Stability in Software Engineering

Salama, Maria; Bahsoon, Rami; Lago, Patricia

published in
IEEE Transactions on Software Engineering
2021

DOI (link to publisher)
10.1109/TSE.2019.2925616

document version
Publisher's PDF, also known as Version of record

Stability in Software Engineering: Survey of the State-of-the-Art and Research Directions

Maria Salama, Rami Bahsoon, Patricia Lago

Abstract—With the increasing dependence on software systems, longevity is becoming a pressing need. Stability is envisioned as a primary property to achieve longevity. Stability has been defined and treated in many different ways in the literature. We conduct a systematic literature review to analyse the state-of-the-art related to stability as a software property. We formulate a taxonomy for characterising the notion, analyse the definitions found in the literature, and present research studies dealing with stability. As architecture is one of the software artefacts with profound effect throughout the software lifecycle, we focus on software engineering practices for realising architectural stability. The analysis results show a wide variation in dimensions when dealing with stability. The state-of-the-art indicates the need for a shift towards a multi-dimensional concept that could cope with runtime dynamics and emerging software paradigms. More research efforts should be directed toward the identified gaps. The presented taxonomy and analysis of the literature aim to help the research community in consolidating the existing research efforts and deriving future developments.

Index Terms—software architecture, longevity, quality, stability, architecture design, architecture evaluation, architectural stability

1 Introduction

Stability is a long-term property of utmost importance for any software system throughout its whole lifecycle, from design and implementation to actual operation, management, maintenance and evolution. A system, if engineered and developed with stability in mind, can provide a good basis for supporting technical changes and cost-effective maintenance and evolution [1].

Motivation. The increasing dependence on software systems and services will make software *longevity* a highly desired feature [2] [3]. A long-lived system is capable to remain largely intact while supporting changes (e.g. evolution, maintenance and runtime changes) [4]. Generally, it is widely accepted that stability is a property to reflect such concerns, as it deals with the impact resulting from changes [1]. Longevity and stability are essentially two sides of the same quality issue and affect each other. A stable basis provides a foundation for building quality and long-living systems [1], as longevity is highly dependent on the ability to retain stability.

Large industrial software systems require delivering an acceptable level of performance for their end-users. For instance, end-users of cloud-based systems or Amazon Web Services would not tolerate performance levels to disregard their service level agreements (SLAs) [5]. Given the dynamic unpredictable operational conditions, these challenges are affecting the quality of service (QoS) provisioned. By that, stability is becoming essential for providers and practitioners to prevent performance degradation and enforce QoS at runtime, especially in peak demand [5]. From an economic point of view, stability is desirable to safeguard customers' satisfaction and service provider reputation, as well as avoid SLAs penalties. The paramount importance of stability is evident in the industrial context since it influences software quality, cost and longevity.

As software systems undergo through many cycles of maintenance [6], associated costs and efforts are reduced in case ripple changes and extensive re-factoring are limited [1]. As such, software artefacts, designed in a way that the impact of changes is minimal, i.e. stable, remarkably affect the maintenance process [7]. Since evolution is unavoidable, iterative long-term changes are implemented during evolution cycles for facing changing requirements [8]. Software artefacts capable to support evolutionary changes would bring long-term benefits without phasing-out [6]. Interest in software reuse is also increasing, as stakeholders are concerned with building software systems that are larger, more reliable, less expensive and within shorter time-to-market [9]. A software artefact, component or module could be easily reused in multiple contexts and projects, if modifications and ripple changes are controlled. That is, stability is advantageous for effective software reuse [9].

Problem Description. Dealing with stability as a software property poses questions on how to characterise it and consider it during all software lifecycle phases, i.e. how to design, operate, maintain and cost-effectively evolve software systems. In characterising stability, definitions found in the software engineering literature refer to different perspectives and phases. Researchers in the field of software engineering have studied stability with respect to different software artefacts. For instance, some have considered code stability with maintenance efforts, and architecture stability with software evolution, while others studied the design of stable components, and requirements stability. The term was also found inter-linked with various software quality attributes (e.g. resilience, robustness), and sometimes within software engineering practices (such as evolution and maintenance).

As for considering stability, stability-related studies are scattered across many research communities within the software engineering discipline that has widely-spread during the last decades, with many emerging paradigms and domains (e.g. cloud-based, service-oriented, embedded, real-

time systems) [2]. Also, there have been many stability developments in other disciplines (e.g. control theory, dynamic systems theory), and we are interested in studying how software engineering could benefit for classical notion of stability for incepting and realising stability as a software property. This indicates the need for a taxonomy for clearly defining and characterising the notion of stability in software engineering, in order to analyse the current literature and help in realising it as a software property for all artefacts.

Research Approach and Aim. To provide a comprehensive overview of the current state-of-the-art and connect knowledge, we conducted a systematic literature review and examined 166 primary studies from multiple research databases. We iteratively developed the taxonomy from the analysis of the primary studies. The taxonomy is, then, used to classify and analyse current research for a comprehensive overview. We also performed cross-analysis of different dimensions, to derive gaps and directions for further research.

In light of such characterisation and findings in the literature, we are interested in further investigating related engineering practices in the software architecture sub-discipline. We argue that architecture is the appropriate abstract level for understanding stability given the complexity and scale of modern software systems. First, architecture has been recognised as a key asset in building complex software-intensive systems [2], and have a profound effect throughout their life-span. Second, architecture has been recognised as key for software reuse [9], maintenance and evolution [8]. Third, architectures have a strong impact on the quality of service delivered and system quality attributes [9], as well as a bridge between requirements, design and implementation. Also, the state-of-practice has shown that architectures help accommodating changes resulting from the high volatility in requirements that are becoming the norm [10].

With this review, we aim to achieve the following: (i) provide a holistic and comprehensive understanding of the notion of stability and related problems, and provide systematic guidance for the use of the term in software engineering, (ii) motivate the need for a new perspective in considering stability as a software property, and (iii) help in identifying research gaps, get new insights from the taxonomy, and guide the research community to develop further methods based on the taxonomy.

Contributions. In this survey, we review the state-of-the-art related to stability in software engineering. In details, the contributions of our work include:

- a characterisation taxonomy for the notion of stability as a software property emerged from the current literature of software engineering.
- analysis of the stability definitions found in the literature, to study how the property has been considered and treated, and shed its relation with other quality attributes and software engineering practices.
- an overview of the current state-of-the-art related to the stability of the different software artefacts.
- a review of the software engineering practices supporting architectural stability.
- a discussion of a new perspective for considering stability as a software property, and identify current research gaps, point out directions for future research.

Scope of the survey. In this survey, we focus on the stability of the software product itself. Aspects related to the stability of the development process (e.g. project management, social aspects, knowledge management), however, should be studied to complement it, as aspects of the product itself and its development process are inter-wined [11]. Stability of software product lines and software ecosystems is not covered, as both are different in nature from software systems [12] [13], and require special consideration. Studies about contextual aspects (e.g. changes [14], uncertainty [15]) are considered in the context of stability.

Related surveys. There are several related surveys in the field of software engineering, but to the best of our knowledge, they do not focus on stability. There are studies reviewing related quality attributes in software engineering, such as sustainability [16] [17] [18], maintainability [19], reliability [20]. Other studies reviewed related engineering practices, such as software reuse [21], self-stabilisation [22], software ageing and rejuvenation [23]. In the area of software evolution, authors in [7] presented a review on the research of architecture evolution, and [24] presented a framework for classifying and comparing architecture evolution.

With respect to software architecture, there are many surveys, such as [25] [26]. Others focused on architecture related practices, such as architecture optimization methods [27], architecture design rationale [28], decision-making techniques for architecture design [29], analysis methods [30], evaluation methods [31], and methods that handle multiple quality attributes in architecture-based self-adaptive systems [32]. There are other studies reviewed architectural concepts similar to architectural stability, such as architecture erosion (the result of modifications that violate architectural principles and can degrade system performance and shorten its lifetime span) [33], architectural degeneration [34] and architectural decay [35]. With respect to architecture properties, authors in [36] presented a survey on reliability and availability prediction methods for software architectures, maintainability prediction in [37], and sustainability evaluation in [38].

Organisation. This paper is organised as follows. In section 2, we present the basic concepts and background on the notion of stability. In section 3, we briefly describe the survey method. In section 4, we present a taxonomy for characterising stability, and analyse the concept definitions in section 5. In section 6, we review the treatment of stability in software engineering, and in section 7, we present the research findings related to architectural stability. Gap analysis and research directions are presented in section 8, and threats to validity are discussed in section 9. The manuscript is concluded in section 10.

2 Background

As mentioned in the introduction, this paper is concerned with the stability of software as a property and focuses on the architecture level. Before presenting our survey, however, we introduce the perspective adapted on relevant concepts and terms. To this aim, the following introduced the basic concepts (in section 2.1) and background underlying the notion of stability (in section 2.2).

2.1 The Basic Concepts

Software system. A software system is a set of software components, computer programs, procedures, rules (and possibly associated documentation and data) pertaining to the operation of a computer system or an information processing system that satisfies an end-use function [39]. The **system boundary** is the common frontier between the system and its operating environment. The **function** of a system is "what the system is intended to do" and is described by the functional specification in terms of functionality [40]. The **behaviour** of such a system is "what the system does to implement its function" [40]. The **service** delivered by a system (in its role as a provider) is its behaviour as it is perceived by end-user(s) [40].

Software lifecycle. The life cycle of a software system consists basically of the *development* and *operation* phases [40]. The development phase includes all activities till the decision that the software is ready for operation to deliver service, such as requirements elicitation, conceptual design, architectural design, implementation and testing [40]. The operation phase begins when cutover issues are resolved, the product is launched, and the system is deployed, configured and put into operation to start delivering the actual service in the end-users' environment [40] [39]. The former phase is known as *initial development* or *design-time*, and the latter is usually referred as *runtime*. After the development and launch of the first functioning version, the software product enters to different cycles of maintenance and evolution stages till reaching the phase-out and close-down [8] [6] [40] [39]. During a maintenance cycle, minor defects are repaired, while the system functionalities and capabilities are extended in major ways in an evolution cycle [6].

Quality Attribute. We use the definition of the IEEE Standard for Software Quality Metrics [41], defining a quality attribute as "a characteristic of software, or a generic term applying to quality factors, quality sub-factors, or metric values". According to the same standard, a *quality requirement* is defined as "a requirement that a software attribute be present in software to satisfy a contract, standard, specification, or other formally imposed document" [41].

System behaviour. The behaviour of a system is the "observable activity of the system, measurable in terms of quantifiable effects on the environment whether arising from internal or external stimulus" [39]. This is determined by the state-changing operations the system can perform [39].

Software architecture. The concept of software architecture has been defined in different ways under different contexts. In our work, we adopt the definition of the ISO/IEC/IEEE Standard that defines software architecture as the "fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [39]. This definition is in line with early [42] [43] and later definitions [44]. Software architectures provide abstractions for representing the structure, behaviour and key properties of a software system [43]. They are described in terms of software components (computational elements), connectors (interaction elements), their configurations (specific compositions of components and connectors) and their relationship to the environment [45] [46].

Architectural structure. Architectural Structure is "a physical or logical layout of the components of a system design and their internal and external connections" [39].

Architectural style. The architectural style is the pattern of structural organisation and semantic properties that provides a domain-specific architectural design vocabulary together with constraints on how the parts may fit together [39] [46] [47]. An architectural pattern is described by its structure (what are the components) and its behaviour (how they interact) [48]. Examples include publish-subscribe, peer-to-peer, client-server, pipes and filters, layers.

Architecturally-significant requirements. The architecture should fulfil the software requirements, both functional requirements (what the software has to do) and quality requirements (how well the software should perform) [49] [50]. Functional requirements are implemented by the individual components, while the quality requirements are highly dependent on the organisation and communication of these components [51]. In this context, it is worth to mention that we focus on the *architecturally-significant requirements*, as not all requirements have equal effect on the architecture [52]. This special category of requirements, describing the key behaviours that the system should perform, plays an important role in taking architectural decisions and has a measurable effect on the software architecture [44]. Architecturally-significant requirements are a subset of requirements technically challenging, technically constraining, or central to the system's purpose, and should be satisfied by the architecture [52]. Architecturally-significant requirements are categorised as functional and quality requirements.

Architecture design phase. The architecture design phase is "the lifecycle phase in which a system's general architecture is developed, thereby fulfilling the requirements laid down by the software requirements document and detailing the implementation plan in response to it" [39]. The output of the architectural design phase is an architectural model that describes how the system is organised as a set of communicating components [51].

2.2 The Notion of Stability

The Latin origin "stabilitas" refers to both *firmness* and *steadfastness* [53]. In modern English, stability refers to "the condition of being stable or in equilibrium state", "resistance to change", and "the tendency to recover from perturbations" [53]. The condition of being stable, thus, implies that certain properties of interest do not (very often) change relative to other things that are dynamically changing. These meanings raise further questions, such as what is the stable condition, what is the equilibrium state, what are the types of changes to resist (long-, short-term), what are the perturbations to recover from.

The concept of stability is studied in many disciplines. It has been originated in Physics, as "the degree of being firm, steadfast and free from change or variation when outside conditions change" [53]. Different forms of stability have been defined in many domains, such as ecology, chemistry, economics and mathematics. For instance, *ecological* stability is defined as the ability of an ecosystem to resist changes

and return to an equilibrium state in the presence of perturbations [54] [55]. The *evolutionary* and *dynamic* stability have also been introduced in biology [56]. In the Six Sigma methodology (developed for manufacturing and business process improvement), the stability of a business process is defined as "the ability of the process to perform in a predictable manner over time" [57].

Stability in Systems Theory. In Systems Theory, stability is used to describe "the ability of a system, when kept under specified conditions, to maintain a stated property value within specified limits for a specified period of time" [53] [**?**].

Stability in Dynamic Systems Theory. In the context of dynamic systems, stability is considered as "the ability of a component or system to maintain a fixed level of operation within specified tolerances under varying external conditions" [53]. The basic definition is that "a bounded input produces a bounded response" [58]. Several notions of stability have been introduced in this area, such as Poisson, structural, exponential and asymptotic stability [59]. The modern mathematical Theory of Stability has been established by A. M. Lyapunov —also known as "Lyapunov stability" —and has been widely adopted in dynamic and autonomous systems [60].

Stability in Control Theory. Developed to deal with the behaviour of dynamical systems and support automatic control of closed-loop (feedback) systems, Control Theory has extensively studied stability, as an unstable system will not maintain the controlled variable with the desired value [61] [62] [63]. Stability is an essential property for control systems to capture the robustness of the system, where most closed-loop systems become unstable as gains increase with the attempts to achieve high performance [61] [62] [63]. According to the classical notion of Lyapunov stability, small perturbations to the initial state of the system will affect its behaviour in small variations [61]. Intuitively, in control theory, a stable system is one that, "when perturbed from an equilibrium state, will tend to return to that equilibrium state" [60] [61]. In Optimal and Continuous Control Theory, stability refers to "the continuous behaviour of optimal solutions under perturbations of the problem data", where bounded disruptions have bounded effects [64].

Stability in Distributed Systems. In the paradigm of distributed computing, stability is considered as "a measure of the ability of a mechanism to detect when the effects of further actions (which potentially consume the resource being scheduled) will not improve the system state as defined by a user-defined objective" [58]. Given the importance of schedulers for distributed systems, their stability has been explicitly studied, where a distributed scheduling algorithm has been considered stable if its performance (e.g. response time, throughput) is bounded for any reasonable input (e.g. arrival rate) [65], and would return the system to an equilibrium state following a perturbation [58].

Applying control theory concepts to distributed scheduling, the author has concluded that a definition for stability should include boundaries for reasonable input and behaviour, as well as stability issues of the scheduling algorithm and triggered by the environment [65]. The proposed scheduling algorithms have demonstrated that handling stability is subject to the algorithm and environment under consideration (e.g. real-time environment) [65]. Analysis and experiments conducted on a number of dynamic, globally distributed scheduling algorithms have shown that absolute stability is not always needed for dynamic systems, and relatively minimal instabilities could be tolerated (inspired by control theory) [58]. A stable scheduling algorithm, "following a perturbation of the system state from equilibrium, will return the system to a state of equilibrium and additionally will cease continuing to take actions, which cause changes in system state in finite time" [58].

Self-stabilisation was initially introduced by Dijkstra in the context of robust distributed algorithms [66]. This property ensures that the system autonomously recovers and converges to legitimate behaviour in a finite time after any transient fault [66] [67] [68] [69]. Since Dijkstra's seminal work, recent work by Dolev et al. [70] [71] [72] [73] proposed techniques for designing self-stabilising systems and ensuring that the core layers of the system preserve the property. In more details, Dolev and Rajsbaum [70] have introduced the notion of stability for long-lived consensus distributed systems to reflect the sensitivity of the system decisions between consecutive invocations of the consensus algorithm to input changes. Stability is evaluated using the worst case of output changes when the input changes at most once for each processor in the system [70]. The study of Schmid [74] focused on *structural* (topological) self-stabilisation of distributed systems, to allow dynamic convergence to the desired structure after performance deterioration and ensure continuous availability and functionality.

Discussion. From the definitions in the disciplines presented above, one can notice that the notion of stability encompasses different abilities and different facets, e.g. control theory and distributed systems have mainly focused on the *operational* side of stability, while biology focused on *evolutionary* stability. To summarise, the notion of stability encompasses the following: (i) the ability to resist to changes, (ii) the ability to remain unchanged over time or when external conditions change, (iii) the ability to adapt to changes while remaining intact, (iv) the ability to return to equilibrium state when perturbed from that state, and (v) the ability to maintain a stated property value or fixed level of operation within specified limits under varying external conditions. These abilities have been used to define stability according to the context and purpose of the system subject of question. For example, the ability to resist to changes has been used in the context of evolution, while the ability to maintain a stated property or a fixed level of operation has been used when the system is in operation.

3 The Survey Method

To identify the literature related to stability in software engineering, we conducted the survey following the guidelines of systematic literature reviews [75] [76]. The main research questions (RQs) are defined as follows:

RQ1. How stability can be defined and characterised as a software property?

RQ2. What is the current state of research on software stability?

RQ3. Which engineering practices have been developed by the research community for realising and evaluating architectural stability?

As various definitions are scattered in the literature review, the aim of RQ1 is to identify the definitions of stability, with the goal of getting a sound definition and characterisation of this quality property. RQ2 aims to provide the current state of research on software stability. In RQ3, we focus on architectural stability, with the aim of getting a better insight into the current engineering practices supporting and evaluating architectural stability. This helps us determine how they can fit to new software paradigms and their dynamics, and identify research gaps and potential directions for future research.

The search process was conducted in the following digital libraries: ACM Digital Library, IEEE Xplore, ScienceDirect and SpringerLink. The snowballing technique –following the guidelines [77] –was used to complement the search process. The complete review protocol appears in Appendix A. As a result, we identified a set of 166 primary studies, listed in Appendix B. The characterisation of stability extracted from the primary studies is reported in Appendix C.

In the remaining of this paper, we first present the taxonomy (section 4) we have defined to guide the review. We present the findings of RQ1 in section 5, RQ2 in section 6, and RQ3 in section 7.

4 Taxonomy for Characterising Stability as a Software Property

In reviewing the state-of-the-art in software engineering, we have found that stability has been interpreted in various ways, at different levels and in relation to several aspects. Generally, these efforts point to the multi-dimensional nature of stability and the need for characterisation. In this section, we summarise the different dimensions of stability in software engineering research in general and present them in a comprehensive taxonomy that incorporates the results of our literature review.

Purpose of the taxonomy. Taxonomies of concepts are a basic scientific tool to structure and advance the understanding [78] [79]. A structured representation of concepts and relationships in a certain area is fundamental for representing, understanding and communicating that knowledge, as well as providing the opportunity for further research advances [79]. Taxonomies are found essential to document and accumulate knowledge on software engineering phenomena too [80].

We develop the taxonomy with the purpose of: (i) characterising stability as a software quality property, providing researchers and practitioners with a common vocabulary, and (ii) analysing software engineering practices, identifying gaps and suggesting research directions.

Development of the taxonomy. A plausible way to capture the dimensions of the taxonomy and systematically study a topic is the widely-adopted 5W+1H pattern (What, Where, When, Why, Who and How) [81] [82] [83] [84]. But our taxonomy formulates the questions in a different order, because of the nature of the property under consideration. The *Where* question is our starting point in capturing the other dimensions. The *Who* and *How* questions are determined by the other questions.

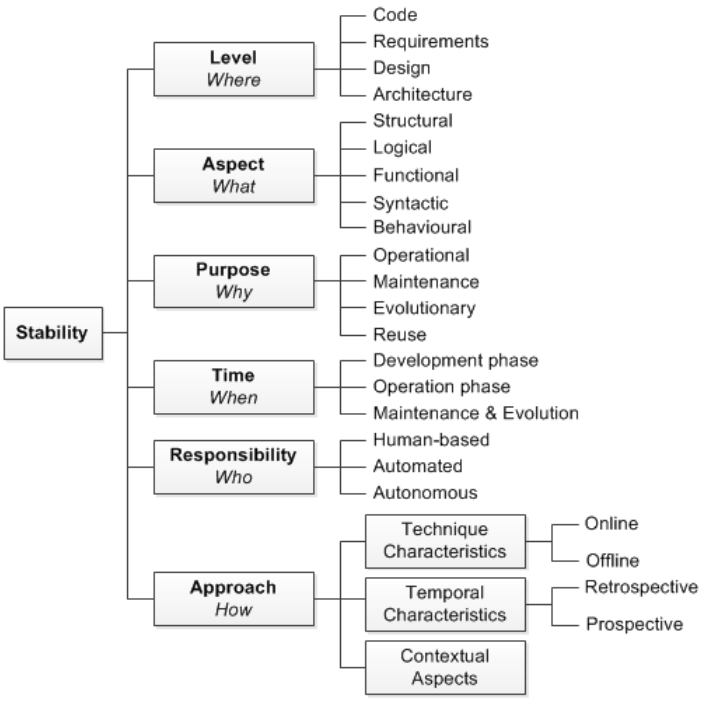

Figure 1: Taxonomy of Software Stability.

As indicated in the review protocol (in Appendix A), we had formulated an initial taxonomy based on the 5W+1H scheme prior to reviewing the primary studies. We, then, refined it after the data extraction and synthesis. The development of the taxonomy followed the common practice found in the literature (e.g. [27]). In more details, the answers to the questions have been extracted from the primary studies and clustered based on similarity. If an answer did not belong to an existing category, a new category has been created and the answers to the previously analysed studies were revisited for possible re-categorisation.

Dimensions of the taxonomy. Figure 1 gives an overview of the proposed taxonomy. Table 1 shows how the taxonomy answers the questions. The dimensions of the taxonomy and the resulting characterisation of stability are described below.

- *Level (Where).* This dimension is concerned the level at which stability is considered, i.e. which artefact of the software. Stability can be considered at different levels of the system, such as code, design, architecture, requirements. In order to realise stability as a software property, it should be aimed at the different software artefacts. This dimension sets out to locate the level at which this property is considered, which in turn determines *who* is responsible about realising stability (e.g. architect, system designer, etc.) and *how* it will be realised (i.e. the techniques to be used).
- *Aspect (What).* It is not sufficient to only identify the level at which stability is considered, the aspect of each level should also be identified. In literature, different aspects have been considered, varying between physi-

Table 1: Dimensions of Stability Taxonomy

Question	Taxonomy	Description	Details
Where	Level	At which level/ artefact in the software is stability considered?	code, requirements, design, architecture
What	Aspect	Which aspect of stability is considered?	structural, functional, logical, syntactic, behavioural, physical
Why	Purpose	What is the purpose/ objective of stability?	operational, maintenance, evolutionary, reuse
When	Time	At which phase of the software lifecycle is stability considered?	development phase, operation phase, maintenance and evolution phase
Who	Responsibility	Who is involved in realising stability?	human-based (requirements engineer, software designer, architect, system administrator), automated, autonomous
How	Approach	What is the technique used for realising stability?	characteristics (online, offline), temporal aspects (retrospective, prospective), evaluation, measurement, validation

cal aspect (related to equipment and physical resources, i.e. malfunction of a physical machine may put the system into an unstable state), structural (structure of the artefact), logical (system's configuration). Some aspects could be found at different levels, such as structural stability of the design and the architecture, while other aspects could be unique within a certain level, such as the syntactic aspect of the code.

- *Purpose (Why).* This dimension deals with the purpose/ motivation for considering stability. If we consider the stability of the artefact throughout long-term modifications during software evolution, that is considering stability for the evolutionary purpose. The maintenance purpose could also be considered for short-term modifications. If the objective is to stabilise the runtime behaviour, the operational purpose would be appropriate. Software reuse aims at having stable software artefacts that can be reused across multiple systems or projects with minimal modifications.
- *Time (When).* The time of considering stability shall be addressed in this dimension, i.e. when to consider and evaluate stability. Stability should be considered in the development phase, throughout the operation of the software, and in the maintenance and evolution cycles. In more details, considering stability at the architecture level, architects should evaluate the stability of alternative architectural structures at the development phase. At the operation phase, stability should be considered while handling the runtime concerns of the software product. The time of consideration identifies the approach to be used for realising stability and associated responsibilities. Thus, the distinction between design-time and runtime is essential for considering stability at different phase throughout the software life cycle.
- *Responsibility (Who).* This dimension addresses the degree of human interaction and automation in realising stability, i.e. who is responsible for realising stability and at which degree the process could be automated without human intervention. This is also related to the level, time and approach dimensions. Architects or system designers are responsible for realising this property during the development phase at different levels. At the operating phase, stability could be considered by system administrators responsible for managing the operational concerns of the software product, or by automated processes, or autonomously evaluated in the case of self-adaptive systems. [85].
- *Approach (How).* This dimension addresses the approach or engineering practices for realising stability, i.e. *how* stability is realised. As mentioned in the previous dimensions, the techniques to be employed for realising stability are to be determined by the level and the time at which stability is addressed, such as architecture evaluation during development or evolution phases where techniques could be retrospective or predictive. During operation, techniques to evaluate stability could be online or offline.

On the use of the taxonomy. The taxonomy could be used to: (i) guide the systematic evaluation for stability and replace ad-hoc practices, (ii) help in scenario consideration and generation, (iii) help in identifying queries for the modelling, simulation and analysis for qualities that interrelate with stability, (iv) sensitivity analysis of the stability evaluation to other qualities, (v) improve the coverage in situation where it is difficult to guarantee completeness for the dimensions that could affect stability.

The dimensions of the taxonomy can be useful also for evaluating stability. As scenario-based evaluation is promising for evaluating stability [86] [87], the evaluation can be highly sensitive to the chosen scenarios. The exercise and the conclusion can be sensitive and/or biased to the input and expertise of the evaluators. As an example, the evaluator may not systematically arrive on comprehensive lists of scenarios that can evaluate for the stability related issues of interest. It may further focus on some scenarios while ignoring the others. As a result, it would be difficult to ensure completeness when choosing these scenarios. The taxonomy, then, can systematically assist architects and evaluators to think about scenarios that relate to the indicated dimensions. It can also provide the architect with systematic guidance that can replace the ad-hoc practice in reaching these scenarios.

