sebastian Ventura. 2019. A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software* 149 (2019), 382–395.

[119] Samuel T Redwine Jr and William E Riddle. 1985. Software technology maturation. In *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 189–200.

[120] Ralf H Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and simulating software architectures: The Palladio approach*. MIT Press.

[121] Matthias Rohr, Simon Giesecke, Marcel Hiel, Willem-Jan van den Heuvel, Hans Weigand, and Wilhelm Hasselbring. 2006. A classification scheme for self-adaptation research. (2006).

[122] Banani Roy and TC Nicholas Graham. 2008. Methods for evaluating software architecture: A survey. *School of Computing TR* 545 (2008), 82.

[123] Thomas L Saaty. 2008. Decision making with the analytic hierarchy process. *International journal of services sciences* 1, 1 (2008), 83–98.

[124] Software Engineering Institute (SEI). 2018. *Reduce Risk with Architecture Evaluation*. Technical Report. SEI/CMU.

[125] Mary Shaw and David Garlan. 1996. *Software architecture: perspectives on an emerging discipline*. Vol. 1. Prentice Hall Englewood Cliffs.

[126] Dalia Sobhy, Rami Bahsoon, Leandro Minku, and Rick Kazman. 2016. Diversifying Software Architecture for Sustainability: A Value-based Perspective. *Proceedings of 2016 the European Conference on Software Architecture (ECSA)* (2016).

[127] Dalia Sobhy, Leandro Minku, Rami Bahsoon, Tao Chen, and Rick Kazman. 2020. Run-time evaluation of architectures: A case study of diversification in IoT. *Journal of Systems and Software* 159 (2020), 110428.

[128] Kevin J Sullivan, Prasad Chalasani, Somesh Jha, and Vibha Sazawal. 1999. Software design as an investment activity: a real options perspective. *Real options and business strategy: Applications to decision making* (1999), 215–262.

[129] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. 2001. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, Vol. 26. ACM, 99–108.

[130] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. 2010. Vision and challenges for realising the Internet of Things. (2010).

[131] Michael Szvetits and Uwe Zdun. 2016. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling* 15, 1 (2016), 31–69.

[132] Brendan Tansey and Eleni Stroulia. 2007. Valuating software service development: integrating COCOMO II and real options theory. In *Economics of Software and Computation, 2007. ESC'07. First International Workshop on the*. IEEE, 8–8.

[133] Gerald Tesauro. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing* 11, 1 (2007), 22–30.

[134] Lenos Trigeorgis. 1996. *Real options: Managerial flexibility and strategy in resource allocation*. MIT press.

[135] Catia Trubiani, Indika Meedeniya, Vittorio Cortellessa, Aldeida Aleti, and Lars Grunske. 2013. Model-based performance analysis of software architectures under uncertainty. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 69–78.

[136] M Jeroen Van Der Donckt, Danny Weyns, M Usman Iftikhar, and Ritesh Kumar Singh. 2018. Cost-Benefit Analysis at Runtime for Self-adaptive Systems Applied to an Internet of Things Application.. In *Proceedings of ENASE 2018, Portugal*.

[137] Jan Salvador van der Ven, Anton GJ Jansen, Jos AG Nijhuis, and Jan Bosch. 2006. Design decisions: The bridge between rationale and architecture. In *Rationale management in software engineering*. Springer, 329–348.

[138] Naruemon Wattanapongskorn and David W Coit. 2007. Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty. *Reliability Engineering & System Safety* 92, 4 (2007), 395–407.

[139] Lloyd G Williams and Connie U Smith. 2002. PASA SM: a method for the performance assessment of software architectures. In *Proceedings of the 3rd international workshop on Software and performance*. ACM, 179–189.

[140] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 38.

[141] Jie Yang, Gang Huang, Wenhui Zhu, Xiaofeng Cui, and Hong Mei. 2009. Quality attribute tradeoff through adaptive architectures at runtime. *Journal of Systems and Software* 82, 2 (2009), 319–332.

[142] He Zhang and Muhammad Ali Babar. 2010. On searching relevant studies in software engineering. (2010).

[143] Dongbin Zhao and Zhaohui Hu. 2011. Supervised adaptive dynamic programming based adaptive cruise control. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 318–323.

[144] Ruiqing Zhao and Baoding Liu. 2004. Redundancy optimization problems with uncertainty of combining randomness and fuzziness. *European Journal of Operational Research* 157, 3 (2004), 716–735.

[145] Liming Zhu, Aybüke Aurum, Ian Gorton, and Ross Jeffery. 2005. Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process. *Software Quality Journal* 13, 4 (2005), 357–375.

[146] Peter Zimmerer. 2018. Strategy for continuous testing in iDevOps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 532–533.