The taxonomy hopes to serve as a systematic account of the dimensions that need to be considered when stability is discussed as quality.

On the completeness of the taxonomy. As mentioned in the scope of the study, our taxonomy focuses on stability aspects of the software product itself. Meanwhile, the taxonomy should be complemented with stability aspects of the development process, as aspects of the product itself

and its development process are inter-wined [11]. Technical and social aspects of the development process should also be taken into account. Knowledge (design, requirements and architecture knowledge) management, documentation and drift are influencing factors in extending the software longevity [88] [89] [90] [1]. The well-being of the development community highly affects the software engineering process [91]. Examples include changing organisational structure, the social communities in development [92] [93] [94] [95]. Another example is the stability of the maintenance process itself which has proven to affect the reliability of the maintained software [96]. Such technical and social aspects are hard to be understood without field and empirical studies and need to be studied in deep separately to complement the stability of the software product.

In this section, we presented a taxonomy for characterising stability as a software property. Stability can be considered at different levels, with different aspects, for different purposes at different stages through the software life cycle. Researchers and practitioners should be aware of these dimensions and integrate these dimensions throughout the software lifecycle.

5 Defining and Characterising Stability (RQ1)

In this section, we present the definitions of stability found in the software engineering literature, analyse the characteristics of these definitions (RQ1). We also discuss related quality attributes and software engineering practices, as well as propose an approach for defining it as a software quality property.

5.1 Definitions of Stability

Many definitions for stability were found in the software engineering literature from different perspectives and for different software paradigms. We collected the definitions from the literature and analysed them according to our taxonomy (listed in Table C.1). A cell marked with "–" means that this definition does not give information related to this dimension. It is worth noting that the analysis of the definitions was conducted based on the wording of the definition (not the contents of the studies where they appeared).

Analysing the definitions found in the literature, they partially covered the different abilities of stability (cf. section 2.2): (i) the ability to resist to (ripple effect of) changes (e.g. the definitions of [**?**], [**?**], [**?**], [**?**]), (ii) the ability to remain (largely) unchanged over time or when external conditions change (e.g. [**?**], [**?**], [**?**]), (iii) the ability to adapt to changes while remaining intact (to a big extent) (e.g [**?**], [**?**], [**?**], [**?**]), (iv) the ability to return to equilibrium state (within a defined time) when perturbed from that state ([**?**]), and (v) the ability to maintain a stated property value or fixed level of operation (within specified limits) under varying external conditions ([**?**], [**?**]). Some definitions encompass more than one ability, such as the definitions of [**?**] [**?**] covered the abilities to resist to changes and to remain unchanged.

Table 2 summarises the mapping of the notion of stability abilities, different dimensions and the definitions analysis. Out of the primary studies, 14 papers contributed on explicitly defining stability. We observe that the majority of the definitions focused on the first three abilities related to changes, while other abilities explicitly related to the operational and behavioural side are ignored to a big extent. Precisely, none of the definitions has explicitly focused on the ability to maintain a fixed level of operation. On the terminological side, we noticed that the definitions widely varied between abstraction (e.g. [**?**]) and precision (e.g. [**?**] [**?**]). Also, stability was defined in different ways by the same authors according to the perspective of their study, such as [**?**] and [**?**] for maintenance and evolutionary purposes, respectively.

Examining the aforementioned definitions, we found that these definitions covered some of the taxonomy dimensions. Some definitions considered the resistance to ripple effect of changes for the *maintenance* or *evolutionary* purposes, where it was considered with respect to the changes occurring due to the maintenance or evolution activities respectively (e.g. [**?**], [**?**]). Other definitions considered the ability to remain largely unchanged over time in the context of evolution, where a software artefact is considered stable if it remains unchanged over different versions. The ability to adapt to changes while remaining intact has been used in some definitions in different contexts (i.e. operational, maintenance and evolutionary purposes), where the software artefact adapts responding to different types of changes that result from runtime operation, maintenance or evolutionary activities respectively. The abilities to recover from perturbations and to maintain a fixed level of operation or a stated property within specified tolerances was usually put in the *operational* context, where stability was considered with respect to the level of operation and the operational perturbations during runtime.

5.2 A Working Definition for Stability

In developing a working definition for stability, the task has proved to be challenging when balancing between abstraction, precision and comprehensiveness. As an example, in an attempt for analysing and modelling runtime architectural stability for self-adaptive software in [97], we developed an analysis and modelling approach to understand how stability can be sensitive to different qualities and how the inclusion/exclusion of qualities of interest can impact the analysis. The approach provides a concrete justification on the need for treating stability as a quality that can be studied in isolation of other qualities. Indeed, the stability of the architecture is directly correlated to the ability of the architecture to continue to meet its qualities during operation, maintenance and evolution.

Given the multi-dimensional, case- and context-specific nature of stability, we argue that a unique definition of stability might not be possible or accurate. We opt, instead, for a working definition, based on a set of principles that help consider stability as a software property. The approach is, therefore, to select an ability and build the definition around the taxonomy dimensions. Such approach allows building a conceptual framework for thinking about stability and a set of dimensions to approach.

A working definition should, then, include an ability (e.g. ability to keep unchanged) based on the intended pur-

Table 2: Mapping of stability notion, dimensions and definitions

Ability	Purpose of Stability	Definitions
(i) ability to resist to ripple effect of changes	Mnt, Ev	[?], [?], [?], [?]
(ii) ability to remain largely unchanged over time	Mnt, Ev	[?], [?], [?]
(iii) ability to adapt to changes while remaining intact	Op, Mnt, Ev	[?], [?], [?], [?]
(iv) the ability to return to equilibrium state when perturbed from that state	Op, Mnt, Ev	[?]
(v) the ability to maintain a stated property value or fixed level of operation within specified limits under varying external conditions	Op	[?], [?]

pose (e.g. evolutionary) and cover the different dimensions of the taxonomy (level, aspect). As an example, one possible definition could be *the ability of the architecture's structure to keep unchanged along with the time to endure evolutionary changes*. Such definition targets stability at the architecture level for evolution purpose and focuses on the structural aspect. On the same level, another possible definition could be *the ability of the architecture's behaviour to maintain a fixed level of operation (or recover from operational perturbations) within specified tolerances under varying external conditions*, to consider the behavioural aspect at the architecture level for operational purpose. Considering the design level and structural aspect from the maintenance perspective, a possible definition could be *the ability of the design's structure to resist changes (or adapt to changes) due to maintenance activities*.

The sensible treatment for stability depends, then, on the system of interest, its domain and context, the attribute(s) in question of stability, and the time of consideration. For instance, the treatment of the architecture's structural stability could be considered during the development phase (i.e. a prospective approach by the architect) to plan for possible future evolution, or later during the maintenance and evolution phase (i.e. retrospective approach) for possible lessons learnt. Similarly, the treatment of the architecture's behavioural stability could be considered at design-time (for making architecture decisions capable to keep the desired level of operation), or during runtime (either by autonomous online adaptation or offline maintenance). Yet, the desired level of operation depends on the attribute(s) subject of interest (e.g. response time for real-time systems should be kept stable), and on the software artefact (i.e. architectural stability considers architecturally-significant requirements only).

While we believe the studies reviewed provide important aspects of stability, we view that a pragmatic view is required to offer a systematic way for practitioners and researchers to deal with stability as a software property. Figure 2 shows our proposed pragmatic view for stability integrating the dimensions of the taxonomy. The intersection between different dimensions could help the community to identify the dimensions ignored in the literature and motivate possible research directions. Also, rather than thinking in an isolated manner about stability, we should be looking for methods and tools to explore inter-dependencies between the different dimensions throughout the software lifecycle for a more integrated thinking to achieve software longevity.

5.3 Related Quality Attributes

An investigation of the literature has shown the existence of a number of quality attributes related to similar abilities defined under the stability umbrella, such as maintainability, resilience, robustness, reliability, etc. Below, we present the definitions of these quality attributes, shedding lights on their relationship with the different aspects of stability and discuss their differences. Table 3 summarises the quality attributes inter-related with stability and the respective dimension of stability.

The related quality attributes have been extracted from the primary studies, and mainly based on the ISO/IEC 9126 standards for software quality model [**?**]. When definitions were not found in the primary studies, we performed cross-referencing in the primary studies and used on seminal work and other surveys published in the same data sources, in order to find how these concepts are defined and related to stability. The use of the standards for software quality model hopes to guarantee a comprehensive coverage for the widely acknowledged qualities inline with the standards and hopes to reduce selection bias if the qualities would be discussed in an ad-hoc way in case of absence of similar standards. The practice also brings unification to the terminology reducing subjectivity.

Resilience. Resilience is defined as "the ability to successfully accommodate unforeseen environmental perturbations or disturbances" [98]. Resilience was also considered a sub-characteristic of dependability, where the former is defined as "the system's ability to continue providing available, responsive and reliable services under external perturbations such as ... unexpected load spikes or fault-loads" [105], and "the persistence of the avoidance of failures that are unacceptably frequent or severe, when facing changes" [98] [99] [100] [102]. In the previous definitions, changes are runtime changes during operation —that is operational stability. Resilience is the concept with high interference with stability and often used as a synonym in the context of software evolution. Inspired by ecological systems [112] [98], resilience was seen as the persistence of a property and a measure for the ability to absorb changes and still persist. Resilience has been, then, presented as "the persistence of service delivery that can justifiably be trusted, when facing changes" [98] [99] [100] [102]. In this context, evolutionary changes are the concerned ones. Another definition for resilience inter-linked with trustworthiness is that resilience enables to "assess whether the system is able to maintain trustworthy service delivery in spite of changes in its environment" [101].

Trustworthiness. Trustworthiness is the "assurance that a system will perform as expected" [40]. Trustworthiness

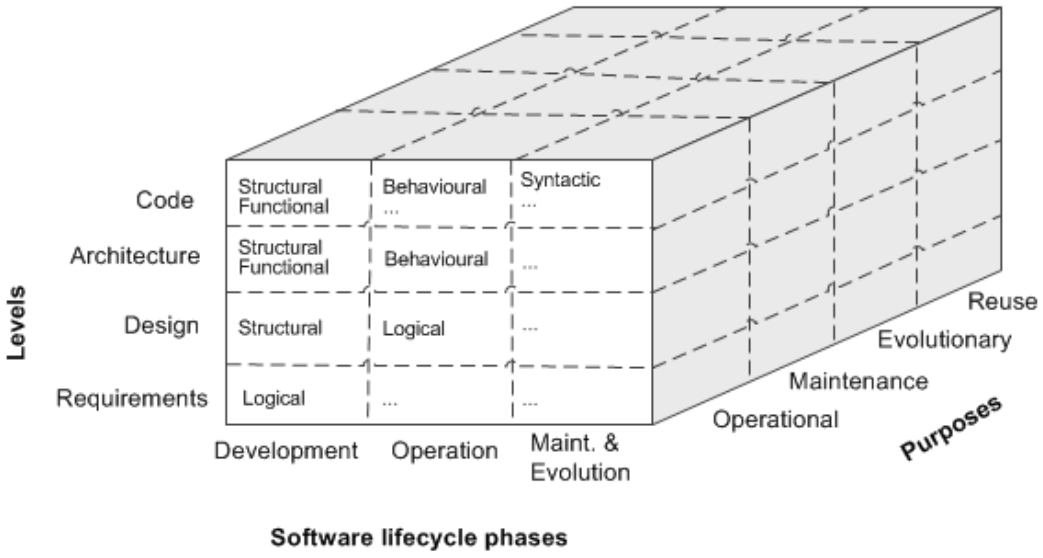

Figure 2: Pragmatic view of stability as a software property

was not widely used as a term within the research community, though the concept is usually found expressed informally in the context of dependability.

Robustness. Robustness is the other synonym for stability found in literature, as both words have close meaning. Robustness has been commonly accepted as a mean to differentiate candidate architectures and mitigate the risk of architecture decisions through the development [103]. A system is considered robust if it "retains its ability to deliver service in conditions which are beyond its normal domain of operation" [104] [98]. This attribute has usually been put in the context of abnormal operating conditions. From a control-theoretic perspective, robustness has been considered as "the property that a system only exhibits small deviations from the nominal behaviour upon the occurrence of small disturbances" [**?**], that could be behavioural stability.

Reliability. Reliability has earlier been concerned with "how well the software meets the requirements of the customer" [113] [3]. Following the ISO/IEC/IEEE standards and vocabulary, reliability is "the capability of the software product to maintain a specified level of performance when used under specified conditions" [**?**] for a specified time [39]. According to the seminal work on the taxonomy of dependable and secure computing [40], reliability is considered as one of the attributes of dependability (where both encompasses fault-tolerance). It is defined as "the continuity of a correct service", that is the extent to which the system is available when required and behave as expected [3]. The previous definitions are interconnected with the stability abilities for operational purposes. According to the latest IEEE Recommended Practice on Software Reliability [**?**], code stability and release stability are considered measures of software product reliability. The former is measured by corrective action effectiveness, while the latter is measured by the MTBF (mean time between failure) metric. By these definitions, reliability could be seen as a form of stability in maintenance setting.

Dependability. Dependability has been considered as "the ability of a system to provide dependable services in terms of availability, responsiveness and reliability" [105]. A widely adopted definition is "the ability to deliver services that can justifiably be trusted in spite of continuous changes" [40] [98]. This definition puts emphasis on the justification of trust of the delivered service. An alternate definition is "the ability to avoid service failures that are more frequent and more severe than acceptable" [40] [98]. The dependability attribute abstractly encompasses the trustworthiness attribute [114]. In the context of stability, one can characterise dependability as a kind of behavioural stability that ensures the quality of service provided during operation.

Maintainability. According to the ISO/IEC 9126 standards for software quality model [**?**], maintainability is one of the main characteristics of software, defined according to the standards as "the capability of the software product to be modified" [**?**]. It is divided into a set of attributes related to the ability to make specified modifications (analysability, changeability, testability and stability) [**?**]. Modifications may include corrections to handle errors, improvements or adaptations in response to changes in the environment and functional requirements [**?**]. Such modifications could be for operation, maintenance, or evolution purposes.

Modifiability. Modifiability is the ability of a system to be easily modified quickly and cost-effective to changes in the environment, requirements or functional specification [44] [115]. Modifications to a system can be categorised as extensibility (the ability to acquire new features), deleting unwanted capabilities (to simplify the functionality of an existing application), or restructuring (rationalising system services, modularising, creating reusable components). Portability (adapting to new operating environments) was also considered as one of the sub-characteristics of modifiability [44], while it was identified as one of the main quality characteristics in the ISO/IEC 9126 standards for

Table 3: Quality attributes inter-related with Stability

Quality Attribute	Goal	Intersecting Attributes	Ref.	Purpose of stability
Resilience	ability to accommodate unexpected perturbations/ absorb evolutionary change and still persist	dependability, robustness, evolvability, trustworthiness	[98], [99], [100], [101], [102]	Op, Ev
Trustworthiness	ability to perform as expected	dependability, reliability	[40]	Op
Robustness	ability to operate beyond normal operational conditions	resilience, dependability, reliability	[103], [104], [98]	Op
Reliability	ability to be available when required and behave as expected/ accept corrective actions effectively	dependability, fault-tolerance, maintainability	[?], [?]	Op, Mnt
Dependability	ability to deliver justifiably trusted services in spite of continuous changes	trustworthiness, reliability	[40], [98], [105]	Op
Maintainability	capability to be modified	modifiability, changeability	[?], [?]	Mnt
Modifiability	ability to make changes quickly and cost-effectively	maintainability, changeability, flexibility	[?], [44]	Mnt, Ev, Re
Changeability	ability to enable implementation of modifications	maintainability, modifiability, flexibility	[?]	Mnt, Ev
Flexibility	ability to be modified for use beyond the original design with acceptable effort	maintainability, modifiability, changeability	[39], [106]	Ev, Re
Adaptability	capacity to adjust to changes in the environment	sustainability, dependability, trustworthiness	[?], [107]	Op, Mnt, Ev
Evolvability	capacity to support adaptation and accommodate future changes in requirements on the long-term	sustainability	[108], [109]	Ev
Sustainability	capacity to preserve the function over an extended period of time and to be cost-effectively maintained and evolved	evolvability	[110], [111]	Op, Mnt, Ev

software quality model [?]. In both cases, the type of changes concerned is the long-term evolutionary, which could be regarded as stability for evolutionary and reuse purposes.

Changeability. Another equivalent term to modifiability is changeability, which is defined in the ISO/IEC 9126 standards for software quality model [?] as "the capability of the software product to enable a specified modification to be implemented". According to this standard [?], changeability reflects the ability of the software artefact to accept possible future changes, while stability is observed after the change has taken place [?].

Flexibility. Flexibility is "the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed" [39]. Flexibility is mainly about future changes of software and is considered relative to these expected changes, similar to modifiability [106]. Distinguishing it from other properties like adaptivity and changeability, flexibility is defined as "the property of a software system to allow conducting certain changes to the system with acceptable effort for modifying the system's implementation artefacts" [106].

Adaptability. Adaptability is the capacity of software to dynamically adjust itself (behaviour, structure or configuration) when reacting to changes in its operating environment in order to keep its services in a good condition, i.e. meeting the requirements (including functionalities and QoS) [?] [107] [116] [117]. Meanwhile, adaptive maintenance is the "modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment" [39]. Here, adaptability is put for two different purposes, operational and maintenance, given to the time and type of adaptation. The difference between adaptability and runtime stability is the former is concerned with runtime changes only and the latter is concerned with runtime changes while keeping other attributes unchanged (like the structure of the architecture).

Evolvability. Evolvability is the capacity of software systems to support adaptation and accommodate long-term changes of new requirements and contexts of use over time with the least possible cost [108] [109]. The continuously changing stakeholders' requirements make evolvability an important software property to be explicitly addressed throughout the system's lifespan [109]. This property focuses mainly on the long-term evolutionary properties and changes without becoming progressively less useful [108].

Sustainability. Sustainability is defined as "the capacity to endure and preserve the function of a system over an extended period of time" [111]. Though the concept of sustainability has usually been considered in the sense of green computing and associated with the ecological environment [4] as the capability of meeting the present needs without compromising future needs [110], a modern vision according to Lago et al. [111] should consider four major dimensions —economic, social, environmental (improving human welfare by protecting natural resources), and technical (supporting long-term use and evolution). Also, sustainability is associated with longevity, where a sustainable software is "a long-living software system which can be cost-efficiently maintained and evolved over its entire life cycle" [110] and architectural sustainability is "the set of factors that promote an architecture's stability and longevity during system evolution" [1].

To comprehensively address these quality attributes, it

is worth mentioning that in some contexts, dependability, trustworthiness and resilience were addressed in the context of security, i.e. dependability in delivering reliable, secured and confidential services [40], trustworthiness in delivering confidential and trusted services, and resilience to security attacks and correlated faults [118] [105], or failures during operations [119]. In this research, we do not consider security as part of these attributes, as the former requires special attention.

The side-by-side comparison in Table 3 elucidates that these concepts are essentially intersecting in some aspects. For instance, both dependability and performance metrics were embedded in benchmarking resilience, in order to evaluate "if a system is *effective* and *efficient* in accommodating changes" and thus considered to be resilient [99] [120]. Another example is dependability and resilience. While dependability is considered design-time attribute that deals with possible faults [100], as well as runtime attribute as previously mentioned in definitions [40] [98], resilience by definition is maintaining the same properties if evolution takes place in environmental factors [100]. Thus, resilience "encompasses the ability to resist and recover from changed environment, operational domains or requirements unknown at design-time" [100], i.e. dependability.

As developed over the aforementioned concepts and definitions, it could be concluded that stability as a property partially integrates some aspects of these attributes. As an example, stability as the ability to adapt while remaining intact partially covers the adaptability property while considering ripple effect changes or a fixed value for a stated property. These properties could also be a mean to achieve stability (such as flexibility), or an indicator of stability (e.g evolvability). As Bass et al. pointed out [44], these terms can be confusing and less meaningful without a concrete scenario, which is the approach we will adopt in defining and realising stability.

5.4 Related Software Engineering Practices

As mentioned above that related software quality attributes could be a mean or an indicator of stability, software engineering practices for achieving these qualities could also be related to achieving stability. We briefly discuss related engineering practices below.

Stability and software maintenance. Stability has been considered as an important factor contributing to the maintenance process [**?**] [121] [2]. Stability is used in the maintenance process to indicate the accounting of ripple effects as a consequence of modifications [**?**]. Stability measures are used in conjunction with other factors affecting the maintenance process, to estimate maintenance costs and possible errors when generating maintenance plans [**?**].

Stability and evolution planning. It has been argued that the primary long-term goal of software artefacts is to guide the system evolution, and stability has been strongly suggested as a primary criterion for evaluating alternative designs and taking design decisions [**?**] [**?**]. Such decisions are taken based on the long-term impact on stability when planning for possible evolution paths or in automated planning for evolution [122] [123]. According to the Lehman's laws of software evolution [124], stability "means planned and controlled change, not constancy".

Stability and software ageing. Software ageing is the phenomenon facing long-running complex systems over time as long as they evolve [23] [125], with a number of visible signs, such as performance degradation, design degradation, or quality reduction [126] [23]. It has been attributed in many ways [127], such as architectural/ design erosion and architectural drift [42] [128]. Architectural and design erosion refers to conflicts occurring in previous decisions due to changes leading to system brittleness (i.e. fragility or instability), while architectural drift refers to "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture" [42]. Software ageing has been usually associated with preventive maintenance, meanwhile, recent research identifies proactive rejuvenation and preventing premature software ageing (poor decisions made during development phase will age software quicker) as counteract strategies to software ageing [23] [127]. The challenge is, then, to keep the architecture or design aligned throughout the system lifetime [129], which should consider stability as a quality characteristic. As an example, the ability to adapt to changes while remaining intact is important for a long-running system, i.e. the architecture structure is said to be eroded when changes become risky, cost-ineffective and time-consuming [126].

Stability and software reuse. Software reuse is the engineering practice of using existing software artefacts (e.g. architecture, knowledge) or software knowledge to build new software [9]. The purpose is to increase productivity and software reliability, as well as reduce development cost and time [9]. Stability is an important factor to consider both when building software artefacts to be reused later and when selecting the reusable artefact [**?**].

Stability and incremental software development. Incremental software development has been used in the software industry, as an alternative to the waterfall model, when shorter development periods and time-to-market are required [130]. This requires dividing the work into increments with prioritised features. When the new features are added to the previous increments, the resulting design and architecture might change [**?**]. Stability should, then, be evaluated with each increment, in order to ensure continuity of the development without difficulty and unnecessary expenses

Stability and Adaptation. Adaptation and self-adaptation have emerged to deal with dynamic/runtime changes in the system itself or in its operating environment [82] [83] [131]. As inspirations were drawn from Control Theory in building adaptive systems, stability has been suggested as primary criteria for evaluation [**?**] [**?**]. Stability measures the system responsiveness, as such a system is said stable "if its response to a bounded input is itself bounded by a desirable range" [**?**], i.e. the controlled variables are within a required range. This is characterised as the stability of the adaptation goal [**?**]. Stability is also considered as an observable property for the adaptation process, defined as "the degree in that the adaptation process will converge toward the control objective". An adaptation, indefinitely repeating the action or making frequent adaptations, will risk not improving or even degrading the system to unacceptable levels [**?**] [132]. Even though adap-

tation mechanisms have been widely investigated, stability was not explicitly tackled [?]. The shortcoming of current software engineering practices regarding stability is that the stable provision of certain quality attributes essential for end-users (e.g. response time for real-time systems) is not explicitly considered in the adaptation decision taken during runtime [131]. Besides, the adaptation process does not address the adaptation properties that affect the quality of adaptation, such as accuracy, settling time and resources overshoot [?] [132].

6 Stability in Software Engineering (RQ2)

In this section, we review stability in software engineering (RQ2). First, the primary studies are classified and analysed based on the taxonomy described in Section 4. Then, we present the findings at the different levels, perspectives and aspects. As the stability level is the main dimension identifying the other dimensions, we present the survey on stability based on the different levels. For each level, we discuss the other dimensions.

6.1 Analysis Results of Primary Studies

A mapping between the data extracted from primary studies and the taxonomy dimensions is shown in Appendix C. Tables C.2, C.3, C.4 and C.5 show the characterisation of stability at the code, requirements, design and architecture levels respectively, classified according to the dimensions of the taxonomy found in the reviewed studies. Below, we present analysis results of the primary studies.

6.1.1 Demographic Analysis

Figure 3 shows the distribution of the selected primary studies over time. It is noted that the interest in stability as a software property started back to 1977, with a few number of studies throughout the next two decades. The hype has remarkably started to increase since 1998. As the search was performed during September 2017, this interprets the decrease in the number of studies in 2017. The studies focusing on architectural stability fall almost under the same distribution.

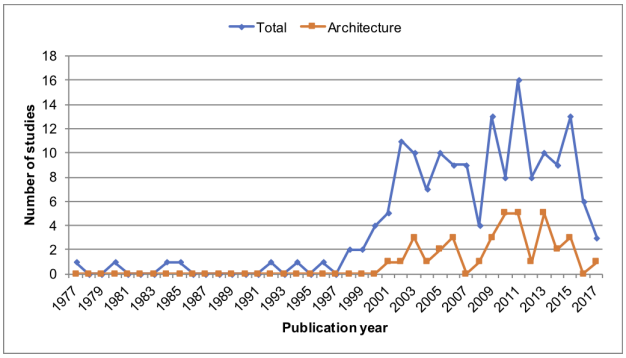

Figure 3: Number of studies per year

The distribution of the type of publications (Figure 4) is: 97 conference papers, 48 journal articles, 18 book chapters, and 3 technical reports (ISO and IEEE standard documents). The good percentage of journal articles and book chapters relatively indicates the maturity of the subject.

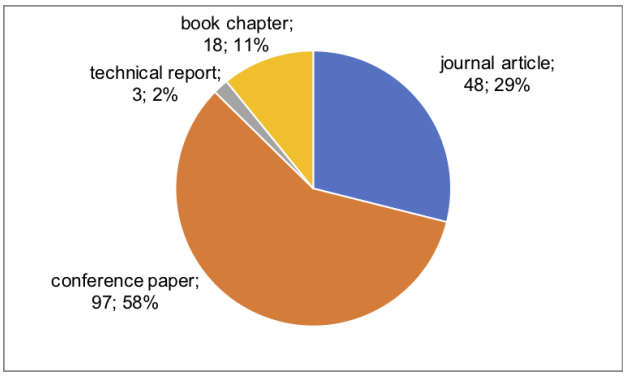

Figure 4: Number of studies per publication type.

6.1.2 Quantitative Analysis

In the following analysis, we put a study under the N/A category when we found that no information is given in that study with respect to a certain dimension.

6.1.2.1 Level (Where): Figure 5 shows the distribution of studies considering stability at different levels. The results show that a significant number of studies for the design level (77 studies), followed by a less significant number for the code (42 studies) and architecture (37 studies) levels. The requirements level is ignored to a big extent compared to the other levels (12 studies).

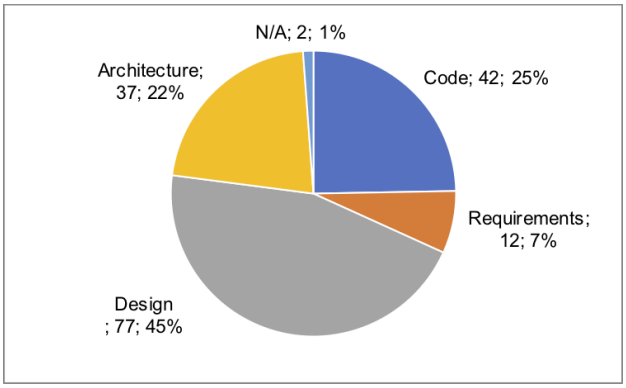

Figure 5: Number of studies per level.

6.1.2.2 Aspect (What): Analysing the different aspects of stability found in the studies, Figure 6 shows this distribution. The majority of the studies covered the structural, logical and syntactic aspects (91, 81 and 38 studies respectively). The behavioural and functional aspects of stability received much less attention (30 and 14 studies respectively).

6.1.2.3 Purpose (Why): The distribution of studies considering stability for different purposes is shown in Figure 7. We found that 61, 42 and 37 studies were concerned about stability for evolutionary maintenance and reuse purposes respectively. The operational purpose was not extensively considered like the other purposes (21 studies).

6.1.2.4 Time of consideration (When): Analysing the time dimension, Figure 8 shows the distribution of studies according to the time where stability has been considered in the studies. This shows that 36% of the studies were

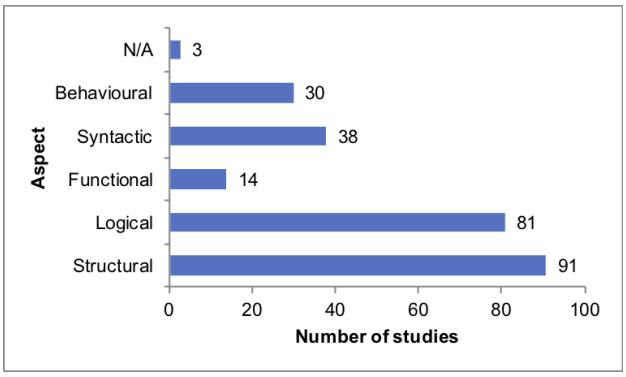

Figure 6: Number of studies per aspect.

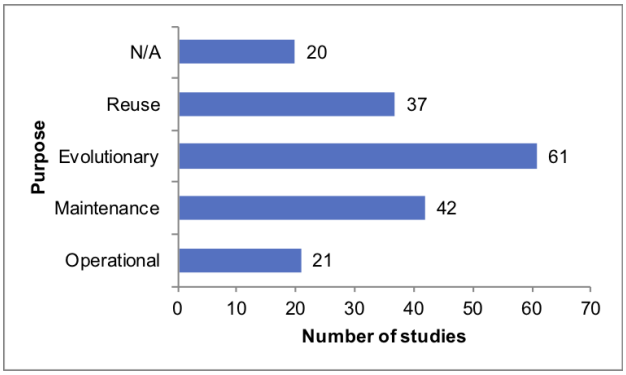

Figure 7: Number of studies per purpose.

concerned about stability during the development phase, and 38% during maintenance and evolution, whereas the operation phase received much less attention (10%).

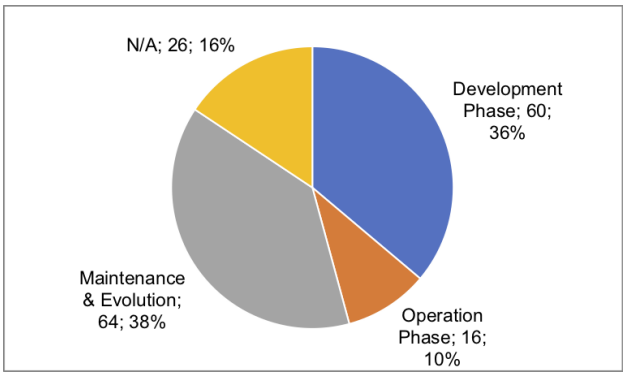

Figure 8: Number of studies per time of consideration.

6.1.2.5 Technique (How): The distribution of techniques by their temporal characteristics found in the studies is shown in Figure 9. The results show a significant number of prospective techniques (49%) in comparison with retrospective ones (26%), with a similar percentage of studies not proposing techniques (i.e. standard documents, philosophical papers describing the concept). The percentage of prospective technique could be interpreted as studies discussing design techniques and design patterns fall under this category.

6.1.2.6 Responsibility (Who): Figure 10 shows the distribution of studies for stability related to the *who* dimension. For most of the studies (113 studies, 65%), the proposed techniques are human-based, i.e. the analysis or evaluation for stability is performed manually or using human judgement. The next largest sets are automated (20 studies, 12%) and autonomous approaches (15 studies, 9%).

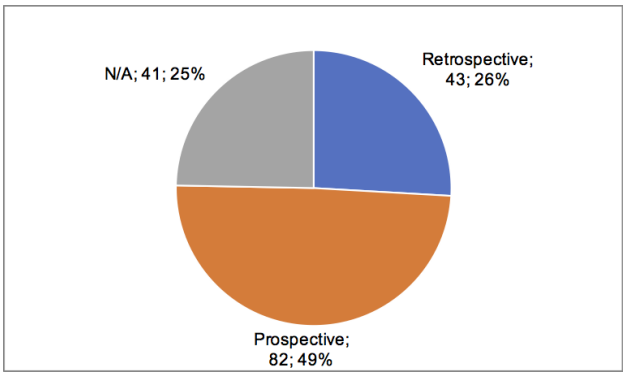

Figure 9: Number of studies per technique.

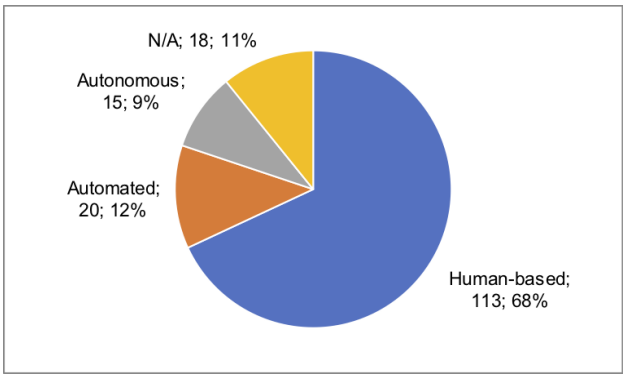

Figure 10: Number of studies per responsibility.

6.1.3 *Correlating Stability Dimensions*

To zoom into the current research state, we used the levels, aspects and purposes for stability as crosscutting dimensions, as shown in Figure 11. The number of studies appears in the circle of each two crosscutting dimensions. We can clearly see that maintenance, evolution and reuse purposes of stability are the most dominant across all levels. The operational purpose appears to be a research gap on all the levels. It is also noticeable that the design and code levels have received attention for the different purposes (with the exception of the operational one) and different aspects. Though it might be argued that stability is not required in some cases of these correlations, such as the requirements level at the operation phase, other correlations are strongly required, as in the case of architecture level during the operation phase.

Correlating the different levels of stability and the phase when stability is considered (Figure 12), the height of each column in the plot represents the number of studies for each level and phase. The development, maintenance and evolution are the most considered phases at all levels. Considering stability at the operation phase is another research gap to be filled for almost all the levels.

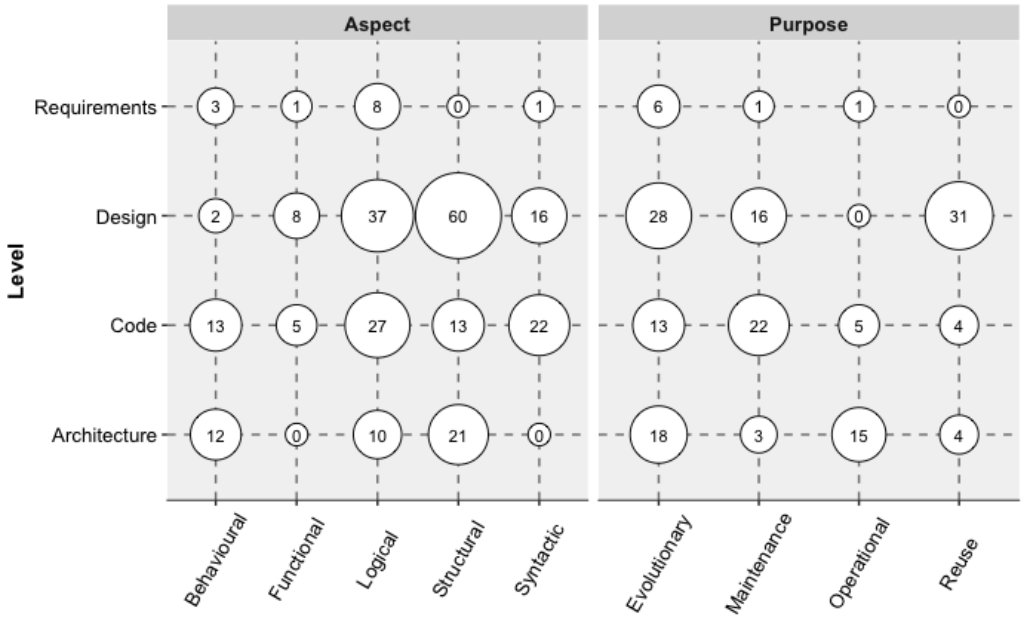

Figure 11: Correlating stability levels, aspects and purposes.

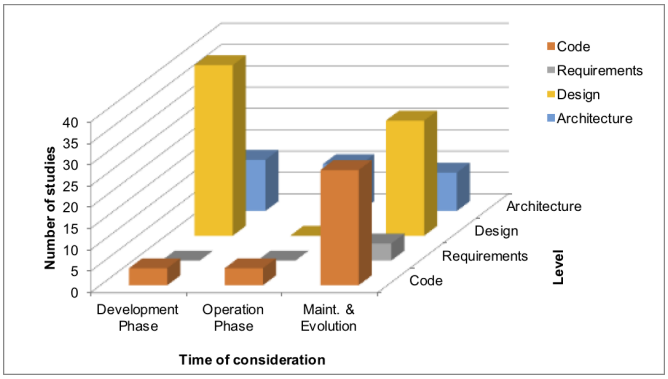

Figure 12: Correlating stability levels and time of consideration.

6.1.4 *Architectural Stability*

To provide a closer overview of stability at the architectural level, we constructed a systematic map for the different purposes during the different phases of the software lifecycle, as shown in Figure 13. The references in each circle represent the studies related to both dimensions for different stability aspects, with the total number of studies appearing between brackets beside the aspect. The systematic map clearly identifies that studies focused mainly on the evolutionary perspective during the development phase, as well as the different purposes during the maintenance and evolution stage. Meanwhile, all other purposes during the development phase are ignored. Also, the operational purpose is only considered while at operation, without earlier planning during development or at later stages. Details of these studies will be discussed in Section 7.

6.2 Levels, Aspects and Purposes of Stability

In the Software Engineering discipline, explicit discussions about stability are traced back to 1977, where Soong [?] studied the stability of a program code with respect to the propagation of changes when maintenance activities are undergoing. Over the following decades, the software engineering community made significant advances in software requirements, design and architecture. Each of these sub-disciplines has studied stability in many different ways and provided insights on software engineering practices for improving the quality of software systems.

6.2.1 *Code level*

6.2.1.1 Maintenance purpose: The earliest mention of stability is found at the code level of software programs [?], where stability has been defined as "the resistance to the amplification/ propagation of changes that have been made to a given program". In this work, the *strucutral* stability aspect of a program has been considered, where distinctions are made between the logical structure and the information structure of a program. Quantitative analysis is derived to measure the information structure of a program. The techniques used are the method of connectivity matrix and random Markovian process, assuming that stability involves the behaviour of the program undergoing alterations, i.e. *behavioural* stability [?].

Following the same concept, Yau and Collofello have considered stability for *maintenance* and defined it as "the resistance to the potential ripple effect that the program would have when it is modified" [?]. The two aspects of stability considered are the *logical* and performance (*behavioural*) stability, where the former is "a measure of the resistance to the impact of a modification to a module on other modules in the program in terms of logical considerations", and the latter is the same measure in terms of performance considerations [?] [?].

In the studies mentioned above, given the definition and measurement of stability in relevance to changes made to a program, stability has been considered for *maintenance* purpose. The difference between the works of [?] and [?]

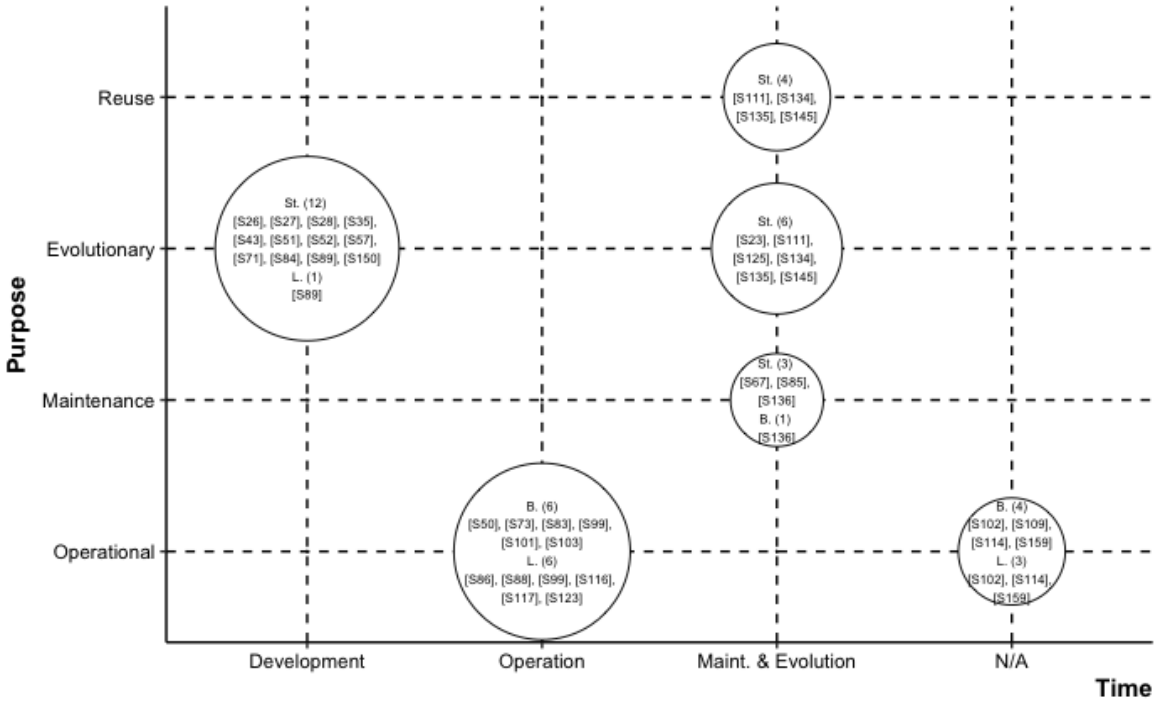

Figure 13: Systematic map of stability at the architecture level.

in defining and considering stability is that the former is a *retrospective* approach (i.e. studying the propagation of changes made) and the latter is a *prospective* approach (i.e. studying potential effects of changes).

The work of Yau and Collofello has been applied to literate programs [**?**]. The studies of Black [**?**] [**?**] have also considered the reformulation of Yau and Collofello's ripple-effect algorithm, and proposed an approximation algorithm for automatic computation of ripple effect measures. Bevan and Whitehead [**?**] developed an approach for identifying and classifying unstable components and code regions (identified as a set of related elements that have changed together many times), using history from configuration management.

In setting a practical model for measuring maintainability, many studies have considered source code metrics. For instance, authors in [**?**] have studied the mapping between stability —as one of the sub-characteristics of maintainability according to [**?**] —and code level properties (such as the size of the system, duplication of code, unit complexity, unit length, number of units and number of modules). Meanwhile, the survey of [**?**] has refined the mapping of the latter model into a weighted mapping and considered further system properties, such as unit interfacing, inward and outward coupling. In studying code quality benchmarking for improving maintainability, Baggen et al. [**?**] have considered mapping source code metrics and maintainability sub-characteristics of the ISO standards [**?**]. Stability has been related to the unit interfacing and module coupling properties. The study of [**?**] have also considered analysis and quantification of maintainability sub-characteristics, where stability metrics are based on the change density in the number of subclasses, coupling between objects, depth of inheritance tree, directly called components and the number of entry/exit points. The same metrics, along with the number of unconditional jumps, have been considered in [**?**] for open-source software.

The development practice known as "code cloning" (i.e. duplication of code fragments with/out modification) and its stability are known to affect the maintenance efforts. If the cloned code is changed less often (i.e. more stable), it will require fewer maintenance efforts. The study of Krinke [**?**] has measured code clones stability by additions and deletions to the code. The results of this study suggested that cloned code is more stable than non-cloned with respect to changes while ignoring the case of deletions. These results are supported by another study on the age of cloned code, where the cloned code is found on average older than non-cloned code (not changed for longer) [**?**]. The studies [**?**] [**?**] have also confirmed the same results and showed that clone stability varies by the clones' characteristics (e.g. length) and the development environment over time. Meanwhile, the empirical studies [**?**] [**?**] [**?**] have revealed that cloned code is generally unstable than non-cloned code, owing such result to differences in the development language, development strategy, stability scenarios and clones types. Another empirical study [**?**] has focused on the different types of clones, types of changes and the frequency of changes in cloned and non-cloned code, where stability has found differing in each type of change.

In studying the stability of logging statements (code snippets for tracking the execution of applications, changes in log statements have been found affecting log processing tools in testing and monitoring. The work of [**?**] helps in determining the likelihood of change in logging statements, in order to select which statements to be used in the processing tools and hence reduce the maintenance efforts.

With the recent shift towards ecosystem-based develop-

ment, Bogart et al. [?] discussed the need for an awareness mechanism based on different stability indicators, such as historical and contextual, to assist developers in analysing the stability of the changed code and evaluating the impact of changes.

6.2.1.2 Evolutionary purpose: In considering stability for *evolutionary* purpose, Li et al. [?] proposed two instability metrics at the implementation level of object-oriented systems —the Class Implementation Instability metric and the System Implementation Instability metric. The first metric measures the evolutionary changes in a class implementation in terms of changes in lines of code (i.e. *syntactic*) between two successive versions. The system metric is the summation of the changes in the classes of the entire system. In [?], the author assessed the stability of concerns implementation, by counting the number of times a given fragment became inconsistent as the code of a system evolves. In [?], stability has been defined as "the ability of a module to remain largely unchanged when faced with newer requirements and/or changes in the environment". It has been measured based on version differences of evolving software modules, where the differences in both source code and structure have been represented by the distance concept (*syntactic* and *structural*). The study of [?] has explored the use of code micro-patterns to evaluate the stability of a system during its evolution and monitor the development of different releases. Information-level metrics were proposed by [?] for measuring the evolutionary stability of software artefacts at the binary level, such as version stability, branch stability, structure stability.

Stability has been studied as an application- and domain-specific. Hou and Yao [?] have studied the stability of APIs (Application Programming Interfaces), for their importance in separating software frameworks and libraries from the implemented applications, where an application can continue to use the updated libraries as long as there is no change in the syntax (*syntactic*) and semantics *logical* of the APIs. This was performed by a detailed analysis of the evolution of APIs, by categorising the changes made to the API according to the domain semantics and design intent. The study of McDonnell et al. [?] has considered the stability of APIs in the Android ecosystem for studying their evolution. This study has focused on the co-evolution *behaviour* of Android APIs and mobile applications, by examining the relationship between API stability and the degree of usage, adoption and bugs in client code. Similarly, *logical* stability of web services —with interfaces described using standard XML-based Web Services Description Language (WSDL) —has been considered in [?]. The WSDL of a web service describes the interface, operations, exchanged data, and the protocol and endpoint to contact the service. Stability has been determined by the unchanged elements of the service interface (*syntactic*) during its evolution from one version to another.

In the context of incremental software development, release planning involves assigning functionalities and bug fixes to different releases, while ensuring quality requirements and factoring efforts needed. Stability analysis for release planning aims at analysing the alternatives release plans against unanticipated changes, such as functionalities changes in their tasks size (*functional*) and dependency (*logical*) [?] [?].

The *functional* aspect of stability has been studied in [?] for providing better evolvability characteristic of software systems. The transformation of functional requirements into implementation-related concepts (e.g. functions or classes) is used to study the process of software coding and derive theorems for implementation that contribute to achieving stability.

6.2.1.3 Reuse purpose: It is widely accepted that stability is an advantageous property for software reuse. A module is considered stable "if its interface or implementation is not undesirably modified and ripple effects do not manifest in the presence of changes" [?] [?], while it is reused in a project "if it is used in more than one context within the software system" [?]. In the exploratory study [?], authors have analysed the relation between stability and reuse, for reaching a better trade-off between them. The stability metrics focused on the degree of modifications in code implementation, considering two forms of modifications: (i) refactoring (structural changes without modifying the code semantics), and (ii) modification (adding, removing or modifying functionalities). The research introduced in [?] focused on analysing the impact of code modularity and composition on stability and reuse. The study of [?] has empirically investigated the impact of reuse of software components on stability (as the degree of modification), i.e. *syntactic* and *logical* aspects. This study has shown that highly reused components are less modified (i.e. more stable) and more concrete to be used across several products and releases.

6.2.1.4 Operational purpose: According to the latest IEEE Recommended Practice on Software Reliability [?], code stability is measured by the corrective action effectiveness (i.e. the percentage of corrective actions that are not adequate) when determining the reliability of a software product. Inspired by Dijkstra self-stabilising distributed systems, self-stabilising has been adopted as an approach for fault-tolerance, where self-stabilising programs automatically recover from bugs and faults to reach the correct state after a finite number of steps [?]. The SJava system was proposed for checking that Java programs are self-stabilising, by adding annotations to the code that captures the flow of execution and return it to the correct state in case of detecting incorrect values [?].

In the domain of multi-agent systems, Bracciali et al. [?] have proposed semantics for defining the notion of stability for the set actions performed by the agents, where the agents' behaviour is coordinated to reach a state similar to the Nash equilibrium state.

In the context of multi-threaded programs, Cui et al. [?] have realised stable multi-threading through schedule memorisation, where past working schedules are memorised and reused on future inputs, which makes the program behaviour stable on different inputs. Based on this idea and using a small set of working schedulers, the stable multi-threading (StableMT) approach reuses each schedule on a range of inputs, mapping all inputs to a reduced set of schedules [?]. By mapping many inputs to the same schedule, the program behaviour is stable against small input perturbations. Practically realising StableMT, a runtime tool has been proposed in [?] to make threads stable during

runtime, by allowing developers to write performance hints in their code for changing schedules when default ones are slow.

6.2.2 Requirements level

The importance of requirements engineering and its role in the success and sustainability of the software product have been recognised and widely accepted by researchers and practitioners in the software engineering discipline [133] [134] [135] [136]. According to the IEEE Recommended Practice for Software Requirements Specifications [**?**], the degree of requirements stability can be determined using "the number of expected changes based on experience or knowledge of forthcoming events that affect the organisation, functions, and people supported by the software system".

The earliest mention about stability requirements was found in [**?**], where the function point metric was used to measure the rate of change. Also, several techniques were discussed to stabilise requirements, such as prototypes, requirements inspection, change and configuration management [**?**]. Yet, the earliest and simplest measurement of requirements stability has been performed using the following equation [**?**]:

$$\frac{\text{Number of initial requirements}}{\text{Total number of requirements}}$$

But this equation does not consider the changes occurring to requirements. The factors affecting the requirements stability have been analysed in [**?**], such as user-, developers-side, system and work environment factors. A process to control requirements change has also been proposed to ensure the success of the software project.

6.2.2.1 Evolutionary purpose: According to the IEEE Standard of Measures to Produce Reliable Software [137] [138], the Requirements Maturity Index (RMI) has used changes from a previous release relatively to the current release as an indication of stability [**?**] [**?**]. These *retrospective* measurements are assessing requirements stability for *evolutionary* purpose.

The RMI was also used for estimating requirements maturity during the development phase [**?**] [**?**]. Such *prospective* approach used the requirements elicitation history to derive the Requirements Maturation Efficiency (RME), which represents "how quickly the requirements reach 100% maturation". Another *prospective* approach has been proposed in [**?**] [**?**] for assessing requirements stability at early development stages. The approach used goal-based model and environmental scenarios with the aim of planning for change and supporting later decisions (design and architecture).

6.2.2.2 Maintenance purpose: An empirical analysis has investigated the correlation between crosscutting concerns and stability at the requirements level, focusing on changes in *functional* requirements that affect the maintainability of the system over time [**?**]. The study provided evidence that certain crosscutting properties have a negative effect on stability.

6.2.2.3 Operational purpose: Focusing on the quality requirements during runtime, the study of [**?**] proposed a solution for autonomous monitoring and extraction of stable behavioural patterns. The extracted stable behavioural patterns are used to detect deviations of the expected behaviour.

6.2.3 Design level

6.2.3.1 Evolutionary purpose: Stability has been widely studied as a design characteristic. The earliest research is the study of [**?**], where design stability is defined as "the extent to which the structure of the design is preserved throughout the evolution of the software from one release to the next" [**?**]. Kelly [**?**] has studied the characteristics of a design that would be stable for long-term software evolution, defined as "if, when observed over two or more versions of the software, the differences in the metric associated with that characteristic are considered, in the context, to be small". Differently, Mannaert et al. [**?**] have considered a system stable with respect to a set of anticipated evolutionary changes, where a bounded input (i.e. a set of anticipated changes) should result in a bounded output (impact on the system). Meanwhile, authors in [**?**] proposed metrics that take into consideration the environmental factors driving software changes in assessing the stability of design decompositions, beside the conventional metrics (e.g. coupling). Namely, the "Decision Volatility" metric assesses the stability of a design decision based on the number of environmental conditions that can cause design change and their impact scope. The metric sums all the design decisions values, and can be formalised using logical models for automated quantitative assessment.

Some studies have explored stability in evolving designs when adding new features. The study [**?**] has been performed for analysing the effect of stability on model composition efforts for evolving design models to add new features. Another study has been performed on comparing different *logical* stability estimation models of classes in the case of incremental development, where stability of the design *structure* is assessed after the changes of adding each increment [**?**].

In the context of object-oriented design, Bansiya [**?**] [**?**] has studied both the *structural* and *functional* stability of object-oriented framework systems for *evolutionary* purpose, where stability is determined by the "extent-of-change" between versions. *Structural* stability is limited to the classes structuring in inheritance hierarchies, while *functional* stability is related to the object's methods of individual classes between versions. Mattsson and Bosch [**?**] [**?**] extended Bansiya's work with an additional aggregated metric, which is the "relative-extent-of-change" metric. Grosser et al. [**?**] [**?**] proposed a case-based reasoning *predictive* approach for stability of Java classes using evolution knowledge of previous versions. They defined stability as "the ease with which a software system or a component can evolve while preserving its design as much as possible", restricting such design preservation to the class interfaces. A distance function is defined to compute the similarity between components and derive stability from it. Similarly, Tsantalis et al. [**?**] proposed a probabilistic *predictive* approach for the same problem, by calculating the probabilities of changes effect for each class in the case of adding and modifying functionalities from previous versions (i.e. *syntactic* and *functional*).

For capturing the stability of evolving object-oriented designs, the "System Design Instability" metric has been

defined by Li et al. [?], where stability is measured by the percentage of classes with changing names (*syntactic*), added and removed (*logical* and *functional*) in two successive designs. This metric has been redefined by [?] [?] [?] for object-oriented systems developed using the agile software process. The work of Azar and Vybihal [?] proposed an ant-colony based *predictive* technique for predicting *syntactic* stability of classes in object-oriented software systems at early development stages. In this work, classes are considered to be stable if their public interfaces (header of the methods) are kept without changes (*sytactic*), addition or deletion of methods (*functional*) across different versions, i.e. for evolution. The impact of code refactoring on class and architecture stability has been studied in [?]. Meanwhile, Bouktif et al. [?] have based their approach for predicting object-oriented class stability on adapting rule sets, which starts from one stability classifier and adapts its rules using genetic algorithm. Another stability prediction model for open-source software systems was built using a combination of Bayesian classifiers [?], which allowed interpretations of the class stability.

In the context of aspect-oriented design, Greenwood et al. [?] studied *structural* stability —that "encompasses the sustenance of modularity properties and absence of ripple-effects in the presence of change" —in evolving applications. Given the impact of crosscutting concern (critical consideration for stakeholders cutting across the software modular structure) patterns on design stability, the work of [?] have studied the aspect- and object-oriented versions of three evolving systems. Meanwhile, the *predicitve* approach of [?] studied the correlation between crosscutting concerns and design (in)stability, in order to anticipate design decisions at early stages of software development.

6.2.3.2 Maintenance purpose: The *prospective* approach of Yau and Collofello [?] measures design stability at any point during the design process, in order to examine modular programs at earlier stages (before producing code) for possible *maintenance* problems. Here, stability is calculated as the reciprocal of the potential ripple effect of modifying the program modules. Potential ripple effects are regarded with respect to the modules being affected with the modification of a certain module, including the modules that invoke that module or are invoked by that module, or share global data with that module, defined as the total number of assumptions made by other modules.

Elish and Rine [?] [?] have adopted the same perspective of [?]. In [?], the focus was on the *logical* stability of object-oriented designs, which indicates "the resistance to interclass propagation of changes that the design would have when it is modified" (*maintenance*). In [?], they studied *structural* stability of object-oriented design that refers to "the extent to which the structure of the design is preserved throughout the evolution of the software from one release to the next", and provided product-related and process-related indicators for stability. The impact of structural design patterns (adapter, bridge, composite and facade) on class diagram stability was discussed in [?], but no empirical evidence was provided. The relation between class stability (using the previous metrics) and maintainability has been experimentally studied in [?].

In the context of design patterns, the work of [?] examined stability of classes participating in different design patterns, and defined (in)stability in such case as "the degree to which a class is subject to change, due to changes in other, related classes, considering the probability of such classes to change as equal to a certain value" (*structural*, *syntactic*). This work distinguishes between the propagation of changes and (in)stability, as they are not correlated in all the cases of design patterns. A class highly depending on other classes would be unstable; however, if the class does not actually change, change propagation would be not high.

A recent study analysed the correlation between class stability (measured using the class stability metric of [?]) and software maintainability [?]. Using one metric for stability, the experiments showed variations in the correlation, but no direct causality was concluded.

6.2.3.3 Reuse purpose: In the column series that appeared in the Communications of the ACM by Fayad [?] [?] [?], the concept of Enduring Business Themes (EBTs) [139] has been adopted, and the Business Objects (BOs) and Industrial Objects (IOs) have been introduced as design artefacts for producing stable software. The idea is to build the core of the software design of the stable themes (EBTs) and objects (BOs) that remain unchangeable, while the changing objects are identified as IOs. This will yield to a stable design to be reused without changing the core. Heuristics for finding EBTs and BOs were also proposed [?]. By dividing the system into stable and unstable modules, Chiang [?] has discussed the integration of stability into the re-engineering process, in order to reduce the impact of maintenance changes, their costs and efforts.

Applying the concepts of EBTs and BOs, the Software Stability Model (SSM) pattern has been introduced for presenting software stability artefacts [?]. The SSM has been employed in the context of software *reuse* to describe the core of a system, which generates a stable design that are extensible for software reuse [?] [?]. An implementation method for the BOs was proposed in [?] to facilitate the application of the SSM in real developments. The SSM has also been applied for building stable real-time systems with adaptive reconfigurable controls [?], magnetic resonance image (MRI) visual analyser stable application [?], and for realising unified software engineering reuse [?].

The concept of Software Stability Model (SSM) has been applied in different ways for the purpose of software *reuse*. Applied to the software analysis patterns, Stable Analysis Patterns have been introduced for analysing the system under consideration and modelling the knowledge of its domain, with the objective of producing stable models with higher reusability [?] [?]. The Stability Analysis Pattern has been further developed for specific purposes design and analysis pattern to provide a reusable core for applications sharing the same core stable for specific purposes. Examples include the visualisation pattern (identifying and extracting the core knowledge of visualisation from the application-specific knowledge) [?], the classification pattern [?], AnyLog pattern [?], AnyTransaction pattern [?], AnyInformationHiding pattern [?], AnyCorrectiveAction pattern [?], the learning pattern [?], and the reputation analysis pattern [?].

Another stability pattern, called Stable Atomic Knowledge (SAK) pattern, has been introduced for representing domain-neutral knowledge to be reused in different do-

mains [?]. Further, domain-specific and -independent patterns are extracted from existing systems to be reused in modelling applications that share the same core knowledge of the domain and the atomic notions knowledge not related to specific domain respectively [?]. Also, an approach for identifying and reusing domain patterns has been proposed in [?]. A domain analysis method driven by stability has been proposed in [?], with the aim of producing stable design artefacts that can be easily reusable within a specific domain.

Considering the system *evolution*, the stability model has been applied for separating crosscutting concerns and encapsulating concerns into stable modules (i.e. less likely to change) over time [?]; and a semi-automated approach was proposed in [?]. Meanwhile, authors in [?] proposed a probabilistic model for estimating stability, by correlating "function points" (used in estimation techniques) of a system to be developed to the EBTs, BOs and IOs. The probabilistic model has also been applied for building autonomic systems [?].

In the context of component-based design, the correlation between the stability of domain business models and components granularity (*structural*) has been studied in [?], where stability has been specifically defined as "the probability that a business model or a component remains stable in a given period of time". A component identification method has been proposed for making design decision about components and their granularity level for *reuse* purposes.

With respect to the reuse of aspect-oriented design, Van Landuyt et al. have proposed a design method for maximising the reuse of pointcut interfaces –which expose crosscutting behaviours to be used in multiple aspects of an application –in applications of the same problem domain [?] [?]. The method is based on the discovery of stable abstractions for the domain model of the application to be mapped onto pointcut interfaces. Automation of the approach was attempted in [?], where an algorithmic procedure for each activity was defined with introducing abstract extension points as a human-based activity.

Meanwhile, adaptation is used to automatically generate adapters for communicating black-box components (e.g. web services, Software-as-a-Service cloud services), which functionalities are required in the composition, and have incompatible communicating interfaces [140] [141]. In such case, a set of components is stable if from some communication buffer bound, the bounded composition is equivalent to any larger bounded composition [?]. Stability-based adaptation aims at generating an adapter with the smallest bound satisfying stability [?].

6.2.3.4 Syntactic aspect: Different aspects of stability for object-oriented design have been studied. The earliest studied is the *syntactic* aspect in [?], which measured the stability of object-oriented design using simple parameter coupling between different objects in a program. Coupling and cohesion in a package have also been adopted in [?] as factors for stability influencing software *maintenance* and *reuse*. Based on the number of methods in a class, Rapu et al. [?] measured stability by the number of added or removed methods between two versions to be used for automatic identification of design problems for *maintenance* activities. Vasa et al. [?] have also studied a number of stability indicative measures (size, popularity and complexity) of modified and newly added classes and interfaces in consecutive releases (*evolutionary*), where more complex classes are more likely to change over time. The aim is to detect the tendency of complexity and change of classes for effectively managing the *evolution* of complex systems.

6.2.3.5 Structural aspect: The *structural* aspect of object-oriented design has been studied in [?], considering stability as an indicator of the design package resilience to change to support *reuse* and *evolution* activities, where the metric indicates "how much the classes are linked to their package", i.e. a package is stable if its classes do not refer to classes in other packages.

The widely adopted concept of "positional stability" of a software package has been proposed by Martin [?], and is calculated using the number of dependencies changing within the package, where a module is less likely to change when modifying other parts of the system if that module depends only on stable modules, assuming that abstract classes are generally stable. A preliminary investigation of the correctness of this assumption on Java interfaces has been conducted in [?]. The work of [?] has employed this metric on the whole software level based on class-to-class dependencies to quantify the stability of the structure of consecutive releases. The study of [?] has explored the use of time-series cross-sectional regression model for statistically evaluating the metrics of Martin, where empirical results have shown that the use of package-level metrics in statistical inference needs precautions in practice. Also, the works of [?] [?] has aggregated the package level stability for measuring the structural design stability of open source systems.

6.2.3.6 Metrics: A metrics suite has been proposed in [?] at the design level for assessing the stability of the UML diagrams during the development phase, namely class, use case and sequence diagrams. These diagrams represent the UML structural, functional and behavioural views respectively.

6.2.4 Architecture level

6.2.4.1 Evolutionary purpose: Stability has been considered as the main criterion for assessing the long-term value of software architectures throughout their evolution [?]. The earliest discussion about stability at the architectural level is found in [?]. Considering that a primary goal of a software architecture is to guide the *evolution* of the system, the stability of an architecture has been defined as "a measure of how well it accommodates the evolution of the system without requiring changes to the architecture" [?]. The *retrospective* approach proposed by Jazayeri [?] aims at analysing how easily the evolution occurred over the successive releases of the software. Meanwhile, the *prospective* approach [?] aims at predicting how the evolution will take place, by examining how the architecture will endure the likely changes. Architectural stability has, then, been defined as "a quality that refers to the extent an architecture (structure) is flexible to endure evolutionary changes in stakeholder's requirements and the environment while leaving the architecture intact" [?]. Tonu et al. [?] have adopted the same perspective for evaluating architectural

stability using a metric-based approach that combines both retrospective and predictive approaches.

Adopting the same definition of [?], Aversano et al. [?] have studied the stability of architecture core design, by analysing evolution historical data with the aim of measuring to what extent the architecture of a system is stable with respect to its core components and identifying potential components for *reuse* and *evolution*. Using the design stability metrics (proposed in [?]), Aversano et al. proposed the Core Design Instability (CDI) and Core Calls Instability (CCI) metrics for measuring the stability of core architectures. These metrics have been further improved in [?], by considering the stability of the external and internal architectural elements in consecutive versions. The proposed metrics capture the degree of consistency of the architectural elements (external stability) and the interaction between architectural elements between consecutive versions (internal stability). Handani and Rochimah have considered the environmental factors to refine stability metrics with features volatility [?].

Also, an approach based on concern traces has been proposed for assessing and predicting architecture design (in)stability using information of early development stages for sustaining the architecture throughout the system evolution [?]. The effectiveness of concern assessment mechanisms to predict architecture (in)stability has been also studied in evolving architectures [?].

Nord et al. [?] have adopted the change impact view of stability and related structural metrics used at the code level in analysing architectural dependency and its impact on the system evolution cost. According to Nord et al., stability, measuring the percentage of modules that are not affected by changes in other modules in the system and is computed as the inverse of the cumulative component dependency that is the sum of direct and indirect dependencies modules have on each other.

6.2.4.2 Reuse purpose: The stability metrics of [?] have been extended by [?] in the context of software reuse, introducing the Reuse Oriented Stability (ROS) metric. The metric has considered the stability of a software system in terms of the *structural* consistency when introducing new bugs during its evolution, which affects its possible reuse.

6.2.4.3 Maintenance purpose: Architectural stability has been defined as "the ability of the high-level design units to sustain their modularity properties and not succumb to modifications" [?] [?]. Assessment of stability of aspect-oriented software architectures design (i.e. *structural*) has been analysed by studying the effect of aspectual decompositions in the presence of architecturally-relevant changes carried during the maintenance phase [?] [?].

Stability of architectural tactics, essential for realising architectural qualities and meeting quality requirements, has been studied in [?]. The study aims to investigate the architectural solutions (i.e. *structural*) that erode over time as a result of not maintaining quality (i.e. *behavioural*) after modifications and maintenance (*maintenance perspective*). By investigating the relation between architectural decisions and changes, the study has found that tactic-related classes tend to change more frequently than non-tactic ones.

6.2.4.4 Operational purpose: In the context of embedded systems, Rafiliu et al. [?] [?] have studied the stability of online resource managers and adaptive feedback-based resource managers of distributed embedded systems running real-time applications, where the resource manager (i.e. controller) is stable if the resource usage is controlled and the behaviour of the system is within a bounded distance from the desired behaviour under all possible runtime scenarios. Porter et al. [?] have proposed an online technique to validate stability and assure correct behaviour under the destabilising conditions caused by different platform effects, based on the behaviour-bounding stability theory of Zames [142]. Meanwhile, authors in [?] have discussed a layered architecture for embedded systems capable to self-stabilise and return to correct execution when operating on unreliable hardware (with a special focus on stabilising memory management), i.e. *physical stability*.

Adapting the notion of input-output stability (IO-stability) from continuous Control Theory, Tabuada et al. [?] have captured two properties of *behavioural* stability for discrete systems, that are: bounded disturbances lead to bounded deviations, and normal behaviour is resumed after a finite number of steps. Wand and Huhns [?] have employed simulations for assessing cloud-based systems in delivering stable service, where simulations are used for predicting stability condition during operation or planning for resources expansion. Stability is considered with respect to (*logical*) system configurations, in terms of the arrival rate of requests, the number of servers in the cloud and the computing capacity of each server.

In the context of adaptive architectures, the study of [?] has considered the stability of performance and QoS (i.e. *runtime behavioural stability*) for adaptive software systems. This work proposed a software service running separately and monitoring performance degradation of the adaptive system during runtime. Applying control-theoretic concepts to software performance control, the service provides an automated mechanism to detect causality assumption –that describe the system behaviour and regulation policies –violations and recover the system from instability using online statistical method. Focusing on the dynamic learning behaviour during runtime operation of adaptive systems, Yerramalla et al. [?] have proposed a stability monitoring approach based on Lyapunov functions for detecting unstable learning behaviour and mathematically analysed stability to guarantee that the runtime learning converges to a stable state within a reasonable time depending on the application.

On the other hand, the stability of adaptation itself has been considered to guarantee more effective and durable adaptation strategies for parallel computations. In [?], the stability degree of an adaptation strategy is said to reflect "how long this choice will be useful for the execution", and "how frequent reconfigurations are issued by the adaptation strategy" [?]. A stable adaptation strategy is the one that "avoids oscillating behaviours and minimises the number of reconfigurations" (i.e. avoid unnecessary modifications) [?]. That is quantitatively measured by the total number of reconfigurations and the average time between consecutive reconfigurations [?]. The control-theoretic approach and adaptation strategies proposed in [?] [?] aim at determining the optimal sequence of adaptations in advance over a specific time horizon, while achieving QoS requirements, and reducing the number of reconfigurations and reconfig-

uration amplitude (the difference in computing resources used between consecutive configurations). This results in performance improvement and operating costs reduction.

Checking the correctness of the system behaviour (i.e. correctness of adaptation), authors in [?] have defined an unstable adaptation manager "if it switches between the adaptation and normal modes infinitely without evolving to the enforcement mode". If adaptation cycles continue without reaching the desired state, the system is said to be in an unstable state. Formal analysis and checking of adaptation manager stability (expressed by linear temporal logic formula) was proposed by reasoning on the policies of the adaptation manager and detecting a specific class of instabilities.

6.3 Main observations and findings

Dimensions of stability. As mentioned in Section 5, the notion of stability encompasses different abilities. The analysed primary studies have mainly focused on the abilities related to maintenance and evolution purpose (refer to Table 2). Meanwhile, there exist much less number of studies focusing on the operational- and behavioural-related abilities (i.e. return to equilibrium state and maintain a stated property or fixed level of operation). With the many definitions of stability, the proposed methods were not profound on clear stability dimensions and founded solid characterisation.

The analysis of primary studies revealed that design is the widely considered level for stability. While we do not ignore the importance of the design and code artefacts, we argue that the architectural stability needs much more attention for the different purposes and aspects, as architectures have a profound effect on the software life-span and the quality of service provisioned. The syntactic, structural and logical aspects of stability have been widely considered. Meanwhile, the behavioural aspect requires similar attention, especially for the quality attributes critical for the developed and running system (e.g. response time for real-time systems). Similar to the other dimensions, evolution and maintenance are the widely considered purposes for stability. Yet, the importance of stability for the operation of software systems should not be ignored, that is keeping the intended behaviour stable during operation. Stability has been an important characteristic to be considered during the development, maintenance and evolution phases, where most of the proposed techniques are human-based activities. Stability during the operation phase also needs to be studied, due to its importance for the quality of service delivered, which requires development of automated and autonomous techniques to be used for evaluating stability while the system is running.

Stability at the code level. At the code level, the "ripple-effect" measure has been identified as a valid measure for the stability of programs [?], where the changed program and its modules are compared, during maintenance, before and after changes for determining the effect of changes on stability [?]. The *logical* aspect of stability has been computed based on the impact of these changes, but in different ways. One way considers software stable if the propagation of changes to existing artefacts is minimal, including adding new artefacts and modifying existing ones, i.e. ripple effect of changes. Another one views stability with respect to adding new artefacts, making additions to existing ones and keeping existing artefacts unchanged. Examples of the former approach is [?], and the latter approach is [?].

Stability at the requirements level. Stability at the requirements level was found limited in the literature, though it is the only artefact which stability is recognised by IEEE standard and recommended practices [?] [137] [138]. The analysis and metrics for evaluating requirements stability were mainly human-based activities for evolution and maintenance purposes focusing mainly on the logical aspect, with the exception of one study [?] which autonomously monitored stable behavioural requirements. Yet, further developments focusing on stability requirements traceability and monitoring in dynamic environments are required to advance the area of requirements stability. Studies focusing on stability requirements elicitation are also required, similar to other complex quality attributes, such as scalability [143].

Stability at the design level. Stability at the design level has been considered in different ways, similar to the code level; the first considers that stability is resisting to any changes made to the design, the second avoiding ripple effects with the addition of new artefacts or modifications to existing ones, and the third is allowing additions to be made to the existing design. Logical stability has been considered for maintenance in the same way as the code level. Structural and functional aspects have been extensively considered for different software paradigms (e.g. object-oriented [?], aspect-oriented [?]).

Stability at the architecture level. With respect to architectural stability, this has been considered mostly as architectural intactness with respect to architecturally-relevant changes carried out for maintenance, evolution and reuse purposes (e.g. [?] [?] [?]), i.e. the ability to accommodate changes while remaining intact. It has been retrospectively analysed over two or more versions of the software ([?]), or predicted for possible future changes ([?]). It is noticeable that the structural aspect for maintenance and evolution purposes is the one considered the most at the architecture level. Implicit consideration of the behavioural aspect for the operational purpose was found for different computing paradigms, such adaptive distributed systems ([?]), and embedded systems ([?], [?]).

Stability metrics. It is obvious that stability metrics would widely differ according to the level, aspect and purpose considered. For instance, measuring stability for maintenance is based on analysing the artifcat (code or design) and measures the interdependencies between modules/components [?] [?]. Such interdependencies reflect "the degree of probability that changes made to other modules could require corresponding changes to this module" [?], i.e. change propagation is associated with weak dependencies. Meanwhile, evolutionary stability is based on evolution history, measuring the differences between two or more versions of the evolving artefact. The version differences are measured using program-level metrics at the code level (e.g. the number of lines of code, variables, common blocks and modules [?], [?]), and using structural differences at the architecture level (architecture-level metrics) [?]. Meanwhile, the two types of metrics would complement each

other; architecture-level metrics would be appropriate for measuring stability of the entire software product as the architectural level, and program-level metrics would be applicable on a single component at the code level [?].

Stability in practice. Empirical studies, case studies and experience reports were not found among the surveyed primary studies. Yet, there is a need for empirical approaches for studying stability, similar to case studies developed to study architecture and software evolution (e.g. [144], [145], [146], [125]). Also, application samplers and simulators are needed for studying operational and behavioural stability in practice.

7 Engineering Practices Supporting Architectural Stability (RQ3)

In this section, we discuss software engineering practices that support architectural stability, architecture analysis, design and evaluation, during the different phases of the software lifecycle.

7.1 Architecture Analysis and Design

Designing stable architectures for the evolutionary purpose is about making architectural decisions and selecting architecture styles such that evolution could be possible in the future and changes are accommodated without architectural breakdown or phase-out [147] [?]. The universal philosophy "design for change" [147] has been adopted in the early efforts for designing stable architectures [?]. This concept has been promoted as a value-maximising strategy, where the stable and evolvable architectures are expected to add value, throughout the system lifetime, that overbalances the cost of designing for change and the cost of changes as they occur [148] [?].

By that, an economic-based approach has been adopted to develop flexible architectures that will remain stable while the requirements are changing [148]. This required linking the structural decisions to the future value creation [?]. Such linking enables to evaluate the worthiness of designing for change, i.e. comparing the initial cost required to build a changeable architecture versus the expected added value if the uncertain changes occur [?]. The economic-based approach called "ArchOption" has adopted the Real Options Theory from economics to design evolving architectures as a value-maximising activity under uncertainty [?] [?] [?]. In details, the added value was attributed to "the flexibility of the architecture in enduring changes in requirements" [?]. Given the anticipated changes, reaching the design decision for a stable architecture entails searching for the architecture design that maximises the value of adapted flexibility with respect to the likely requirements changes [?] [?] [?]. Such reasoning informs the selection of a stable architecture notwithstanding future changes, and can then be used to derive insights into design and investment decisions related to architectural stability and evolution [?].

In this context, we do not ignore architecture analysis and design classical approaches in the literature. Examples of seminal works include Software Architecture Analysis Method (SAAM) [149], Architecture Tradeoff Analysis Method (ATAM) [150] [151], Cost Benefit Analysis Method (CBAM) [152] [153], behaviour analysis [154], the 4+1 view model [155], viewpoints [156], scenario-based analysis [86]. Yet, none is QA-specific; they could be tailored to consider stability.

Designing self-adaptive architectures. Architecture-based self-adaptations have been regarded as a promising approach to improve the quality of service delivered, cope with runtime changes and improve the system resilience and robustness [157] [158]. Different approaches have been discussed in the literature (e.g. [157], [159], [160]) to help designers building adaptive systems. The agreement about stability as a critical property is yet found in theoretical frameworks of designing self-adaptive architectures. The research community has not taken it forward towards implementing it in design approaches. Also, it has not been explicitly considered in designing adaptation policies [?] [?].

Discussion. The architecture design approach found in the literature have focused on the structural aspect of stability and for the evolutionary purpose only. Yet, it is not adequate for considering the other aspects and other purposes (behavioural, operational). Further, the previously mentioned design classical approaches have considered the stability in the case of classical architectural styles as design alternatives. Meanwhile, modern autonomous systems exhibit more complexities. Though some approaches could be accommodated to consider stability among the design attributes and the emerging software paradigms, yet more sophisticated extensions are required while designing such complex systems (e.g. designing adaptation policies). We will discuss possible developments for these approaches to cope with stability and modern systems in Section 8.

7.2 Architecture Evaluation for Stability

At the design phase. Evaluating stability at the design phase aims at measuring to which extent a particular architecture design is capable to accommodate future changes while remaining intact [?]. This provides the architect with better understandings for the architecture design decisions and architecture investment, by addressing the implications of having a stable architecture design, relevant cost and value [?].

Predictive approaches for evaluating architectural stability are to be used during the software development stage, to predict the threat of future changes on the architecture stability [?]. The predictive evaluation aims at supporting the process of valuing the capability of a particular architecture design relative to the future changes [?]. The outcome of such evaluation is answering the key question at the design phase: which architecture design will facilitate future changes and support the software evolution? [?].

An early survey of design-time evaluation approaches [?] indicated that the evaluation approaches focused explicitly on architecture construction and implicitly on evolution. Examples of architecture evaluation methods include Active Review for Intermediate Designs (ARID) [161], Attribute-Based Architectural Styles (ABAS) [162], Scenario-Based Architecture Reengineering [87], Quality-Attribute-Based Economic Valuation [163], and CHARMY for verifying architectural specifications [164]. These methods focused on evaluating architectural decisions in relevance to

traditional quality attributes [?]. Though they adopted the concern of accommodating changes, none of them explicitly addressed neither architecture stability along with evolution nor behavioural changes during operation [1].

ArchOptions explicitly studied evaluating architectures' stability for evolutionary purpose [?] [?]. The method was built on Real Options Theory for predicting architectural stability. This model has taken the economic perspective in the evaluation, where the design is judged based on the added value and value creation [148] [?]. The major idea of this approach is value-based reasoning about the capability of the architecture to withstand the expected evolutionary changes, i.e. the stable design is seen to add value to the system on the long-term evolution. This added value, under the stability context, can be measured by: (i) accumulated savings as long as long as the architecture accommodates changes without being broken, and (ii) benefit from reusing the architecture. The predictive results of the evaluation can be used in assessing the long-term value of architecture candidates, analysing trade-offs between them for the long-term value, and validating their opportunity for evolution [?]. Though this approach is suitable for evaluating architectures of any software paradigm, it requires further extensions to accommodate the complexity arising when evaluating the stability of adaptive and self-adaptive/ self-* architectures. Such extensions are necessary to evaluate the effect of adaptivity on stability and determine the possible adaptation strategies that will keep the architecture structurally and behaviourally stable.

In the context of aspect-oriented architectures, given the assumption that modularisation of concerns in architecture design has a direct effect on its stability, the work of Medeiros et al. [?] have determined the correlation between concerns and instability by quantifying the "Spearman Correlation Indicator". This work has also evaluated the effectiveness of concern metrics and patterns in evaluating stability [?]. Authors in [?] [?] proposed a domain-specific evaluation approach for analysing the stability of aspect-oriented architecture designs, analysing the influence of the aspect-oriented composition on the stability of multi-agent software architectures using quantitative indicators.

In the domain of adaptive architectures, stability has been considered a critical property for such type of architectures [?] [?]. But stability has not been explicitly considered when evaluating adaptation policies at design-time. Meanwhile, special attention has been given for evaluating robustness and resilience of self-adaptive architectures and their controllers. Below, we discuss representative work that partially tackled some aspects of architectural stability, and we show how they intersect with the notion of stability.

Camara et al. [165] considered resilience as the ability to "deliver a service that can justifiably be trusted when facing changes", i.e. dependability under varying external conditions (runtime and environmental changes). The architecture-based approach evaluates to what extent adaptations are resilient, by advocating the use of architectural models and simulations. The potential changes that the system can encounter during runtime (including changeloads [120] and environmental changes) are simulated, and the system responses obtained are collected and aggregated into a probabilistic model that is employed in the evaluation of system resilience. This approach was developed to be used before deployment, i.e. an offline design-time prospective approach. This work could be considered tackling behavioural stability of self-adaptive architectures, as it mainly focused on the assurance of the service provision, that is "the provision of evidence that the system satisfies its stated functional and non-functional requirements during its operation in the presence of self-adaptation" [166]. With respect to the adaptation controllers, authors in [167] [168] proposed an approach for assessing the robustness of controllers in self-adaptive systems, in order to identify their design faults.

During the operation phase. Runtime evaluation of stability is about assessing the architecture state of fulfilling runtime requirements while the software is operating. In the case of adaptive architectures, it would help in identifying adaptation actions when necessary to fulfil the changing requirements, ensuring the adaptation actions will leave the architecture stable on the long-term, and avoiding unnecessary repetitive adaptations. Runtime evaluation approaches tend to do stability evaluation while the system is operating, either on the system itself or on simulations. The results could be used either to take offline decisions (changing the architecture's structure or adaptation policies) or perform the adaptation autonomously during runtime.

Runtime stability evaluation has not been addressed explicitly in the evaluation approaches found in the literature to date. Some runtime evaluation approaches available in the literature addressed evaluating other attributes related to stability, such as dependability, resilience, reliability and robustness. Here, we also discuss representative work that partially tackled some aspects of architectural stability.

The survey presented in [105] identified the challenges and opportunities for provisioning dependable and resilient cloud-based software services. The work of Ghosh et al. [169] considered the cloud dynamics in demand and available capacity in evaluating the resilience of cloud infrastructure services by "job rejection rate" and "response delay". In [100], the impact of environmental changes on resilience was quantitatively evaluated using Exploratory Data Analysis (EDA). The works of [170] [171] focused on service-oriented architectures, and investigated their behaviour (in)stability (ability to guarantee certain response time and performance) and the (in)stability of the communication medium (*physical* aspect). But instability, here, was considered as dependability (i.e. ability to deliver justifiable trusted services). With the aim of considering, not only the environment as the only source of change, but a wider range of changeloads [120], the resilience benchmarking presented in [99] has addressed the robustness and resilience issues.

At the evolution phase. Late evaluation of architectural stability aims to understand the impact of evolutionary changes on the architecture. Such impact could be on different aspects, such as the architecture's structure or the runtime behaviour.

The work of Jazayeri [?] has considered architectural stability for the evolutionary purpose. The retrospective analysis aims at analysing how easily the evolution oc-

1. Further details about the critical relation between stability and these evaluation methods could be found in [?] [?].

curred, by examining consecutive releases of the software [?]. Such analysis is based on comparing properties of different releases next to each other, to assess if the architecture remained intact through the evolution (i.e. through the different releases of the software). The approach used simplistic metrics (e.g. software size, modules number) to summarise the software evolution and coupled these metrics with "colour visualisation" to illustrate the evolution among the consecutive releases [?]. A distinct contribution of Jazayeri work is the visualisation of design components, thus it makes understanding stability easier. But a drawback of this approach is that it appears not to be practical with the absence of dedicated tools, as it requires information about each release, where such data is not commonly maintained and analysed [?] [?].

Another retrospective approach is the metric-based approach proposed in [?]. The approach is based on extracting the architecture from the source code of different software releases. Using the extracted facets, the retrospective analysis is, then, employed to examine whether the architectural decisions remained intact across the different releases, i.e. evaluate the stability of the architecture's structure. This work focused on the source code as a feature to reflect some aspects of the architectural stability [?], which could not be considered as a comprehensive view of stability.

Stability metrics. Stability metrics have widely varied between studies according to the purpose and other aspects of stability. The majority of studies have implicitly provided metrics for stability according to the context they are studying. As an example, authors in [?] considered monetary cost, time-to-release, market-value, and interest rate relative to budget and schedule as metrics when evaluating architectural stability during design-time for the evolutionary purpose. Few studies have explicitly proposed metrics for architectural stability. Constantinou and Stamelos [?] [?] proposed two sets of metrics for measuring stability for the evolutionary and reuse purposes: (i) External Stability and Internal Stability that "capture the degree that the architectural elements remain stable to consecutive versions of the same system", (ii) External Evolution and Internal Evolution that "quantify to what extent a system evolves between consecutive versions and the degree that the newly developed elements interact with the remaining part of the system". These metrics focused on the structural stability and were extended to consider behavioural evolution (Reuse Oriented Stability (ROS) metric) and related design and code evolution (Design Complexity Increase (DCI) and Bug Fixing Rate (BFR)).

In the context of dynamic systems, the significant metric of the dynamic behaviour related to stability is "the time required, following a perturbation in the system state, to reach a new equilibrium state" [58].

In the context of adaptive architectures, stability has been considered a critical property for evaluation. Many studies have considered resilience and dependability (with partial intersect with stability notion) for evaluating adaptive architectures [172] [99]. Authors in [172] have additionally considered adaptation overhead for evaluating the effects of adaptation. The overhead covers the frequency of adaptation, the downtime for reconfiguration and resources cost, and results in thrashing behaviour, that is the continuous reconfiguration with small runtime changes.

Stability has been discussed in theoretical evaluation frameworks of self-adaptive, with special insights from Control Theory. In the descriptive evaluation model of [?], the metrics used Control Theory have been mapped to self-adaptive architectures, arguing that the architecture should be evaluated within its embedded problem-solving context, domain and goal state. Meng [?] has used the same stability definition of control theory, where the system is considered stable "if its response to a bounded input is itself bounded by a desirable range" and "the controlled variables are within allowable range to the set points". Thinking of reconfiguration as the control regime for the architecture, concerning its runtime behaviour, Meng [?] has considered stability as avoiding the thrashing behaviour. In this case, stability evaluation would also require determining whether the system is approaching its target state after reconfiguration.

Villegas et al. have taken inspirations from Control Theory further, by defining a set of adaptation properties derived from control theory properties in terms of quality attributes and metrics widely used in software engineering [?]. In this work, the authors clearly differentiated the evaluation of the managed and managing system (adaptation controller). The former is evaluated by the quality attributes as adaptation goals, and the latter is evaluated by adaptation properties of the controller. Adaptation goals are the quality of service (QoS) properties intended to be achieved by the architecture, while adaptation properties are observed and measured in the adaptation process [?]. In this context, stability is "the degree in that the adaptation process will converge toward the control objective" [?]. An unstable adaptation will repeat the action with the risk of not improving or even degrading the system to unacceptable states [?]. Stability is measured by: (i) accuracy in terms of "how close the managed system approximates to the desired state" within given tolerances, (ii) settling time that is "the time required for the adaptive system to achieve the desired state", (iii) resources overshoot which expresses "the utilisation of computational resources during the adaptation process to achieve the adaptation goal" [?].

Jiao [?] has put the concept of stability in different wording using the adaptation level, that is "how well the system satisfies the user's expectations through adjusting its behaviour or configuration to tackle the changes in the environment". Considering the environmental change, adjustment and requirement satisfaction as the key aspects of adaptation, Jiao [?] has put a mathematical measurement which involves: (i) the degree of change in the environment, (ii) the degree of adjustment of the system which "reflects how much a system adjusts its behaviours and structures", and (iii) the degree of satisfaction to meet the requirements which "implies how well the system meets the requirements".

Discussion. Assuming that the architecture is the primary guide of the system evolution [?], architectural stability evaluation approaches (for the evolutionary purpose) can be performed either during the design phase or at later stages of the system lifecycle. In the former case, the evaluation aims at predicting how the architecture design will endure the likely evolution changes, where predictive

approaches are used (e.g. [**?**] [**?**]). In the latter case, the retrospective evaluation aims at analysing how easily and smoothly the evolution occurred, by comparing successive releases of the software and checking the intactness of architectural decisions (e.g. [**?**]).

Generally, the retrospective evaluation is useful for *planned evolution*, to be used for evaluating how the next release of the software will be stable [**?**], i.e. previous evolution data extracted from the retrospective analysis can be used to identify the components most likely to change and anticipate resources required for the next release. But it is not suitable for use at the early stages of architecture design, as it is done on already existing systems [**?**]. Meanwhile, predictive evaluation is seen to be preventive, by proactively understanding the stability problem and understanding threats related to possible evolutionary changes. It is obvious that using both approaches at their appropriate phases is good practice for evaluating stability throughout the software lifecycle.

The approaches found in the literature for evaluating architectural stability were limited to the evolutionary purpose and the structural aspect. The behavioural aspect was not addressed in the design and runtime phases. Though the evaluation methods were systematic, they are human-based activities, relying on the architect experience and own judgement. Some approaches sound promising for stability evaluation of modern complex systems. But novel extensions are still required to accommodate the complexity of architectures for autonomous systems. Such complexities mainly arise from the heterogeneity and dynamism of both the software itself and the environment in which the software is operating and interacting. Approaches for evaluating architectures and adaptation policies while designing modern software systems should evaluate the effect of adaptation strategy on the architecture stability, so that the architectural decisions taken result in a robust and stable architecture.

In the context of adaptive architectures, stability has been discussed in theoretical evaluation frameworks, with few mathematical or practical measurements. Approaches for evaluation can be used during design-time or offline, i.e. not while the system is operating during runtime. The community still lacks efforts in evaluating architecture adaptation decisions during runtime to comprehensively consider structural and behavioural stability.

8 Gap Analysis and Research Directions

Based on the taxonomy (section 4) and pragmatic view of stability (Figure 2), we identify the research gaps. We discuss a new perspective for considering stability as a software property and topics for further research to advance the state-of-the-art and improve the state-of-the-practice of architectural stability.

8.1 Gap Analysis

Though the research area of software stability has received much attention and significant progress has been made, many important issues are not tackled yet. The identified gaps related to stability are:

- *Clarity of the concept.* From Section 5.1, the concept of stability has been defined in many different ways. This indicates that the concept is not fully established in the software engineering community. The lack of concept clarity could be an interpretation for less attention of certain aspects or lifecycle phases in considering stability.
- *Integration in the different software lifecycle phases.* From the analysis results of primary studies presented in Section 6, stability has not received equal attention in different lifecycle phases. Though there is a lot of work considering stability in the early design and late evolution phases, the operation phase has benefited less. Also, operational stability inter-wined with the development phase has not been yet explored.
- *Stability testing, verification and validation.* While software testing is considered part of the development phase [40], it has been used to test for the dependability of the software and certain quality requirements of interest [173] [174]. The majority of work on software testing has looked at stability as an ”after-thought” and as a measure for quality assurance and compliance for dependability requirements of interest, such as performance, load testing, usability, security and safety [174] [175]. Observations from the primary studies confirm that the topic of stability testing has received little systematic attention from the research community, evidenced by the limited number of papers retrieved on the subject, and has been used to test the product under extreme conditions. The reference to the term has been intertwined with load and performance testing in the literature [174] [175]. Another observation is the generality and relevance of the concept to quality dimensions and queries of interest. We hope that our taxonomy and results can serve as an input for steering systematic and in-depth investigations into the area of stability testing, where the dimensions and observations may help in defining specific areas of investigation and for informing the foundations of stability testing.
- *Consideration of the different aspects of software artefacts.* Stability has received good attention for the different software artefacts, with the exception of the requirements level. As such, much work has looked into the static aspects of stability, such as the architecture and design structure. Yet, stability is not only a static property for software artefacts, but it also comprised of behavioural aspects.
- *Stability metrics.* Though a valid quality attribute should be quantifiable and falsifiable, researchers adopted general metrics from software engineering according to the definition and dimensions considered. The literature lacks metrics focused explicitly on stability.
- *Benchmarking.* Benchmarking provides a generic way for characterising the response of the artefact (e.g. architecture’s behaviour) when subjected to changes, allowing the quantification of stability. Yet, the literature tends to lack of benchmarks explicitly devoted for stability. With the existence of many quality attributes inter-related with stability, stability benchmarking could understandably comprehend constructs and techniques from previous benchmark efforts, such

as dependability benchmarking [176] and resilience benchmarking [99]. It can also benefit from the structure of established benchmarks [99]. But an interesting and unanswered question is what are the components that should be added to a stability benchmarking to reflect the various aspects of stability, or a particular software domain (e.g. behavioural stability of self-adaptive architectures or real-time systems).

- *Support tools.* Though there exist systematic approaches for design and evaluation of stability (e.g. [**?**], [**?**]), the development of support tools is underpinned to a big extent. Such tools would help designers and architects to make stability approaches efficient, practical and usable. They could also help in evaluating stability during runtime.
- *Validation in an industrial context and empirical studies.* While there are many studies theoretically discussing the notion of stability and proposing solutions, empirical studies, experience and practice reports are not well-featured in the primary studies. Such studies, when applied to particular business domains, could reveal the benefits and associated challenges related to those domains. Even though there is massive data generated from the industry, there is little attention devoted to the validation of research studies in industrial context.
- *Engineering approaches considering the different aspects of architectural stability for different purposes.* From the closer analysis of the architecture level (Figure 13), we identified the lack of engineering approaches considering all architectural aspects for different purposes in the different lifecycle phases. For the development phase, studies related to architecture analysis, reasoning and design focused on either quality impact for architecture construction [177], or the provision of specific quality attributes (e.g. dependability and reliability) [178] [179] [180] [181]. Architecture evaluation methods focused on the structural aspect of architectures for evolutionary purposes. Meanwhile, architecture analysis methods should focus on predicting the different aspects of stability, as the case of architecture-level modifiability analysis (ALMA) [182]. Architecture reasoning and design should also explicitly consider stability when translating requirements to an architecture solution [183]. There is also a significant lack in considering the behavioural aspect during operation. Runtime adaptation mechanisms proposed in the literature focused on some adaptation properties, such as tactics latency (the time it takes since an adaptation is started until its effect is observed) [184], settling time (the amount of time the controller takes to achieve the adaptation goal) [185] [186]. Yet, other properties reflecting the quality of adaptation, i.e. how well the adaptation process converges towards the adaptation goal, are not explicitly considered [187] [**?**], while properties reflecting the behaviour of the controller have an impact on the stability of the whole architecture [**?**].

8.2 A New perspective

Our survey showed that the notion of stability is not clearly defined and characterised (section 5.2) and that the different dimensions of stability are not fully considered (section 6). This has hindered proper consideration of stability as a quality attribute that is strategically important for the longevity of software systems. We hence propose a new perspective based on the proposed working definition for the concept of stability (section 5.2), where we consider the different dimensions (5W+1H) of the taxonomy (section 4). The new perspective (depicted in Figure 14) addresses the following issues.

Stability definition. As mentioned earlier (in Section 5.2), a working definition of stability should be built around the taxonomy dimensions, while focusing on one ability (e.g. ability to keep unchanged) given the intended purpose (e.g. evolution, operation).

Stability analysis. As stability is not an absolute property and given its different dimensions, this requires a further case- and context-dependent analysis. Such analysis should depend on the system in question, its architecture, the variables subject of interest (e.g. behavioural attributes that should be kept stable) and the system context (i.e. contextual aspects influencing the system and its stability). By the system in question, we mean the type of application and/or the software domain that determine the associated dynamics and contextual aspects, while the architecture type determines the aspects that should be kept stable. As an example, in the case of adaptive architectures, the behaviour of the adaptation controller would be considered in the stability analysis.

Integration in the different lifecycle phases. The integration of stability in the different phases would render strategic benefits throughout the software lifetime. We argue that the realisation of stability in the different phases is complimentary. A good realisation plan should, for instance, include evaluating architecture alternatives for long-term evolution and defining runtime adaptation policies in advance, which will ease the evolution process on the long-term and render stable runtime adaptation actions. Meanwhile, putting the architecture in operation with less design-time planning for stability will degrade the architecture performance during runtime. Though stability analysis could be foundational, engineering practices should be distinct for each phase. During the design phase, a candidate architecture should be evaluated by the ability of its structure (structural stability) to maintain fulfilling the functional and behavioural requirements that are known at this stage, as well as the likely changes to occur in the future when put into operation (functional and behavioural stability). While the software is operating, the architecture ability to fulfil the changing requirements and workloads (behavioural requirements) should be continuously assessed during runtime. The evaluation of architecture alternatives during the design phase is evidently different from retrospective evaluation for evolution planning. Runtime evaluation approaches can vary between online and offline techniques.

Requirements Engineering for stability As architectures typically play a key role in achieving quality requirements [149] [51] [188] [189] and guiding the software evolution [190] [**?**], we can evidently agree that realising stability at the design and architecture levels should be based on the quality requirements subject to stability [149] [189] [191], where requirements are the key to long-term stability

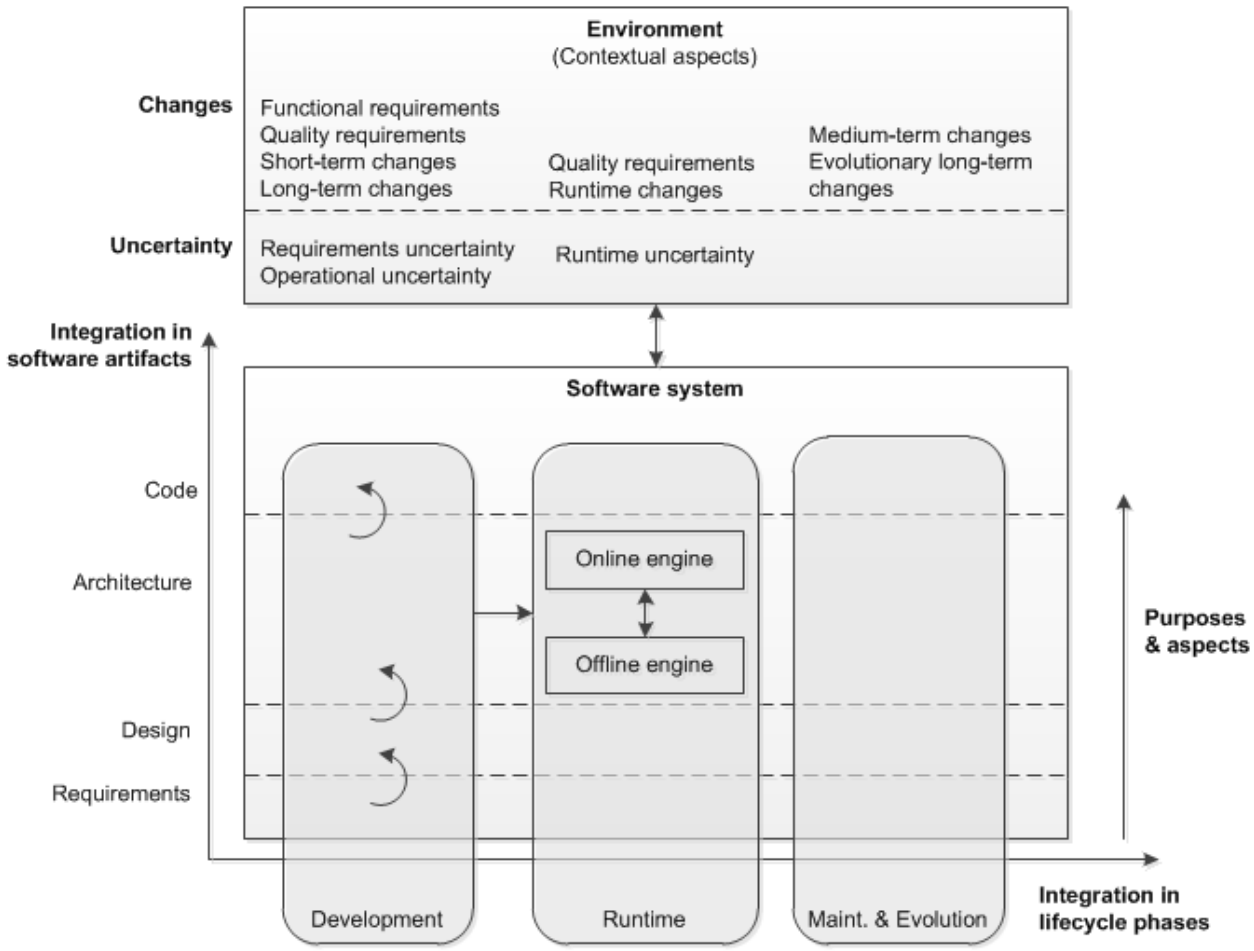

Figure 14: A new perspective for Software Stability.

and sustainability [135] [192]. In other words, the outward requirements goal is concerned with what the system will accomplish for its end-users [49], which will be achieved by the architecture.

Having a stable architecture design should start in an earlier stage prior the design, i.e. in the requirements engineering phase [44], where quality requirements are assessed throughout the architecture's lifespan and will be used in informing architecture decisions, so that the architecture will not break-down easily when coping with increased runtime load demands or evolution [193] [**?**]. Hence, a "behaviourally stable" architecture design should be based on the requirements subject to stability. Requirements engineering for stability will help in capturing and analysing the quality attributes subject to stability while building stable architectures. For instance, if performance is a significant requirement, the architecture should be designed in a way that critical operations are localised within a small number of components deployed on the same computer which reduces the number of component communications if distributed [51]. If availability is a key requirement, the architecture design should include redundant components, so that replacing and updating components is possible without stopping the system [51]. Such requirements subject to stability should be modelled as goals at an abstract level, then technically fine-grained to be allocated to single specific components [194] [195]. Explicit relation between the requirements model and the architecture should also be present to consider the architectural stability [196] [193] [197] [**?**].

Runtime Requirements Models, denoting requirements models that are used at runtime, have the main role in monitoring the satisfaction of requirements during runtime [198]. They should explicitly consider stability attributes, their dynamic traces to architecture components, and the historical information related to the fulfilment of these attributes. The link between the requirements and architecture during runtime should be explicit and symbiotic. From the requirements side, if the architecture will change/adapt in light of the changing requirements, this will ensure fulfilling the changing runtime requirements. From the architecture side, this will ensure that the architecture will have the expected behaviour, avoiding performance degradation and phasing-out.

Engineering approaches coping with new software paradigms. The emergence of new software paradigms, such as self-adaptive [159], self-* [199] [200] and self-aware [201] [202], necessitates amending the current practices of

requirements engineering and architecture design [203], as well as brings the need for novel methods and techniques that explicitly consider the architectural stability of such complex systems and associated trade-offs. Engineering practices shall also address the challenges that the self-* properties and the operating environment impose, along with modelling the future changes and the evolution to take the architectural decision that maximises the value of the selected architecture style relative to the evolution (i.e. that is stable). As an example, the stability of the adaptation controller in self-adaptive systems should be considered when designing the adaptation controller and policies.

Contextual aspects influencing architectural stability. Dealing with stability should be associated with the contextual aspects of the system, which should be tackled in the different engineering practices during design-time and runtime. This includes changes and uncertainty. Practices should be moving towards a new era, where architectures are considered in the environment of unpredictability. Designing and evaluating under the unknown should benefit from the information value [204] to evaluate the risk, value perception and quantify the unpredictability.

Changes have been classified by timing: (i) short-term, dynamic changes in the system or requirements, (ii) medium-term, reconfigurations for maintenance, and (iii) long-term, radical changes, reconfigurations and reorganisations for evolution [98]. Changes differ also in their nature, they could be functional changes, quality requirements changes, operational (changing behaviour of a service component when sharing resources) and technological (either software or hardware) [98]. Different types of changes are affecting the architectural stability at the different time phases and should be handled. For instance, architects should consider the long-term evolutionary changes when designing architectures for stability. In designing adaptive architectures, it is important to capture the possible changes that will drive adaptations [14]. Dealing with the operational and behavioural aspects of stability, architects should also cater to the runtime changes in user and quality requirements. Evaluating adaptation decisions during runtime would require estimating possible future changes, in order to avoid unnecessary frequent adaptations.

Modern complex systems exhibit uncertainty from many sources, arising from the heterogeneity and dynamism of both the system itself and the operating environment [159] [199] [200]. Runtime uncertainty is associated with changes in workload [120], runtime requirements [205] [206] and the nature of the software paradigm. As an example, considering the cloud paradigm, that offers pay-per-use service for the end-users, the architecture exhibits a high degree of uncertainty in the workload received from different users with different SLAs. In the case of adaptive and self-adaptive software, added the uncertainty arising from the effect of the adaptation actions [207] [208]. Although major advances have been made for handling uncertainty, existing works do not systematically address the stability of the architecture notwithstanding uncertainty. Requirements engineering should consider the uncertainty associated with the requirements subject to stability according to the stability purpose, which will be passed to the architecture design phase. Evaluating architectures (and their adaptations) during runtime should consider how stability will be affected by any form of uncertainty. This could be done either online or offline. Like other quality attributes (e.g. reliability, robustness and resilience) [209], sensitivity analysis for parameters of the probabilistic quality models is needed for the stability of these attributes. Online evaluation approaches should consider the stability of the architecture decisions and relate their evaluation results to the online autonomous architectural decisions.

8.3 Research Directions

The new perspective for stability as a software property integrated throughout the lifecycle, along with the analysis of the current research status, reveal some recommendations that can help to direct future research efforts in the community. A description of these challenges is presented as follows (summarised in Figure 15). We differentiate between challenges related to design-time and pre-deployment of the architecture, runtime while the architecture is under operation, and support tools.

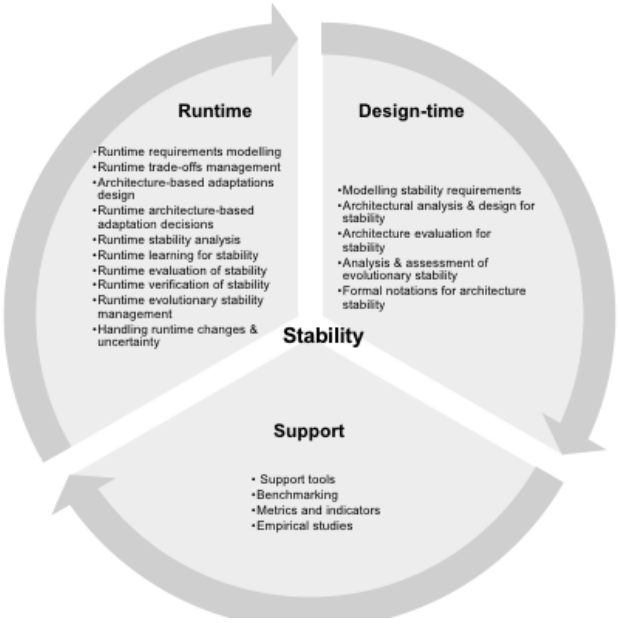

Figure 15: Challenges for Architectural Stability.

8.3.1 Design-time Challenges

- *Modelling stability requirements.* As advocated in the new perspective for stability, requirements are the starting point for long-term stability and sustainability [135] [192]. Requirements engineering for stability has reflected some challenges that still need to be addressed. The first issue is extending requirements models to explicitly consider modelling stability requirements and their trace to architecture artefacts, as the case of scalability requirements [143]. The second issue is predicting and modelling the changes in stability requirements that the systems are likely to experience during their lifetime [**?**], which requires a systematic way to predict the changes, quantify their likelihood [**?**] and their impact on the architecture. The third issue is the

traceability of stability requirements to the architecture components. Designing for stability needs traceability (forward and backward) techniques to trace and model dependencies from the requirements and their likely changes to the architecture design [**?**]. The forward dependencies shall demonstrate which architectural element(s) is responsible for satisfying a specific requirement. The backward dependencies shall demonstrate which requirement(s) are related to an architectural element. Modelling such dependencies allows managing the change across software artefacts. Traceability is important for managing the changes of requirements and the evolution of the architecture [193] [14], which will help in attaining architectural stability. The ideas of assessing the quality requirements throughout the architecture's lifespan and the traceability of requirements to architectures have been promoted in [193] [196] [**?**]. But modelling stability requirements has not been approached yet. Stability requirements should be differentiated to reflect the quality attributes essential for end-users to be kept stable without violations (e.g. response time for real-time systems). Such requirements should be modelled taking into consideration both the short-term and long-term impact of the changes in these requirements so that the architecture will not breakdown easily when coping with increasing load demands [193] [**?**]. The stability ranges for these requirements will, then, be used to better inform architecture design decision when selecting the architecture style and component technologies to induce the selected style. These open issues become more relevant with the emergence of more complex systems, the wide mode of uses, and the higher degrees of uncertainty that the system will encounter in the future once in operation. The use of Goal-Oriented Requirements Engineering (GORE) [210] [211] could be a feasible approach for elicitation, modelling, analysis and reasoning of stability requirements, as goals are defined at an abstract level, then technically fine-grained and allocated to single specific components, allowing backward traceability from low-level technical prescriptions to high-level objectives [194] [195].

- *Designing for architectural stability.* Architecture design requires novel approaches for guiding the architectural decisions and exploring architecture solution space, where this guidance should explicitly consider architectural stability [212] [213]. Approaches should also consider the uncertainty of the future that poses a considerable challenge when designing architectures. Design decisions can be articulated in terms of utility and risk. For instance, structural stability needs to be linked to the utility for designing intact architecture structure. This calls for systematic approaches to managing, handling and rectifying uncertainty to achieve long-term stability. We envision that architectural decisions at design-time should be seen as planning for stability. A possible interesting design approach could be adopting the "transplantation" process. This process, likened to organ transplantation in humans, has been explored at the code level [214]. When performing transplantation from one piece of software to another, a feature's code (organ) in a program (donor) is isolated and transplanted to the software lacking it (vein) [214]. It could be interesting to explore how transplantation of architectural decisions can be possible, in order to improve stability and how the features that had the promise in providing stable states could be adopted.
- *Architecture evaluation for stability.* While there exist notable efforts in the literature for architecture evaluation, yet there is still need for systematic evaluation approaches that explicitly consider stability, as the case of modifiability [182] and scalability [215] [216]. We call for extending the evaluation approaches available in the literature that addressed other attributes related to stability. Also, approaches that partially considered certain aspects of stability could be extended to cover stability evaluation in more depth. With respect to related contextual aspects, design-time evaluation should also address and anticipate the uncertainties arising from the future changes that the architecture might face [159] [204], their likelihood and their expected effect on the architecture. These should be quantified from the architecture perspective to be suitable for use in stability evaluation. Though there exist many attempts in the literature in addressing different facets of uncertainty, there is still need for novel approaches to quantify and rectify uncertainty from the architecture perspective, so that these approaches would be suitable to be used when evaluating the long-term stability of the architecture. As part of stability evaluation, risk assessment needs to be performed, as such risk associated with the architecture in the shed of different uncertainty factors is analysed and stability is, then, explicitly evaluated [204]. Such evaluation calls for novel approaches with the capability of evaluating under the unknown.
- *Analysis and assessment of evolutionary stability.* Architecting for long-term stability (i.e. evolutionary stability) requires evidently analysing and assessing potential evolutions, by assessing the ranges of changes in stability requirements, elucidated and known during design-time [14]. This includes assessing the timing of likely changes, the long-term cost of materialising these changes, and the long-term value of the architecture capability in enduring these changes [**?**]. Evolution assessment can make use of several existing techniques, including the use of emerging implied scenarios and technology roadmaps. Scenario-based techniques can employ several types of scenarios in the assessment process of evolution, such as anticipatory scenarios and exploratory scenarios [217] [109] [7]. But these techniques are human-centric, thus their effectiveness tends to be sensitive to human's expertise, and previous knowledge of the domain. Alternatively, the architecture can be simulated, where scenarios can be inferred through analysing execution traces to learn more about emerging requirements that may call for change and drive evolution [7]. Long-term evolutionary changes, such as moving to a new paradigm or operating environment, can make use of evolutionary paths [7]. As a fact, treatment of evolution assessment and related long-term changes can differ across different domains. Change impact analysis is also useful to perform what-

if-analysis for architectural analysis, i.e. which component or requirement will be affected by an architectural change [218] [219]. Change impact analysis might also include: (i) assessment of the cost-effectiveness of the design for change, where the upfront costs incurred from "designing for change" (to include flexibility in the architecture design relative to the likely changes) are traded-off against the long-term benefits, and (ii) assessing the cost-effectiveness of the architecture change, where a trade-off analysis is undertaken between the decision of leaving the architecture structure intact and changing the architecture to accommodate future changes. Architecture change impact analysis should also be accompanied by automated reasoning tools for handling changes and guiding the architecture evolution [**?**], where an effective change impact analysis would assist in taking design decisions leading to stable architectures.

- *Formal notations for architecture stability.* Architecture Description Languages (ADLs) range from formal to semi-formal notations for representing and analysing architectural designs [**?**] [220], i.e. provide a conceptual framework and concrete syntax for modelling and characterising the conceptual architecture of a software system [**?**] [221]. ADLs, when introduced, had potential in supporting architectural analysis and evaluation [190] [**?**] [220]. For instance, using ADLs specifications, one can formally analyse for properties on interest affecting some qualities like security, performance, to compare properties of different releases of a software and give a better understanding how the changes are expected to affect the stability in alternative solutions [**?**]. This retrospective analysis could serve in planning future releases with better stability. Research efforts need to be directed towards exploring the role of ADLs in evaluating architectural stability.

8.3.2 Runtime Challenges

- *Runtime requirements modelling for stability.* Modern software systems are operating with continuously changing requirements during runtime. Managing requirements during runtime is evidently becoming an important matter [205] [198]. Certain software paradigms impose more challenges from their nature. For instance, cloud-based software needs to handle emerging requirements as a result of operating in dynamic, open and uncertain contexts. With the advances and complexity of systems, these requirements might also include requirements from the environment where the system is operating, as in the case of cloud federations [222]. Though there has been growing research in requirements engineering handling runtime requirements, authors are not aware of any work tackling the runtime behavioural stability problem from requirements engineering perspective. Meanwhile, runtime requirements models should explicitly consider stability requirements, their dynamic traces to architecture components, and the historical information related to their fulfilment. The link between the requirements and architecture during runtime should be explicit and symbiotic, where the traceability links between requirements and architectures should be kept updated. From the requirements side, if the architecture adapts in light of the changing requirements, this will ensure fulfilling the changing runtime requirements. From the architecture side, the adaptation takes place according to the changing requirements, which will ensure that the architecture will keep its intended behaviour. Runtime Goal Models, extending design-time goals with further data about the runtime fulfilment of goals [198] [223], could be a promising approach for handling stability requirements during runtime.
- *Runtime trade-offs management.* Having architectures that efficiently manage trade-offs between multiple quality attributes is becoming a pressing need with the advancements of different software paradigms [224] [225]. Achieving such good trade-offs is challenging, due to the complexity of the imposed trade-offs, arising from the conflicts that might appear between different stability requirements and the consideration of multiple dimensions of stability. The architecture type might also impose trade-offs, as the case of self-adaptive architectures (i.e. adapting to achieve quality requirements vs. frequency of adaptations that might cause instability) [**?**]. The recently emerged self-aware class of software also exhibits complex trade-offs across the different levels of self-awareness (i.e., goal-, time-, and interaction-awareness) [226] [201] [227] [228]. Thus, we call for novel approaches for managing trade-offs during runtime that result in fulfilling multiple qualities and sustaining behavioural stability.
- *Designing stable architecture-based adaptations.* Architecture-based adaptations employ architectural models for designing software with robust behaviour and accommodating runtime changes [229] [230]. Adaptations strategies and policies are defined by the architect/designer at the development phase, and enactment decisions are taken autonomously during runtime. Among the wide literature on autonomic computing, there are studies that tackled designing robust self-adaptive architectures, as robustness is considered as the ability to recover (return to equilibrium state) when perturbed by any kind of problems, which is intersecting with the notion of physical stability (e.g. [231] [232]). Efforts for designing adaptation controllers also need to be directed towards considering properties reflecting the behaviour of the controller, which have an impact on the stability of the whole architecture [**?**], as the upper limit of the performance of the architecture is often set by stability considerations [61]. Controllers design could be inspired from stability studies in the Control Theory discipline [60] [61] —that has been widely used to incorporate self-adaptive capabilities into software systems [233].
- *Stable runtime architecture-based adaptation decisions.* Stability has been defined as one of the properties reflecting the quality of adaptation, i.e. how well the adaptation process converges towards the adaptation objective, but not explicitly considered by adaptation mechanisms [187] [**?**]. Adaptation mechanisms proposed in the literature focused on some adaptation

properties, such as tactics latency (the time it takes since an adaptation is started until its effect is observed) [184], settling time (the amount of time the controller takes to achieve the adaptation goal) [185] [186]. Meanwhile, runtime adaptations, if engineered with stability in mind, can render benefits towards architectures intended behaviour. Runtime decisions should be seen as continuous realisation and assessment of behavioural stability. More challenges are imposed from the nature of the architecture, as in the case of self-adaptive architectures where the continuous runtime adaptation might cause architecture instability. As computational intelligence has proven to be promising to enable intelligent behaviour in adaptive systems [234], its paradigms (e.g. evolutionary computation, swarm intelligence) are promising to be applied for stability problem. Self-stabilisation of distributed systems [66] [67] [70] could also be employed for guaranteeing requirements satisfaction and ensuring a stable behaviour in a finite time following workload perturbations.

- *Runtime stability analysis.* Runtime adaptation decisions, engineered with stability in mind, call for novel approaches that can complement design-time with runtime analysis for stability. Practical approaches are needed to: (i) assess if an architecture will maintain its stability at runtime in spite of the unexpected changes in requirements and the environment, and (ii) evaluate alternative adaptations for retaining the stability of the architecture. Such approaches should not cause extra unnecessary overhead, i.e. it should rely on self-awareness capabilities to run only when necessary. Runtime analysis for stability can be performed in different modes: either offline, online or symbiotic simulators. Execution can be mined, analysed and/or visualised offline to consider areas which are likely to cause instability by examining the behavioural and/or structural aspects of the architecture. Live-logs of execution traces can also assist in runtime decisions. Verifying and visualising execution traces can, for example, benefit from the Labelled Transition System Analyser (LTSA) tool, which performs exhaustive searches for any possible violations of the properties required to be achieved by the system, and provides animation for the system specification for interactive exploration of the system behaviour [235]. Analysing live-logs can also benefit from mining software repositories techniques that are performed for software evolution [236] [237].
- *Runtime learning for stability.* The consideration of stability in runtime decisions should be complemented with online learning approaches and runtime dynamic impact analysis [238]. Such learning approaches are capable to use historical monitoring data about the fulfilment of requirements and the stable states of the architecture to predict the stability state prior to performing changes in the architecture or its configurations. Such further advancement would lead to a more stable architecture that would sustain for longer. Stability analysis can also employ machine learning techniques to mine execution logs and predict areas that require improvements for stabilising future runs.
- *Runtime evaluation for stability.* Modern software systems, relying on runtime adaptations, require runtime evaluation approaches during operation that explicitly consider how the behaviour is stable during runtime. This requires a continuous assessment prior to and after taking the adaptation action. The pre-assessment aims to evaluate its effect on the current state and its expected effect on the future state given expected workloads. The post-assessment aims to evaluate the actual effect of the adaptation action to ensure the fulfilment of runtime requirements and call for further adaptation actions if necessary. Stability assessment for corrective, preventive, or adaptive changes may require different method for the likelihood, magnitude and significance of the change [14]. At runtime, architectures can be simulated for testing the continuous fulfilment of stability requirements. It would be possible to draw inspirations from models@runtime to infer requirements that may need to be reflected on the system and consequently lead to further refinement and adaptation [239] [240].
- *Runtime verification for stability.* Runtime verification is "the process of evaluating, while the system operates, whether it meets certain expected behaviour and goals" [241] [242] [243]. Such quantitative verification is essential for self-* systems [244] [245] [246]. Given the uncertain operating environment of these systems, probabilistic model checking could be employed for continuous assurances of the intended behaviour [247].
- *Runtime evolution management for stability.* In situations where the costs and risks associated with shutting down and updating the system, runtime evolution in unavoidable [248] [160] [98]. In such case, evolutionary stability is becoming an important topic to be considered when performing runtime modifications in the architecture.
- *Management of runtime changes and uncertainty.* In considering runtime stability, changes in workload and runtime requirements should be considered [205] [120]. As the architecture performs adaptations during runtime, the runtime decisions should take into consideration the possible future changes, not only current changes, which will decrease the frequency of adaptations that might cause architectural instability [**?**]. Associated with the runtime changes is the uncertainty of these changes [165]. With the increasing complexity and heterogeneity of software systems, the uncertainty arising from the environment where the architecture is operating should also be considered [159]. The case of adaptive and self-adaptive software adds another facet of uncertainty that is the uncertainty arising from the effect of adaptation actions, that should be considered explicitly.

8.3.3 Support-related Challenges

- *Support tools.* Stability-related support tools give the opportunity for practical application and validation of the solutions developed in the research community. This could be approached by either developing new tools that are stability-specific, or by extending existing tools to support stability.
- *Benchmarking.* A stability benchmark should provide a generic way for characterising the different aspects

(structure, behaviour) of the architecture when subjected to changes, allowing for the quantification of stability. Stability benchmarking shall understandably comprehend constructs and techniques from previous benchmark efforts, such as performance, dependability, and resilience [176] [99] [40], based on their interlink with the stability concept. A stability benchmark can benefit from the structure of established benchmarks [99], but should additionally consider the various aspects of stability. A benchmark specifically targeted to a particular domain is also another good practice. For instance, benchmarking the stability of self-adaptive architectures requires evaluating the effectiveness of self-adaptation and its impact on quality of service (behavioural stability).

- *Metrics and indicators.* Qualitative indicators might be a good practice for design-time, where the experts' judgements can be considered when differentiating between candidate architectures. But quantitative metrics would be a good practice to overcome the subjectivity of the experts' judgements and the time required to consolidate their judgements [7] [103]. We suggest that metrics for stability would consider metrics from other inter-related attributes as a base, such as resilience, dependability and maintainability (e.g. [40] [100] [249]). Metrics should also consider the different dimensions of stability (structural and behavioural). For instance, metrics for behavioural stability would measure to what extent a candidate architecture would satisfy the quality requirements and would keep satisfying them when subjected to changes.
  Runtime evaluation of architectural stability calls for rapid feedback regarding the stability of the architecture during operation. It is evident that qualitative indicators are not suitable for runtime, unless offline decisions are required for a significant change in the architecture [7] [103]. Quantitative metrics would be the practical case for continuous evaluation the architecture while the system is operating at runtime [250]. Specific metrics for certain software paradigms is another good practice. For instance, metrics for self-adaptive systems would consider the quality, cost, overhead and frequency of adaptation action [251]. Such metrics could be inspired by the ecosystems, as these systems have originally been founded on the same concept.
- *Empirical studies.* As noted in the gap analysis, the literature lacks to a big extent empirical studies documenting the stable states of designed architectures, i.e. the extent to which architectures had succeeded or failed in attaining their structure and objectives [**?**] [252]. This calls for systematic empirical studies to analyse real-life cases, where there was an architecture breakage upon accommodating changes. Such case studies shall improve the state-of-the-art by learning from the state-of-practice, as lessons learnt from these studies will improve engineering practices towards stability.

9 Threats to Validity

According to the classification of [253] and [254], we identify the potential threats to the validity of our study as follows:

- **Construct validity:** relates to sources investigated and data collection. This includes:
  - *Missing relevant studies.* One of the main threats of this review is incompleteness. The search was based on meta-data (abstract, title, and keywords) only, and might have missed some relevant studies that considered stability as part of their proposed work and have not mentioned this explicitly in their titles, abstract and keywords. Though the meta-data are specified by the authors of the papers, we reasonably rely on how well the digital databases classify and index papers. We have used multiple data sources, that are basically academic indexing services. These are considered as the largest and most complete scientific databases for conducting literature reviews [76] [255] and most relevant electronic databases to computer science and software engineering [256]. The search strings were carefully tried and verified. We also used the cross-referencing to find potentially relevant studies.
  - *Primary studies selection bias.* In the selection of primary studies, we cannot guarantee that all relevant studies were selected, some relevant publications might have been excluded. The biased selection might be related to subjectivity in finding primary studies, as the selection was conducted by one researcher. To avoid selection bias, we defined the purpose of the study and the research questions in advance and adopted a guided selection process under the supervision of the other two researchers. Defining clear inclusion and exclusion criteria helps in mitigating this threat.
  - *Inaccuracy in extracted data.* As data extraction was conducted by one researcher, inaccuracy can be introduced in the process due to different reasons, such as the background of the researcher and the researcher's subjectivity. The way the authors' studies used to present their approaches and findings might also be a factor. Though it was necessary to fulfil the targeted schedule, data extracted by the main researcher was validated by the second author, which lead us to believe that the effect of this error is minimal.
- **Internal validity:** concerns with the methods used in the study and related conclusions. The following could threaten internal validity:
  - *Scope of the review.* The research questions defined might not have covered the whole research area. More specifically, RQ3 focused on the engineering practices for the architecture. This implies that one may not find relevant information in the review if concerned with other artefacts, such as software design. For this issue, we had several discussions to refine the research questions and decided to focus on the architecture, for it plays an important role during software operation, maintenance and evolution. Yet, our review could be used as a base to conduct similar studies for the other software artefacts.
  - *Completeness of the review.* As the review is mainly focused on a quality attribute that is related to different software artefacts, it is hard to identify set primary studies to be included in the review for completeness.

We acknowledge that the included primary studies might not cover the entire research area. In discussing the engineering practices that support architectural stability (Section 7), when we did not find studies that explicitly considered stability, but found studies considered attributes related to stability, or partially discussing aspects of stability, we presented seminal and/or representative work. We do not claim completeness in this part. But the review is mainly based on the concepts and research questions defined earlier. Though we argue that the search strategy (defined in Section A.2) ensure that the selected primary studies constitute a good representative of the research done in the software engineering community. The set of concepts and taxonomy proposed shall help in mitigating this threat in the future.

  - *Robustness of the taxonomy.* An important threat is whether the constructed taxonomy is robust and comprehensive for the analysis and classification. First, we believe that we constructed the taxonomy in a plausible and systematic way, using the widely-adopted 5W+1H pattern (What, Where, When, Why, Who and How) [81] [82] [83] [84]. After formulating the initial taxonomy, we used an iterative approach to continuously refine taxonomy when new concepts are extracted from primary studies (as indicated in Figure A.1).

- **External validity:** concerns the generalisation and applicability of the study's findings. This includes:
  - *Publication bias.* The scope of our review is the academic research domain. The threat is that relevant engineering practices in the industry are not included, if not reported in academic publications. Nevertheless, we consider this review as a starting foundation, and we will complement it with an industrial study as future work.
  - *Validation and evaluation of primary studies.* In the review protocol, we did not validate or evaluate the research reported in the primary studies. However, the quality of data sources used, where publications from peer-reviewed conferences and journals only are published, have a direct impact on the quality of the research reported in the primary studies, and thus our review.
- **Conclusion validity:** concerns with the degree to which the conclusions drawn from data extracted are reasonable and valid. We derived our conclusions based on logical reasoning and sound analysis of the primary studies. Further, all the conclusions were drawn by the three researchers and double-checked against related studies. We have also been careful in not making conclusions based on a single study. Discussions and conclusions are related to the whole research area. Most importantly, the review protocol specification (details in Appendix A) makes it possible to replicate the study. But the selection and data extraction, based on reading the whole papers, is subjective and might lead to different selection, classification and findings. Yet, this includes research creativity and forms part of the research contribution.

10 Conclusion

Stakeholders and organisations are increasingly looking for software longevity, as such stability could be considered a primary criterion towards achieving it. In this manuscript, we reviewed the state-of-the-art related to stability as a software property, with a special focus on software architectures. We proposed a taxonomy for characterising the concept, analysed definitions found in the literature, and proposed a working definition. Such characterisation paves the way for better understanding of the concept, and consequently motivate research. We discussed how stability was treated for the different software artefacts by the software engineering community. As architectures have a profound effect throughout the software lifetime, we closely reviewed the related engineering practices.

Stability could be envisioned as the next step in quality attributes, combining many related qualities and aspects. Consequently, research and practice shall witness a growing attention to stability. Although it will take considerable time and effort to achieve a comprehensive framework for measuring, evaluating and achieving stability, the surveyed literature indicates that future developments in requirements engineering, architecture design and evaluation may align towards architectural stability, since researchers and practitioners aim for better quality and long-living software.

The review indicates a shift from a narrow concept of the stability (architecture intactness) to a multi-dimensional concept, including many aspects (structural, behavioural). We hope this survey serves as a primary investigator for deeply characterising architectural stability, to take it further towards handling the wider concept and the related challenges. As future work, we plan to complement this study with another review in an industrial setting, to cover related state-of-practice.

References

[1] R. Capilla, E. Y. Nakagawa, U. Zdun, and C. Carrillo, "Toward architecture knowledge sustainability: Extending system longevity," *IEEE Software*, vol. 34, no. 2, pp. 108–111, 2017.

[2] M. Mattsson, H. Grahn, and F. Mårtensson, "Software architecture evaluation methods for performance, maintainability, testability, and portability," in *2nd International Conference on the Quality of Software Architectures*, 2006.

[3] F. Febrero, C. Calero, and M. A. Moraga, "Software reliability modeling based on ISO/IEC SQuaRE," *Information and Software Technology*, vol. 70, pp. 18–29, 2016.

[4] C. Becker, "Sustainability and longevity: Two sides of the same quality?" in *3rd International Workshop on Requirements Engineering for Sustainable Systems co-located with 22nd International Conference on Requirements Engineering (RE)*, 2014.

[5] A. Avritzer, A. Bondi, and E. J. Weyuker, "Ensuring stable performance for systems that degrade," in *5th International Workshop on Software and Performance*. ACM, 2005, pp. 43–51.

[6] V. T. Rajlich and K. H. Bennett, "A staged model for the software life cycle," *Computer*, vol. 33, no. 7, pp. 66–71, 2000.

[7] H. P. Breivold, I. Crnkovicb, and M. Larsson, "A systematic review of software architecture evolution research," *Journal Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.

[8] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Conference on The Future of Software Engineering*. ACM, 2000, pp. 73–87.

[9] W. B. Frakes and K. Kyo, "Software reuse research: Status and future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[10] H. Ji, "Dynamic and static views of software evolution," in *IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2001, p. 592.

[11] C. Hollenbach, R. Young, A. Pflugrad, and D. Smith, "Combining quality and software improvement," *Communications of the ACM*, vol. 40, no. 6, pp. 41–45, 1997.

[12] A. Gurgel, F. Dantas, A. Garcia, and C. Sant'Anna, "Integrating software product lines: A study of reuse versus stability," in *IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 89–98.

[13] J. Bosch, "Architecture challenges for software ecosystems," in *4th European Conference on Software Architecture: Companion Volume*. ACM, 2010, pp. 93–95.

[14] B. Williams and J. Carver, "Characterizing software architecture changes: A systematic review," *Journal Information and Software Technology*, vol. 52, no. 1, pp. 31–51, 2010.

[15] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Press, 2012, pp. 99–108.

[16] B. Penzenstadler, V. Bauer, C. Calero, and X. Franch, "Sustainability in software engineering: A systematic literature review," in *16th International Conference on Evaluation & Assessment in Software Engineering (EASE)*, 2012, pp. 32–41.

[17] B. Penzenstadler, A. Raturi, D. Richardson, C. Calero, H. Femmer, and X. Franch, "Systematic mapping study on software engineering for sustainability (se4s)," in *18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, pp. 1–14.

[18] N. J. E. Wolfram, P. Lago, and F. Osborne, "Sustainability in software engineering," VU University Amsterdam, Report, 2017. [Online]. Available: https://research.vu.nl/en/publications/c0b3f46f-58d0-400a-a345-8b55e53a72a1

[19] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 367–377.

[20] F. Febrero, C. Calero, and M. A. Moraga, "A systematic mapping study of software reliability modeling," *Journal Information and Software Technology*, vol. 56, no. 8, pp. 839–849, 2014.

[21] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[22] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, 1993.

[23] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, pp. 1–34, 2014.

[24] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl, "A framework for classifying and comparing architecture-centric software evolution research," in *17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 305–314.

[25] M. Shaw and P. Clements, "The golden age of software architectures: A comprehensive survey," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU-ISRI-06-101, 2006.

[26] M. A. Babar and I. Gorton, "Software architecture review: The state of practice," *Computer*, vol. 42, no. 7, pp. 26–32, 2009.

[27] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[28] A. Tang, M. A. Babar, I. Gorton, and J. Han, "A survey of architecture design rationale," *Journal of Systems and Software*, vol. 79, no. 12, pp. 1792–1804, 2006.

[29] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design: A comparative survey," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–28, 2011.

[30] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.

[31] A. Patidar and U. Suman, "A survey on software architecture evaluation methods," in *2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, pp. 967–972.

[32] S. Mahdavi-Hezavehi, V. H. S. Durelli, D. Weyns, and P. Avgeriou, "A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems," *Information and Software Technology*, vol. 90, no. Supplement C, pp. 1–26, 2017.

[33] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[34] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, 2005.

[35] M. Riaz, M. Sulayman, and H. Naqvi, "Architectural decay during continuous software evolution and impact of 'design for change' on software architecture," in *Advances in Software Engineering: International Conference on Advanced Software Engineering and Its Applications, ASEA 2009 Held as Part of the Future Generation Information Technology Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings*, D. Slezak, T. Kim, A. Kiumi, T. Jiang, J. Verner, and S. Abrahão, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 119–126.

[36] A. Immonen and E. Niemela, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Journal Software and Systems Modeling*, vol. 7, no. 1, pp. 49–65, 2008.

[37] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 367–377.

[38] H. Koziolek, "Sustainability evaluation of software architectures: a systematic review." ACM, 2011, Conference Paper, pp. 3–12.

[39] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC/IEEE 24765:2010(E) Systems and Software Engineering – Vocabulary," Geneva, Switzerland, Report, 2010.

[40] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[41] Software Engineering Standards Committee of the IEEE Computer Society, "Ieee standard for a software quality metrics methodology," The Institute of Electrical and Electronics Engineers, Inc., Report IEEE Std 1061-1998, 1998.

[42] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[43] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[44] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[45] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[46] C. Seo, G. Edwards, S. Malek, and N. Medvidovic, "A framework for estimating the impact of a distributed software system's architectural style on its energy consumption," in *7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2008, pp. 277–280.

[47] J. S. Kim and D. Garlan, "Analyzing architectural styles," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1216–1235, 2010.

[48] S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[49] B. I. Witt, *Software architecture and design: Principles, models, and methods*. New York: New York: Van Nostrand Reinhold, 1994.

[50] H. Gomaa, *Software modeling and design: UML, use cases, architecture, and patterns*. Cambridge: Cambridge University Press, 2010.

[51] I. Sommerville, *Software engineering*. Boston, Mass. London: Boston, Mass. London : Pearson Education, 2011.

[52] C. Lianping, M. A. Babar, and B. Nuseibeh, "Characterizing architecturally significant requirements," *IEEE Software*, vol. 30, no. 2, pp. 38–45, 2013.

[53] "Oxford english dictionary online." [Online]. Available: http://www.oed.com/

[54] K. S. McCann, "The diversity-stability debate," *Nature*, vol. 405, no. 6783, pp. 228–233, 2000.

[55] A. R. Ives and S. R. Carpenter, "Stability and diversity of ecosystems," *Science*, vol. 317, no. 5834, p. 58, 2007.

[56] G. W. Rowe, I. F. Harvey, and S. F. Hubbard, "The essential properties of evolutionary stability," *Journal of Theoretical Biology*, vol. 115, no. 2, pp. 269–285, 1985.

[57] iSixSigma, "iSixSigma." [Online]. Available: https://www.isixsigma.com

[58] T. L. Casavant and J. G. Kuhl, "Effects of response and stability on scheduling in distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1578–1588, 1988.

[59] N. P. Bhatia and G. P. Szegö, *Stability theory of dynamical systems*. Springer Science & Business Media, 2002.

[60] A. M. Lyapunov, *The general problem of the stability of motion*. London: Taylor & Francis, 1992.

[61] J. R. Leigh, *Control theory*, 2nd ed., ser. IEE Control Engineering Series. London, UK: Institution of Electrical Engineers, 2004, vol. 64.

[62] P. Prabhakar and M. G. Soto, "Foundations of quantitative predicate abstraction for stability analysis of hybrid systems," in *Verification, Model Checking, and Abstract Interpretation: 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015, Proceedings*, D. D'Souza, A. Lal, and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 318–335.

[63] ——, "Hybridization for stability analysis of switched linear systems," in *19th International Conference on Hybrid Systems: Computation and Control*. ACM, 2016, pp. 71–80.

[64] R. Griesse, "Stability and sensitivity analysis in optimal control of partial differential equations," Thesis, 2007.

[65] J. A. Stankovic, "Stability and distributed scheduling algorithms," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1141–1152, 1985.

[66] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[67] ——, "Self-stabilization in spite of distributed control," in *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 41–46.

[68] S. Ghosh, "An alternative solution to a problem on self-stabilization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 4, pp. 735–742, 1993.

[69] Y. Yamauchi and S. Tixeuil, "Brief announcement: Monotonic stabilization," in *29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 2010, pp. 406–407.

[70] S. Dolev and S. Rajsbaum, "Stability of long-lived consensus (extended abstract)," in *19th annual ACM Symposium on Principles of Distributed Computing*. ACM, 2000, pp. 309–318.

[71] S. Dolev and R. Yagel, "Toward self-stabilizing operating systems," in *15th International Workshop on Database and Expert Systems Applications*. IEEE Computer Society, 2004, pp. 684–688.

[72] S. Dolev and Y. A. Haviv, "Self-stabilizing microprocessor: analyzing and overcoming soft errors," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 385–399, 2006.

[73] S. Dolev, Y. Haviv, and M. Sagiv, "Self-stabilization preserving compiler," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 6, pp. 1–42, 2009.

[74] S. Schmid, "Robust architectures for open distributed systems and topological self-stabilization: Invited paper," in *3rd International Workshop on Reliability, Availability, and Security*. ACM, 2010, pp. 1–6.

[75] B. A. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University, UK, Technical Report, 2007.

[76] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571 – 583, 2007.

[77] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, pp. 1–10.

[78] R. L. Glass and I. Vessey, "Contemporary application-domain taxonomies," *IEEE Software*, vol. 12, no. 4, pp. 63–76, 1995.

[79] B. H. Kwasnik, "The role of classification in knowledge representation and discovery," *Library trends*, vol. 48, no. 1, p. 22, 1999.

[80] D. I. K. Sjøberg, T. Dybå, and M. Jørgensen, "The future of empirical methods in software engineering research," in *Conference on The Future of Software Engineering (FOSE)*. IEEE Computer Society, 2007, pp. 358–378.

[81] R. Laddaga, "Active software," in *1st international workshop on Self-adaptive software*. Springer-Verlag New York, Inc., 2000, pp. 11–26.

[82] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, pp. 1–42, 2009.

[83] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, Part B, pp. 184–206, 2015.

[84] C. Jia, Y. Cai, Y. T. Yu, and T. H. Tse, "5W+1H pattern: A perspective of systematic mapping studies and a case study on cloud software testing," *Journal of Systems and Software*, vol. 116, pp. 206–219, 2016.

[85] Q. Gu, F. Cuadrado, P. Lago, and J. C. Dueñas, "3d architecture viewpoints on service automation," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1307–1322, 2013.

[86] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *IEEE Software*, vol. 13, no. 6, pp. 47–55, 1996.

[87] P. O. Bengtsson and J. Bosch, "Scenario-based software architecture reengineering," in *5th International Conference on Software Reuse*. IEEE Computer Society, 1998, p. 308.

[88] I. Ozkaya, P. Wallin, and J. Axelsson, "Architecture knowledge management during system evolution: Observations from practitioners," in *ICSE Workshop on Sharing and Reusing Architectural Knowledge*. ACM, 2010, pp. 52–59.

[89] P. R. Anish, B. Balasubramaniam, A. Sainani, J. Cleland-Huang, M. Daneva, R. J. Wieringa, and S. Ghaisas, "Probing for requirements knowledge to stimulate architectural thinking," in *38th International Conference on Software Engineering*. ACM, 2016, pp. 843–854.

[90] P. Avgeriou, P. Lago, and P. Kruchten, "Towards using architectural knowledge," *SIGSOFT Software Engineering Notes*, vol. 34, no. 2, pp. 27–30, 2009.

[91] D. A. Tamburri, P. Kruchten, P. Lago, and H. V. Vliet, "Social debt in software engineering: insights from industry," *Journal of Internet Services and Applications*, vol. 6, no. 1, p. 10, 2015.

[92] D. A. Tamburri, P. Lago, H. V. Vliet, and E. di Nitto, "On the nature of gse organizational social structures: An empirical study," in *IEEE 7th International Conference on Global Software Engineering*, 2012, pp. 114–123.

[93] D. A. Tamburri, P. Kruchten, P. Lago, and H. V. Vliet, "What is social debt in software engineering?" in *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013, pp. 93–96.

[94] D. A. Tamburri, P. Lago, and H. V. Vliet, "Organizational social structures for software engineering," *ACM Computing Surveys*, vol. 46, no. 1, pp. 1–35, 2013.

[95] ——, "Uncovering latent social communities in software development," *IEEE Software*, vol. 30, no. 1, pp. 29–36, 2013.

[96] N. F. Schneidewind, "Measuring and evaluating maintenance process using reliability, risk, and test metrics," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 769–781, 1999.

[97] M. Salama and R. Bahsoon, "Analysing and modelling runtime architectural stability for self-adaptive software," *Journal of Systems and Software*, vol. 133, pp. 95–112, 2017.

[98] J. C. Laprie, "From dependability to resilience," in *38th IEEE/IFIP International Conference On Dependable Systems and Networks*, 2008.

[99] R. Almeida and M. Vieira, "Benchmarking the resilience of self-adaptive software systems: perspectives and challenges," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 190–195.

[100] A. Pataricza, I. Kocsis, A. Salánki, and L. Gönczy, "Empirical assessment of resilience," in *Software Engineering for Resilient Systems*, ser. Lecture Notes in Computer Science, A. Gorbenko, A. Romanovsky, and V. Kharchenko, Eds. Springer Berlin Heidelberg, 2013, vol. 8166, pp. 1–16.

[101] J. Cámara and R. de Lemos, "Evaluation of resilience in self-adaptive systems using probabilistic model-checking," in *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2012, pp. 53–62.

[102] J. Cámara, P. Correia, R. d. de Lemos, and M. Vieira, "Empirical resilience evaluation of an architecture-based self-adaptive software system," in *10th International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, 2014, pp. 63–72.

[103] F. Chauvel, H. Song, N. Ferry, and F. Fleurey, "Robustness indicators for cloud-based systems topologies," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 307–316.

[104] T. Anderson, *Resilient computing systems; vol. 1*. Wiley-Interscience, 1985.

[105] S. Kounev, P. Reinecke, F. Brosig, J. T. Bradley, K. Joshi, V. Babka, A. Stefanek, and S. Gilmore, "Providing dependability

and resilience in the cloud: Challenges and opportunities," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, Eds. Springer Berlin Heidelberg, 2012, pp. 65–81.

[106] M. Naab and J. Stammel, "Architectural flexibility in a software-system's life-cycle: systematic construction and exploitation of flexibility," in *8th international ACM SIGSOFT conference on Quality of Software Architectures*. ACM, 2012, pp. 13–22.

[107] H. M. Wang, B. Ding, D. X. Shi, J. N. Cao, and A. T. S. Chan, "Auxo: an architecture-centric framework supporting the online tuning of software adaptivity," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–15, 2015.

[108] C. L. Nehaniv and P. Wernick, "Introduction to software evolvability," in *3rd International IEEE Workshop on Software Evolvability*. IEEE CS Press, 2007, pp. vi–vii.

[109] H. P. Breivold, I. Crnkovic, and M. Larsson, "Software architecture evolution through evolvability analysis," *Journal of Systems and Software*, vol. 85, no. 11, pp. 2574–2592, 2012.

[110] C. C. Venters, C. Jay, L. Lau, M. K. Griffiths, V. Holmes, R. R. Ward, J. Austin, C. Dibsdale, and J. Xu, "Software sustainability: The modern tower of babel," in *3rd International Workshop on Requirements Engineering for Sustainable Systems co-located with 22nd International Conference on Requirements Engineering (RE)*, 2014.

[111] P. Lago, S. A. Kocak, I. Crnkovic, and B. Penzenstadler, "Framing sustainability as a property of software quality," *Communications of the ACM*, vol. 58, no. 10, pp. 70–78, 2015.

[112] C. S. Holling, "Resilience and stability of ecological systems," *Annual Review of Ecology and Systematics*, vol. 4, no. 1, pp. 1–23, 1973.

[113] J. D. Musa and W. W. Everett, "Software-reliability engineering: technology for the 1990s," *IEEE Software*, vol. 7, no. 6, pp. 36–43, 1990.

[114] N. Guelfi, "A formal framework for dependability and resilience from a software engineering perspective," *Central European Journal of Computer Science*, vol. 1, no. 3, pp. 294–328, 2011.

[115] P. O. Bengtsson and J. Bosch, "Architecture level prediction of software maintenance," in *3rd European Conference on Software Maintenance and Reengineering (CSMR)*, 1999, pp. 139–147.

[116] N. T. Huynh, M. T. Segarra, and A. Beugnard, "A development process based on variability modeling for building adaptive software architectures," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2016, pp. 1715–1718.

[117] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness-driven resilience evaluation of self-adaptive software systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 50–64, 2017.

[118] J. Sterbenz and P. Kulkarni, "Diverse infrastructure and architecture for datacenter and cloud resilience," in *22nd International Conference on Computer Communications and Networks (ICCCN)*, 2013, pp. 1–7.

[119] H. Seung Yeob, K. Marais, and D. DeLaurentis, "Evaluating system of systems resilience using interdependency analysis," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2012, pp. 1251–1256.

[120] R. Almeida and M. Vieira, "Changeloads for resilience benchmarking of self-adaptive systems: A risk-based approach," in *9th European Dependable Computing Conference (EDCC)*, 2012, pp. 173–184.

[121] L. C. Briand, S. Morasca, and V. R. Basili, "Measuring and assessing maintainability at the end of high level design," in *Conference on Software Maintenance*, 1993, pp. 88–87.

[122] K. S. Herzig, "Capturing the long-term impact of changes," in *ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 393–396.

[123] J. M. Barnes, A. Pandey, and D. Garlan, "Automated planning for software architecture evolution," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 213–223.

[124] M. M. Lehman and J. F. Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, no. 1, pp. 15–44, 2001.

[125] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 21–30.

[126] C. B. Jaktman, J. Leaney, and M. Liu, "Structural analysis of the software architecture — a maintenance assessment case study," in *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22–24 February 1999, San Antonio, Texas, USA*, P. Donohoe, Ed. Boston, MA: Springer US, 1999, pp. 455–470.

[127] M. Lavallée and P. N. Robillard, "Causes of premature aging during software development: an observational study," in *12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. ACM, 2011, pp. 61–70.

[128] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, 2002.

[129] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: A case study of Baan ERP," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, pp. 277–306, 2005.

[130] T. Abbas and A. Ahsan, "Value based incremental software development," in *17th IEEE International Multi Topic Conference*, 2014, pp. 155–160.

[131] F. A. Moghaddam, R. Deckers, G. Procaccianti, P. Grosso, and P. Lago, "A domain model for self-adaptive software systems," in *11th European Conference on Software Architecture: Companion Proceedings*. ACM, 2017, pp. 16–22.

[132] N. M. Villegas, G. Tamura, and H. A. Müller, "Architecting software systems for runtime self-adaptation: Concepts, models, and challenges," in *Managing Trade-Offs in Adaptable Software Architectures*, I. Mistrík, N. Ali, J. Grundy, R. Kazman, and B. Schmerl, Eds. Boston, MA: Elsevier (Morgan Kaufmann), 2017, pp. 17–43.

[133] D. M. Berry, B. H. C. Cheng, and J. Zhang, "The four levels of requirements engineering for and in dynamic adaptive systems," in *11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ)*, 2005.

[134] M. I. Kamata and T. Tamai, "How does requirements quality relate to project success or failure?" in *15th IEEE International Requirements Engineering Conference (RE)*, 2007, pp. 69–78.

[135] C. Becker, S. Betz, R. Chitchyan, L. Duboc, S. M. Easterbrook, B. Penzenstadler, N. Seyff, and C. C. Venters, "Requirements: The key to sustainability," *IEEE Software*, vol. 33, no. 1, pp. 56–65, 2016.

[136] R. Mohanani, "Implications of requirements engineering on software design: A cognitive insight," in *IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 835–838.

[137] Software Engineering Standards Subcommittee of the Technical Committee on Software Engineering of the IEEE Computer Society, "IEEE Standard Dictionary of Measures to Produce Reliable Software," *IEEE Std 982.1-1988*, 1988.

[138] ——, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," *IEEE Std 982.2-1988*, 1988.

[139] M. Cline and M. Girou, "Enduring business themes," *Communications of the ACM*, vol. 43, no. 5, pp. 101–106, 2000.

[140] C. Canal, J. Murillo, and P. Poizat, "Software adaptation," *L'Objet*, vol. 12, no. 1, p. 9–31, 2006.

[141] S. Becker, C. Canal, N. Diakov, J. M. Murillo, P. Poizat, and M. Tivoli, "Coordination and adaptation techniques: Bridging the gap between design and implementation," in *Object-Oriented Technology. ECOOP 2006 Workshop Reader: ECOOP 2006 Workshops, Nantes, France, July 3-7, 2006, Final Reports*, M. Südholt and C. Consel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–86.

[142] G. Zames, "On the input-output stability of time-varying nonlinear feedback systems–part ii: Conditions involving circles in the frequency plane and sector nonlinearities," *IEEE Transactions on Automatic Control*, vol. 11, no. 3, pp. 465–476, 1966.

[143] L. Duboc, E. Letier, D. S. Rosenblum, and T. Wicks, "A case study in eliciting scalability requirements," in *16th IEEE International Requirements Engineering (RE)*, 2008, pp. 247–252.

[144] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[145] C. Del Rosso, "Continuous evolution through software architecture evaluation: A case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 5, pp. 351–383, 2006.

[146] E. Y. Nakagawa, E. P. M. de Sousa, K. d. B. Murata, G. d. F. Andery, L. B. Morelli, and J. C. Maldonado, "Software architecture relevance in open source software evolution: A case study," in *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, 2008, pp. 1234–1239.

[147] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 128–138, 1979.

[148] B. W. Boehm and K. J. Sullivan, "Software economics: A roadmap," in *Conference on The Future of Software Engineering*, 2000, pp. 319 – 343.

[149] R. Kazman, L. Bass, M. Webb, and G. Abowd, "SAAM: a method for analyzing the properties of software architectures," in *16th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 1994, pp. 81–90.

[150] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *4th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 1998, pp. 68–78.

[151] M. Barbacci, P. Feiler, M. Klein, H. Lipson, T. Longstaff, C. Weinstock, and S. Carriere, "Steps in an architecture tradeoff analysis method: Quality attribute models and analysis," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU/SEI-97-TR-029, 1998. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12927

[152] R. Kazman, J. Asundi, and M. Klein, "Quantifying the costs and benefits of architectural decisions," in *23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 297–306.

[153] R. Nord, M. Barbacci, P. Clements, R. Kazman, M. Klein, L. O'Brien, and J. Tomayko, "Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM)," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU/SEI-2003-TN-038, 2003. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6557

[154] J. Magee, J. Kramer, and D. Giannakopoulou, "Behaviour analysis of software architectures," in *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22–24 February 1999, San Antonio, Texas, USA*, P. Donohoe, Ed. Boston, MA: Springer US, 1999, pp. 35–49.

[155] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[156] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 01, pp. 31–57, 1992.

[157] D. Garlan, B. Schmerl, and S. W. Cheng, "Software architecture-based self-adaptation," in *Autonomic Computing and Networking*, Y. Zhang, L. T. Yang, and M. K. Denko, Eds. Springer US, 2009, pp. 31–55.

[158] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, "Incorporating architecture-based self-adaptation into an adaptive industrial software system," *Journal of Systems and Software*, vol. 122, pp. 507–523, 2016.

[159] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *Future of Software Engineering (FOSE)*, 2007, pp. 259–268.

[160] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[161] P. Clements, "Active reviews for intermediate designs," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, Technical Report CMU/SEI-2000-TN-009, 2000.

[162] M. Klein and R. Kazman, "Attribute-based architectural styles," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU/SEI-99-TR-022, 1999.

[163] I. Ozkaya, R. Kazman, and M. Klein, "Quality-attribute-based economic valuation of architectural patterns," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU/SEI-2007-TR-003, 2007.

[164] P. Pelliccione, P. Inverardi, and H. Muccini, "CHARMY: a framework for designing and verifying architectural specifications," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 325–346, 2009.

[165] J. Cámara, R. de Lemos, M. Vieira, R. Almeida, and R. Ventura, "Architecture-based resilience evaluation for self-adaptive systems," *Computing*, vol. 95, no. 8, pp. 689–722, 2013.

[166] J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, *Assurances for self-adaptive systems : Principles, models, and techniques*, J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds. Berlin New York: Springer, 2013.

[167] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness evaluation of controllers in self-adaptive software systems," in *6th Latin-American Symposium on Dependable Computing*, 2013, pp. 1–10.

[168] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Testing the robustness of controllers for self-adaptive systems," *Journal of the Brazilian Computer Society*, vol. 20, no. 1, pp. 1–14, 2014.

[169] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, "Quantifying resiliency of IaaS cloud," in *29th IEEE Symposium on Reliable Distributed Systems*, 2010, pp. 343–347.

[170] A. Gorbenko, V. Kharchenko, O. Tarasyuk, Y. Chen, and A. Romanovsky, "The threat of uncertainty in service-oriented architecture," in *RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*. ACM, 2008, pp. 49–54.

[171] A. Gorbenko, V. Kharchenko, S. Mamutov, O. Tarasyuk, Y. Chen, and A. Romanovsky, "Real distribution of response time instability in service-oriented architecture," in *29th IEEE Symposium on Reliable Distributed Systems*, 2010, pp. 92–99.

[172] J. Andersson, R. de Lemos, S. Malek, and D. Weyns, "Modeling dimensions of self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, 2009, vol. 5525, pp. 27–47.

[173] T. Dong and M. Hecht, "Evaluation of software dependability based on stability test data," in *25th International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1995, pp. 434–443.

[174] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[175] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[176] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008.

[177] T. Keuler, D. Muthig, and T. Uchida, "Efficient quality impact analyses for iterative architecture construction," in *7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2008, pp. 19–28.

[178] M. Neil, M. Tailor, D. Marquez, N. E. Fenton, and P. Hearty, "Modelling dependable systems using hybrid Bayesian networks," *Reliability Engineering & System Safety*, vol. 93, no. 7, pp. 933–939, 2008.

[179] D. Marquez, M. Neil, and N. E. Fenton, "A new Bayesian network approach to reliability modelling," in *5th International Mathematical Methods in Reliability Conference (MMR)*, 2007.

[180] A. Bobbio, D. Codetta-Raiteri, S. Montani, and L. Portinale, "Reliability analysis of systems with dynamic dependencies," in *Bayesian Networks*. John Wiley & Sons, Ltd, 2008, pp. 225–238.

[181] R. Roshandel, N. Medvidovic, and L. Golubchik, "A Bayesian model for predicting reliability of software systems at the architectural level," in *Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications (QoSA)*. Springer-Verlag, 2007, pp. 108–126.

[182] P. O. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-Level Modifiability Analysis (ALMA)," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[183] L. Bass, J. Ivers, M. Klein, P. Merson, and K. Wallnau, "Encapsulating quality attribute knowledge," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2005, pp. 193–194.

[184] J. Cámara, G. A. Moreno, and D. Garlan, "Stochastic game analysis and latency awareness for proactive self-adaptation," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 155–164.

[185] J. A. Stankovic, H. Tian, T. Abdelzaher, M. Marley, T. Gang, S. Sang, and L. Chenyang, "Feedback control scheduling in distributed real-time systems," in *22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, 2001, pp. 59–70.

[186] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for qos management," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 218–227.

[187] J. L. Hellerstein, S. Singhal, and Q. Wang, "Research challenges in control engineering of computing systems," *IEEE Transactions on Network and Service Management*, vol. 6, no. 4, pp. 206–211, 2009.

[188] K. S. Barber, T. Graser, J. Holt, and G. Baker, "Arcade: early dynamic property evaluation of requirements using partitioned software architecture models," *Requirements Engineering*, vol. 8, no. 4, pp. 222–235, 2003.

[189] D. Ameller, C. Ayala, J. Cabot, and X. Franch, "Non-functional requirements in architectural decision making," *IEEE Software*, vol. 30, no. 2, pp. 61–67, 2013.

[190] D. Garlan, "Software architecture: a roadmap," in *Conference on the Future of Software Engineering*, 2000, pp. 91–101.

[191] K. Angelopoulos, V. E. S. Souza, and J. Pimentel, "Requirements and architectural approaches to adaptive software systems: A comparative study," in *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2013, pp. 23–32.

[192] R. Chitchyan, C. Becker, S. Betz, L. Duboc, B. Penzenstadler, N. Seyff, and C. C. Venters, "Sustainability design in requirements engineering: State of practice," in *IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 533–542.

[193] A. van Lamsweerde, "Requirements engineering in the year 00: A research perspective," in *22nd International Conference on Software Engineering (ICSE)*. ACM, 2000, pp. 5–19.

[194] E. Letier and A. van Lamsweerde, "Agent-based tactics for goal-oriented requirements elaboration," in *24th International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 83–93.

[195] T. Becker, A. Agne, P. R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. R. Jensenius, and S. C. Stilkerich, "EPiCS: engineering proprioception in computing systems," in *IEEE 15th International Conference on Computational Science and Engineering (CSE)*, 2012, pp. 353–360.

[196] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34, no. 3, pp. 115–119, 2001.

[197] W. Emmerich, "Distributed component technologies and their software engineering implications," in *24th International Conference on Software Engineering (ICSE)*, 2002, pp. 537–546.

[198] A. Borgida, F. Dalpiaz, J. Horkoff, and J. Mylopoulos, "Requirements models for design- and runtime: A position paper," in *5th International Workshop on Modeling in Software Engineering (MiSE)*, 2013, pp. 62–68.

[199] B. H. C. Cheng et al., "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, 2009, pp. 1–26.

[200] R. de Lemos et al., *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2013, vol. 7475, pp. 1–32.

[201] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting self-aware software systems," in *11th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2014, pp. 91–94.

[202] P. R. Lewis, A. Chandra, F. Faniyi, K. Glette, T. Chen, R. Bahsoon, J. Torresen, and X. Yao, "Architectural aspects of self-aware and self-expressive computing systems: From psychology to engineering," *Computer*, vol. 48, no. 8, pp. 62–70, 2015.

[203] P. Petrov, U. Buy, and R. L. Nord, "Enhancing the software architecture analysis and design process with inferred macro-architectural requirements," in *IEEE 1st International Workshop on the Twin Peaks of Requirements and Architecture (Twin Peaks)*, 2012, pp. 20–26.

[204] E. Letier, D. Stefan, and E. T. Barr, "Uncertainty, risk, and information value in software requirements and architecture," in *36th International Conference on Software Engineering*. ACM, 2014, pp. 883–894.

[205] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements reflection: requirements as runtime entities," in *ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 199–202.

[206] K. Welsh, N. Bencomo, P. Sawyer, and J. Whittle, "Self-explanation in adaptive systems based on runtime goal-based models," in *Transactions on Computational Collective Intelligence XVI*, ser. Lecture Notes in Computer Science, R. Kowalczyk and N. T. Nguyen, Eds. Springer Berlin Heidelberg, 2014, pp. 122–145.

[207] B. Chen, X. Peng, Y. Yu, and W. Zhao, "Uncertainty handling in goal-driven self-optimization – limiting the negative effect on adaptation," *Journal of Systems and Software*, vol. 90, no. 0, pp. 114–127, 2014.

[208] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Springer Berlin Heidelberg, 2013, vol. 7475, pp. 214–238.

[209] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske, "Architecture-based reliability evaluation under uncertainty," in *Joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. ACM, 2011, pp. 85–94.

[210] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[211] W. Heaven and E. Letier, "Simulating and optimising design decisions in quantitative goal models," in *19th IEEE International Requirements Engineering Conference (RE)*, 2011, pp. 79–88.

[212] K. D. Evensen, "Reducing uncertainty in architectural decisions with aadl," in *44th Hawaii International Conference on System Sciences (HICSS)*, 2011, pp. 1–9.

[213] O. Celiku, D. Garlan, and B. Schmerl, "Augmenting architectural modeling to cope with uncertainty," in *International Workshop on Living with Uncertainties (IWLU), co-located with 22nd International Conference on Automated Software Engineering (ASE)*, 2007.

[214] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *Genetic Programming*, ser. Lecture Notes in Computer Science, M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. García-Sánchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, Eds. Springer Berlin Heidelberg, 2014, vol. 8599, pp. 137–149.

[215] G. Brataas and P. Hughes, "Exploring architectural scalability," *SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 125–129, 2004.

[216] L. Duboc, D. Rosenblum, and T. Wicks, "A framework for characterization and analysis of software system scalability," in *6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 375–384.

[217] G. Buchgeher and R. Weinreich, "An approach for combining model-based and scenario-based software architecture analysis," in *5th International Conference on Software Engineering Advances (ICSEA)*, 2010, pp. 141–148.

[218] J. Zhao, H. Yang, L. Xiang, and B. Xu, "Change impact analysis to support architectural evolution," *Journal of Software Maintenance*, vol. 14, no. 5, pp. 317–333, 2002.

[219] F. Tie and J. I. Maletic, "Applying dynamic change impact analysis in component-based architecture design," in *7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2006, pp. 43–48.

[220] D. Garlan, "Software architecture: A travelogue," in *International Conference on Future of Software Engineering*. ACM, 2015, pp. 29–39.

[221] R. K. Pandey, "Architectural Description Languages (ADLs) vs UML: a review," *SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 1–5, 2010.

[222] A. Celesti, F. Tusa, M. Villari, and A. Puliafito, "How to enhance cloud architectures to enable cross-federation," in *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, 2010, pp. 337–345.

[223] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos, "Runtime goal models: Keynote," in *2013 IEEE Seventh International Conference on Research Challenges in Information Science (RCIS)*, 2013, pp. 1–11.

[224] A. Koziolek, Q. Noorshams, and R. Reussner, "Focussing multi-objective software architecture optimization using quality of service bounds," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, J. Dingel and A. Solberg, Eds. Springer Berlin Heidelberg, 2011, vol. 6627, pp. 384–399.

[225] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: automated application of tactics in multi-objective software architecture optimization," in *Joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS (QoSA-ISARCS)*, I. Crnkovic, J. A. Stafford, D. Petriu, J. Happe, and P. Inverardi, Eds. ACM, 2011, pp. 33–42.

[226] P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, and X. Yao, "A survey of self-awareness and its application in computing systems," in *5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, 2011, pp. 102–107.

[227] T. Chen, F. Faniyi, R. Bahsoon, P. R. Lewis, X. Yao, L. Minku, and L. Esterle, "The handbook of engineering self-aware and self-

expressive systems," University of Birmingham, School of Computer Science, Birmingham, UK, Technical Report, 2014.
[228] M. Salama and R. Bahsoon, "Quality-driven architectural patterns for self-aware cloud-based software," in *IEEE 8th International Conference on Cloud Computing (CLOUD)*, 2015, pp. 844–851.
[229] R. Laddaga, "Creating robust software through self-adaptation," *IEEE Intelligent Systems and their Applications*, vol. 14, no. 3, pp. 26–29, 1999.
[230] J. C. Georgas, "Knowledge-based architectural adaptation management for self-adaptive systems," in *27th International Conference on Software Engineering*. ACM, 2005, pp. 658–658.
[231] B. A. Caprarescu, "Robustness and scalability: a dual challenge for autonomic architectures," in *4th European Conference on Software Architecture: Companion Volume*. ACM, 2010, pp. 22–26.
[232] M. E. Shin, "Self-healing components in robust software architecture for concurrent and distributed systems," *Science of Computer Programming*, vol. 57, no. 1, pp. 27–44, 2005.
[233] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2012, pp. 33–42.
[234] A. P. Engelbrecht, *Computational intelligence: An introduction*. Chichester: J. Wiley & Sons, 2002.
[235] J. Magee, J. Kramer, R. Chatley, S. Uchitel, and H. Foster, "LTSA - Labelled Transition System Analyser," accessed: July 2015. [Online]. Available: http://www.doc.ic.ac.uk/ltsa/
[236] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.
[237] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Journal*, vol. 19, no. 1, pp. 35–64, 2011.
[238] P. R. Lewis, *Computational Self-awareness and Learning Machines*. London, UK: Imperial College Press, 2014, pp. 267–280.
[239] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.
[240] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, pp. 1–39, 2013.
[241] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.
[242] Y. Falcone, J. C. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties," in *Runtime Verification: 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009, Selected Papers*, S. Bensalem and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 40–59.
[243] G. Chatzikonstantinou and K. Kontogiannis, "Run-time requirements verification for reconfigurable systems," *Information and Software Technology*, vol. 75, pp. 105–121, 2016.
[244] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, 2012.
[245] R. Calinescu et al., "Synthesis and verification of self-aware computing systems," in *Self-Aware Computing Systems*, S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, Eds. Springer International Publishing, 2017, pp. 337–373.
[246] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: Continuous assurance of non-functional requirements," *Formal Aspects of Computing*, vol. 24, no. 2, pp. 163–186, 2012.
[247] ——, "Run-time efficient probabilistic model checking," in *33rd International Conference on Software Engineering*. ACM, 2011, pp. 341–350.
[248] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *20th International Conference on Software Engineering*. IEEE Computer Society, 1998, pp. 177–186.
[249] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: A new metric for architectural maintenance complexity," in *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 499–510.
[250] R. L. Nord, I. Ozkaya, H. Koziolek, and P. Avgeriou, "Quantifying software architecture quality: Report on the First International Workshop on Software Architecture Metrics," *SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 32–34, 2014.
[251] D. Perez-Palacin, R. Mirandola, and J. Merseguer, "Software architecture adaptability metrics for QoS-based self-adaptation." ACM, 2011, pp. 171–176.
[252] D. Falessi, M. A. Babar, G. Cantone, and P. Kruchten, "Applying empirical software engineering to software architecture: challenges and lessons learned," *Empirical Software Engineering*, vol. 15, no. 3, pp. 250–276, 2010.
[253] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
[254] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *Conference on The Future of Software Engineering*. ACM, 2000, pp. 345–355.
[255] T. Dyba, T. Dingsoyr, and G. Hanssen, "Applying systematic reviews to diverse study types: An experience report," in *1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 225–234.
[256] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th international conference on Evaluation and Assessment in Software Engineering*. British Computer Society, 2008, pp. 68–77.
[257] JabRef Development Team, *JabRef*. [Online]. Available: http://www.jabref.org
[258] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.

APPENDIX A
REVIEW PROTOCOL

In this appendix, we present the research method and systematic process we followed in conducting the review (depicted in Figure A.1).

The procedure of this study followed the guidelines for conducting systematic literature reviews [75] [76]. In more details, the following steps were undertaken: (1) defining the research questions, (2) defining the search strategy, (3) executing the search, (4) selecting primary studies, (5) extracting data and analysing results, and (6) reporting the review. Details of each step are presented in the following sections.

A.1 Definition of Research Questions

The overall objective is to identify the current state-of-the-art related to stability as a software property, with a special focus on architectural stability. This review focuses on addressing the following research questions (RQs):

RQ1. How stability could be defined and characterised as a software property?

RQ2. What is the current state of research on software stability?

RQ3. Which engineering practices have been developed by the research community for realising and evaluating architectural stability?

In RQ1, we identify the definitions of stability proposed in the literature, with the goal of getting a sound definition and characterisation of this quality property. RQ2 provides information on the current state of research on software stability. By studying and categorising related studies, we can identify research gaps and potential directions with respect to software stability. In RQ3, we aim to get better insight into the current engineering practices supporting and evaluating architectural stability, to help us to determine how they can fit new software paradigms and their dynamics.

A.2 Search Strategy

A.2.1 Data sources

We conducted the search process using the automated search in the following digital libraries and indexing systems: ACM Digital Library, IEEE Xplore, ScienceDirect, and SpringerLink (details in Table A.1). These are considered as the largest and most complete scientific databases for conducting literature reviews [76] [255] and most relevant electronic databases to computer science and software engineering [256]. The selected trustworthy search sources have a direct impact on the quality of conferences and journals when retrieving the search results.

Table A.1: Search Data Sources

Database	Location
ACM Digital Library	http://dl.acm.org/
IEEE Xplore	http://ieeexplore.ieee.org/
ScienceDirect (Elsevier)	http://www.sciencedirect.com/
SpringerLink	http://link.springer.com/

A.2.2 Search String

The aim of the search string is to capture all results related to stability in software engineering. In order to check the feasibility of the search string and adjust it accordingly, we performed trial searchers in each database checking the number of returned papers and their relevance.

In the course of the search, we used a simple search string that places fewer restrictions with the aim to capture all results related to stability. The general search terms used in all databases are: `(stability OR stable) AND (software)`. The first two terms capture the different ways stability could be used. The third term makes it explicit for software. The keywords `system(s)` returned a huge number of results related to computing systems, hardware, robots and networks. Other combined keywords, such as software engineering and software systems led to a vast wide set of irrelevant results.

A.2.3 Cross-references Check

Furthermore, in order to ensure a more comprehensive set of primary studies, we used the snowballing technique –following the guidelines of [77] —to complement the search process, i.e. checking the references of the selected primary studies to find potentially relevant studies. When other quality attributes and engineering practices related to stability (e.g. resilience, robustness) were found in the selected primary studies, we have conducted separate searches to find how these concepts are defined and related to stability (reported in Section 5.3 and 5.4).

A.3 Search Execution

We used the search strings in the automated search engines of the data sources, searching by meta-data (i.e. title, abstract and keywords). For each data source, we conducted two rounds of search, one using the keyword `stability` and the other using the keyword `stable`.

The search was executed during October 2017 by the main researcher according to the search strategy under the supervision of the other researchers. In practice, particular settings were built for each search engine (details in Table A.2), since each digital library works in a specific manner. This was attempted to minimise duplications and rejections by setting the appropriate options in each search engine. Particularly, filters were applied –when available –for setting the search engine to retrieve only studies published by its own engine or to retrieve documents in English language only. Minimising results by excluding irrelevant disciplines was also used, whenever available.

During the course of the search execution, we used a spreadsheet to keep track of the search execution process. This spreadsheet contains:

- Data source –the name of the data source;
- URL –the URL of the data source;
- Search query and filters –the query string as entered to the search engine and filters used to refine the search results (e.g. language, discipline);
- Search results –the total number of search results retrieved;
- Search results file –the bibliography files exported of the search results

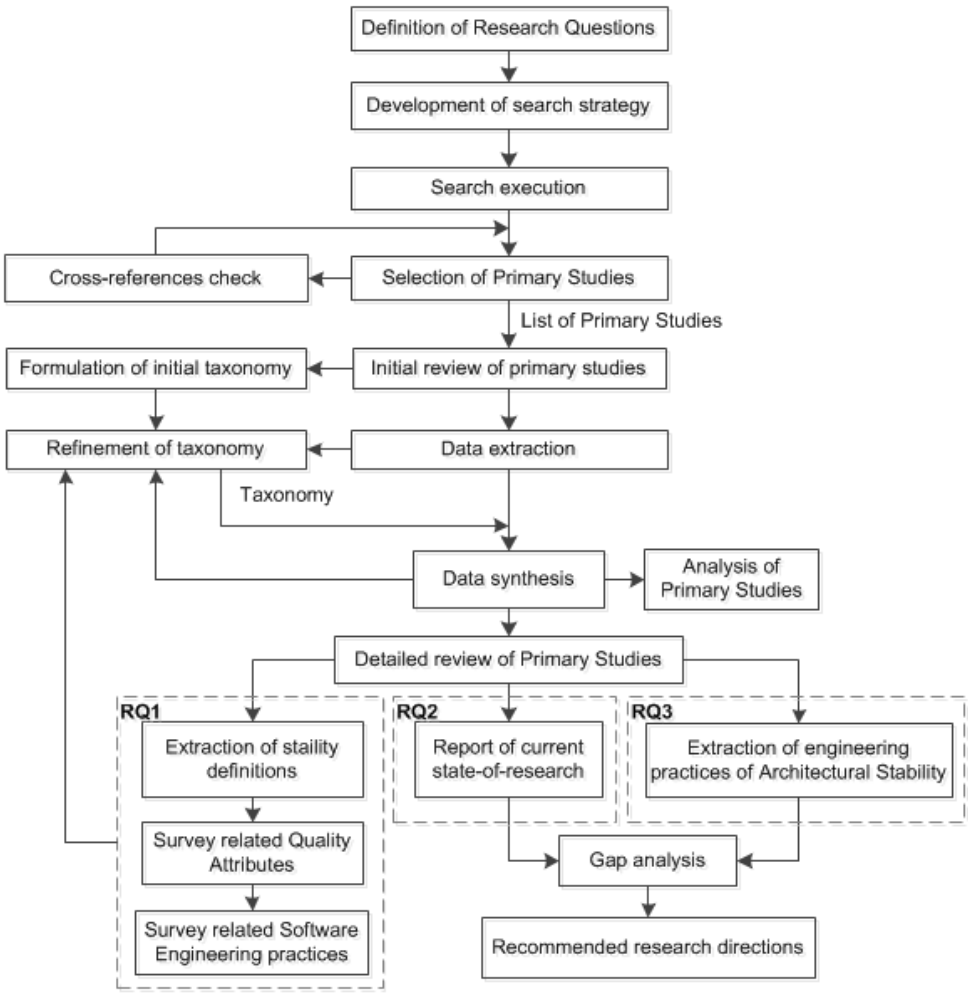

Figure A.1: Review Protocol.

As a result of this step, we obtained a total of 2418 papers (details in Table A.3). Search results were extracted as a bibliography in BibTeX format, having a final collection of bibliographies for each data source. We, then, used JabRef [257], an open source reference manager system that is able to manage BibTeX databases, to merge the search results files into one bibliography file after detecting and removing duplicates.

A.4 Selection of Primary Studies

During the screening of the search results, we closely examined the title, abstract, introduction and conclusion for each candidate paper to determine the relevance of the paper. In some cases when these do not provide enough information to decide the relevance of the paper, we read the whole paper. When similar studies are reported in several papers as work-in-progress, the most comprehensive version is considered, unless significant details were reported in the earlier version.

The selection was performed with respect to the inclusion and exclusion criteria defined in Table A.4. The references of the selected primary studies were checked to find possible missed relevant studies, where these papers are, then, taken through the same process of primary studies selection.

A total of 166 papers have been selected as primary studies after the study selection and cross-referencing steps. The reference list of the primary studies appears in Appendix B.

A.5 Data Extraction, Synthesis and Analysis

For each selected primary study, the whole paper was read to extract the relevant information answering the research questions. The data extraction and analysis were motivated by finding information for defining stability, describing different aspects of stability, related software engineering practices, and contextual aspects affecting stability. For each study, data items were extracted and recorded in a spreadsheet.

Data synthesis involved collating and summarising data extracted from primary studies. In this stage, we further analysed the results and extracted statistics. For the data synthesis, the extracted data was inspected for similarities, in order to define how results could be encapsulated. Our

Table A.2: Search Execution (search strings and settings)

Database	Search query and settings
ACM Digital Library	(+stability +software) Publisher: ACM Content Formats: PDF
	(+stable +software) Publisher: ACM Content Formats: PDF
IEEE Xplore	stability AND software in Metadata only Publisher: IET, IEEE Content Type: Conference Publications, Journals & Magazines, Books & eBooks Publication Title: International Conference on Computational Intelligence and Software Engineering (CiSE) 2009, International Conference on Computer Science and Software Engineering 2008, IEEE Transactions on Software Engineering, IEEE Software, IEEE Transactions on Computers, Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD) 2007, IEEE International Conference on Information Reuse and Integration (IRI) 2007, International Conference on Computer Application and System Modeling (ICCASM) 2010, IEEE Transactions on Industry Applications,
	stable AND software in Metadata only Publisher: IET, IEEE Content Type: Conference Publications, Journals & Magazines, Books & eBooks Publication Title: IEEE Transactions on Software Engineering, International Conference on Computer Science and Software Engineering 2008, International Conference on Computational Intelligence and Software Engineering (CiSE) 2009, Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD) 2007, Computer, IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Software, IEEE Transactions on Reliability, IEEE Transactions on Knowledge and Data Engineering,
ScienceDirect	TITLE-ABSTR-KEY(stability) and TITLE-ABSTR-KEY(software) [All Sources(Computer Science)]
	TITLE-ABSTR-KEY(stable) and TITLE-ABSTR-KEY(software) [All Sources(Computer Science)]
SpringerLink	with all of the words: software where the title contains: stability within Discipline: Computer Science Subdiscipline: Software Engineering
	with all of the words: software where the title contains: stable within Discipline: Computer Science Subdiscipline: Software Engineering

Table A.3: Search Results

Database	Search results
ACM Digital Library	1222
IEEE Xplore	342
ScienceDirect	668
SpringerLink	186
Total	2418

approach for synthesising findings is based on the synthesis method "thematic analysis/synthesis" [258], where we identified themes derived from data extracted from primary studies and targeted to answer the research questions.

The analysis of extracted information aims at investigating the notion of stability, its characteristics, and how software engineering practices can contribute to achieving it. We also performed quantitative analysis on the results (reported in Section 6.1).

APPENDIX B

APPENDIX C
STABILITY IN PRIMARY STUDIES

The definitions of stability extracted from the primary studies and their analysis are listed in Table C.1. The characterisation of stability extracted from the primary studies according to the taxonomy dimensions is shown in Tables C.2, C.3, C.4, C.5 for the different levels (code, requirements, design, architecture respectively). A study may appear in multiple tables if it considers stability at more than one level, with the exception of the ISO/IEC 9126-1 standard [**?**] and the IEEE Recommended Practice on Software Reliability [**?**], which are not listed in any table.

Within the tables in this appendix, we used the following abbreviations: **C** = code; **D** = design; **A** = architecture; **R** = requirements; **St** = structural; **L** = logical; **F** = functional; **Sy** = syntactic; **B** = behavioural; **DevPh** = Development phase; **OpPh** = Operation phase; **M&EvPh** = Maintenance and Evolution phase; **Op** = Operational; **Mnt** = Maintenance; **Ev** = Evolutionary; **Re** = Reuse; **Retro** = Retrospective; **Pro** = Prospective; **H** = Human-involved; **Auto** = Automated; **Auton** = Autonomous.

Table A.4: Studies Selection Criteria

	Inclusion Criteria
I1.	Papers published in conferences and journals as full research paper
I2.	Literatures published as books and book chapters
I3.	Papers including definitions of stability in software engineering
I4.	Papers discussing general or particular aspects of software stability
I4.	Papers defining and characterising other quality attributes related to stability
I5.	Papers implementing or extending software engineering practices for stability
I6.	Papers discussing aspects influencing stability
	Exclusion Criteria
E1.	Papers not in the form of a full research paper, i.e. in the form of abstract, tutorials, presentation, or essay
E2.	Short and position papers presenting new and emerging ideas, as well as papers presented in doctoral symposiums
E3.	Papers with abstract not available
E4.	Papers not written in English language
E5.	Papers focusing on stability in other computer science areas (e.g. operating systems, robotics, networks, hardware, algorithms, logic programming, computational logic)
E6.	Papers focusing on stability in other disciplines (e.g. control theory or dynamic systems)
E7.	Papers focusing on stability of software product lines, project management or development process

Table C.1: Definitions of Stability in Software Engineering Literature

Ref.	Definition	Dimensions of Stability		
		Level	Aspect	Purpose
[?]	"the resistance to the amplification / propagation of changes that has been made to a given program"	C	-	Mnt
[?]	"the resistance to the potential ripple effect that the program would have when it is modified"	C	St	Mnt
[?], [?]	"a measure of the resistance to the impact of a modification to a module on other modules in the program in terms of logical considerations"	C	L	Mnt
[?], [?]	"a measure of the resistance to the impact of a modification to a module on other modules in the program in terms of performance considerations"	C	B	Mnt
[?]	"the extent to which the structure of the design is preserved throughout the evolution of the software from one release to the next"	D	St	Ev
[?]	"requirements stability can be determined using the number of expected changes based on experience or knowledge of forthcoming events that affect the organisation, functions, and people supported by the software system"	R	-	-
[?]	"the capability of the software product to avoid unexpected effects from modifications of the software"	-	-	Mnt
[?]	Stability "characterises the sensitivity to change of a given system that is the negative impact that may be caused by system changes"	-	-	-
[?]	"a measure of how well it accommodates the evolution of the system without requiring changes to the architecture"	A	St	Ev
[?], [?]	"the ease with which a software system or a component can evolve while preserving its design as much as possible"	D	St	Ev
[?], [?], [?], [?], [?], [?], [?], [?]	"a quality that refers to the extent an architecture (structure) is flexible to endure evolutionary changes in stakeholder's requirements and the environment, while leaving the architecture intact"	A	St	Ev
[?]	"the resistance to interclass propagation of changes that the design would have when it is modified"	D	L	Mnt
[?]	"the ease with which a software item can evolve while preserving its design"	D	St, L	Ev
[?]	"the degree of modification of the component"	C	-	Re
[?]	"the extent to which the structure of the design is preserved throughout the evolution of the software from one release to the next"	D	St	Ev
[?]	"A design characteristic of software is stable if, when observed over two or more versions of the software, the differences in the metric associated with that characteristic are considered, in the context, to be small."	D	-	Ev
[?]	"the ability of a software artefact to keep unchanged along with the time"	-	-	Ev
[?]	"the ability to adapt to changes by its flexible configuration mechanism"	-	-	Op
[?]	"the probability that a business model or a component remains stable in a given period of time"	-	St	Ev
[?]	"Design stability encompasses the sustenance of system modularity properties and the absence of ripple-effects in the presence of change"	D	St	Ev
[?]	"how easy or difficult is it to keep the system in a consistent state during modification?"	C	-	Mnt
[?], [?]	"the ability of the high-level design units to sustain their modularity properties and not succumb to modifications"	A	St	Mnt
[?]	"how well does the system avoid unexpected effects after a modification"	C	-	Mnt
[?]	"the ability of a module to remain largely unchanged when faced with newer requirements and/or changes in the environment"	C	Sy, St	Ev
[?], [?]	A software or a module is stable "if its interface or implementation is not undesirably modified and ripple effects do not manifest in the presence of changes"	C	St, Sy	Mnt, Re
[?]	"robustness against input or code perturbations"	C	B	Op
[?]	"a quality characteristic that shows a software product's resilience to changes in the original requirements of the product"	-	-	Mnt, Ev
[?]	A software system is said to be stable "if changes result in a new version that is substantially identical to a version that has been thought to be reasonably well tested and assumed not to have any significant problems".	-	-	Op, Mnt, Ev
[?]	"the degree to which a class is subject to change, due to changes in other, related classes, considering the probability of such classes to change as equal to a certain value"	D	St, Sy	Mnt

Table C.2: Characterisation of stability in primary studies at the code level

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]	x				x			x		x	x		x		x		
[?]		x			x			x		x				x	x		
[?]		x			x			x		x				x	x		
[?]		x			x			x		x			x		x		

Table C.2 (*cont.*)

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]				x				x			x		x		x		
[?]		x			x			x		x			x			x	
[?]		x			x			x		x			x		x		
[?]	x	x						x				x	x			x	
[?]		x			x	x			x					x	x		
[?]		x	x			x					x			x		x	
[?]				x				x			x		x		x		
[?]	x	x		x						x					x		
[?]		x	x			x					x			x		x	
[?]		x			x			x		x			x			x	
[?]		x		x				x		x					x		
[?]	x	x		x						x							
[?]	x	x		x						x					x		
[?]	x			x				x			x		x		x		
[?]					x	x			x					x			x
[?]	x		x	x								x					
[?]	x		x	x								x					
[?]		x		x				x		x					x		
[?]		x		x				x			x		x		x		
[?]		x		x				x		x					x		
[?]			x			x					x			x		x	
[?]	x	x		x						x							
[?]					x		x		x					x		x	
[?]		x		x				x		x					x		
[?]		x		x				x		x					x		
[?]					x		x		x					x			x
[?]		x		x				x		x					x		
[?]					x			x			x		x				
[?]		x		x				x		x			x				
[?]		x			x		x							x			x
[?]		x			x			x			x		x				
[?]		x			x			x			x		x				
[?]		x			x			x			x		x				
[?]			x					x		x					x		
[?]					x		x		x				x				x
[?]		x				x					x			x	x		

Table C.3: Characterisation of stability in primary studies at the requirements level

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]		x															
[?]				x				x					x		x		
[?]																	
[?]		x			x			x			x		x		x		
[?]		x			x			x			x		x		x		
[?]		x				x					x			x	x		
[?]		x				x					x			x	x		
[?]		x				x					x			x	x		
[?]		x				x								x			
[?]			x					x		x					x		
[?]					x		x		x				x				x
[?]		x				x					x			x	x		

Table C.4: Characterisation of stability in primary studies at the design level

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]	x					x				x				x	x		
[?]				x											x		

Table C.4 (*cont.*)

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]	x		x					x			x		x		x		
[?]	x		x					x			x		x		x		
[?]	x		x					x			x		x		x		
[?]				x				x			x		x		x		
[?]	x		x					x			x		x		x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]				x				x						x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x							x		x	x		x			x	
[?]		x						x		x			x		x		
[?]	x	x				x						x		x	x		
[?]				x				x						x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]				x				x		x						x	
[?]				x				x			x		x		x		
[?]	x							x		x			x				
[?]		x						x			x		x		x		
[?]				x				x			x		x		x		
[?]	x					x						x		x	x		
[?]	x	x				x				x				x	x		
[?]	x	x				x						x		x	x		
[?]	x	x				x						x		x	x		
[?]	x										x		x		x		
[?]	x							x		x	x		x			x	
[?]				x				x			x			x	x		
[?]	x	x				x						x		x	x		
[?]	x					x									x		
[?]	x					x									x		
[?]	x					x								x	x		
[?]	x	x				x								x	x		
[?]	x					x								x		x	
[?]		x												x		x	
[?]	x		x			x								x			
[?]	x					x						x		x			
[?]	x			x		x						x		x			
[?]	x			x		x				x				x			
[?]	x			x		x						x		x			
[?]	x					x				x	x			x			
[?]	x		x		x	x								x	x		
[?]				x				x		x							
[?]					x	x						x		x		x	
[?]	x	x				x						x		x	x		

Table C.4 (*cont.*)

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]	x	x				x						x		x	x		
[?]	x							x			x	x	x		x		

Table C.5: Characterisation of stability in primary studies at the acrchitecture level

Ref.	What					When			Why				How		Who		
	St	L	F	Sy	B	DevPh	OpPh	M&EvPh	Op	Mnt	Ev	Re	Retro	Pro	H	Auto	Auton
[?]		x			x				x								
[?]	x							x			x		x		x		
[?]	x					x					x			x	x		
[?]	x					x					x			x	x		
[?]	x					x					x			x	x		
[?]	x					x					x			x	x		
[?]	x					x					x			x	x		
[?]					x		x			x				x			x
[?]	x					x					x				x		
[?]	x					x					x			x	x		
[?]	x					x					x			x	x		
[?]	x							x		x					x		
[?]	x					x					x			x	x		
[?]	x					x					x			x			x
[?]					x		x			x				x			x
[?]					x		x			x							x
[?]	x					x					x			x	x		
[?]	x							x		x					x		
[?]		x					x			x							x
[?]		x					x			x				x		x	
[?]	x	x				x					x				x		
[?]		x			x		x			x				x			x
[?]					x		x			x				x			x
[?]		x			x					x							
[?]					x		x			x			x				x
[?]					x					x							
[?]	x							x			x	x	x		x		
[?]		x			x					x							
[?]		x					x			x				x			x
[?]		x					x			x				x			x
[?]					x												
[?]		x					x			x				x			x
[?]	x							x			x		x			x	
[?]	x							x			x	x	x		x		
[?]	x							x			x	x	x		x		
[?]	x				x			x		x			x		x		
[?]	x							x			x	x	x		x		