ARCHITECTURAL STABILITY OF SELF-ADAPTIVE SOFTWARE SYSTEMS

by

MARIA MOURAD EBEID MELEKA SALAMA

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
July 2018

Abstract

Stakeholders and organisations are increasingly interested in software longevity, given the increasing dependence on software systems. Stability is a long-term property of utmost strategic 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 runtime operation, technical changes and cost-effective maintenance and evolution. As architectures have a profound effect on the operational lifetime of the software and the quality of service provision, architectural stability could be considered a primary criterion towards achieving the longevity of the software.

This thesis studies the notion of stability in software engineering with the aim of understanding its dimensions, facets and aspects, as well as characterising it. The thesis further investigates the aspect of *behavioural stability* at the architectural level, as a property concerned with the architecture's capability in maintaining the achievement of expected quality of service and accommodating runtime changes, in order to delay the architecture drifting and phasing-out as a consequence of the continuous unsuccessful provision of quality requirements.

The research aims to provide a systematic and methodological support for analysing, modelling, designing and evaluating architectural stability. The novelty of this research is the consideration of stability during runtime operation, by focusing on the stable provision of quality of service without violations. As the runtime dimension is associated with adaptations, the research investigates stability in the context of self-adaptive software architectures, where runtime stability is challenged by the quality of adaptation, which in turn affects the quality of service. The research evaluation focuses on the effectiveness, scale and accuracy in handling runtime dynamics, using the self-adaptive cloud architectures.

To all those from whom I have learnt...

Dédié à la mémoire de mon père et ma grand-mère...

Acknowledgements

First and foremost, my enormous gratitude and thanks go to the Almighty God for his ever blessings.

I am perpetually indebted in thanks to my supervisor Dr. Rami Bahsoon for his dedicated supervision, helpful and endless support, patience and motivation. His guidance has provided me with great help throughout my PhD and in every aspect of my research work. His kindness and care have encouraged me to stay on track. His outlook and comments have inspired me.

My sincere thanks to my external supervisor Prof. Rajkumar Buyya for his support, insights and encouragements on my research. This work would never have been completed without his support, warm welcome and dedication during the research visit and afterwards. His kindness, prudence and work of ethics have made me enjoy the research work with him.

Special thanks to the members of my Thesis Group who took time to provide guidance. I would like to thank Prof. Xin Yao for his timely comments, and Dr. Dave Parker for his insightful and encouraging comments. Their critical eye, words of insight and perspective have greatly guided my research and helped in paving the way.

I am also deeply indebted and grateful to Prof. Patricia Lago for lending me time with her great knowledge. Her enthusiasm for doing research, helpful cooperation and genuinely constructive comments were valuable and informed my research, in more ways than one.

Acknowledges are given to all the administrative staff of the School of Computer Science for their great support, kindness and welcome throughout the PhD course, especially Patrycja Adams, Sarah Brookes and Helen Whitby. Thanks also to Peter Hancox, Dave Parker and Steve Vickers for their role as Research Students Tutor, as well as Jon Rowe and Andrew Howes for their role as Head of School. Thanks to the Research Committee for their support and listening, especially Achim Jung.

Thanks to the people I have been fortunate to have throughout this fascinating journey. I would like to thank Dr. Tao Chen and Dr. Abdessalam Elhabbash for the stimulating and useful discussions. I would like also to thank Khulood, Bram, Mohab, Sara, Wad, Carlos and Satich, who made this period of my life so enjoyable.

Special thanks to the people I met during my research visits. Thanks to the CLOUDS Lab at the University of Melbourne, especially Dr. Maria Rodriguez and Dr. Rodrigo Calheiros for the useful discussions. Thanks also to Dr. Giuseppe Procaccianti and Dr. Nelly Condori-Fernandez from the Software and Services research group at Vrije Universiteit Amsterdam for their support. The experience that I gained during those

visits was very worthwhile and everyone there was very welcoming and always willing to help.

Many thanks also to Dr. Amir Zeid for his continuous mentorship regardless of distance and busyness. I would like to express my sincere appreciation to Father John Yanny, who was always standing by my side, for his kindness, unconditional love and sincere support. Warm thanks to my friends who were always there, understanding and supportive during these years.

Gratitude and sincere appreciation to my family for their continuous support. Immense thanks beyond measure to my mother for her dedicated and moral support. My gratefulness goes to my sister for her continuous encouragements. Never to forget my father, whom I owe where I am today.

I acknowledge the School of Computer Science for providing me with the scholarship to pursue my doctoral studies. The research visit to the University of Melbourne was supported by the U21 PhD Scholarship and the CLOUDS Lab. The visits to Vrije Universiteit Amsterdam were supported by Stevie Jivani Student Development and the Software Architecture Summer School scholarships.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A question of need is a question of taste.
— Neil Tennant & Chris Lowe

1.1 Motivation

The increasing dependence on software systems and services is making software *longevity* a highly desired feature [6] [7]. Given the different types of dynamics throughout the software lifetime, long-lived software systems should handle different types of changes (e.g. evolution, maintenance and runtime changes) [8]. A long-lived system is capable of remaining largely intact while supporting these changes [8]. Informally, *stability* refers to remaining intact or unchanged. As such, it is widely accepted that stability is a property to reflect longevity concerns. 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 [9], as longevity is highly dependent on the ability to retain stability.

Large industrial software systems require delivering acceptable levels 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) [10] [11]. The dynamic and unpredictable operational conditions of many open systems challenge the Quality of Service (QoS) provision and its stability. By that, stability is essential for providers and practitioners to prevent performance degradation and enforce QoS at runtime, especially in peak demand [10].

From an economic point of view, stability is desirable to safeguard customers' satisfaction and service provider reputation, as well as to avoid SLAs penalties. The emergence of new software paradigms (e.g. mobile, cloud, cloud federations, smart environments) and new architectural styles (e.g. self-adaptive) brings into concerns the need for a shift to a wider concept of stability that tackles *runtime changes* of modern software systems and their environment, as well as the uncertainty faced by architectures during operation.

As software systems undergo many cycles of *maintenance changes* [12], a stable architecture can reduce maintenance costs/effort and lessen the ripple effect of changes and the

need for re-factoring [13]. As such, software artefacts, designed in a way that the impact of changes is minimal, i.e. stable, remarkably affect the maintenance process [14].

Since evolution is unavoidable, iterative long-term changes are implemented during evolution cycles for facing changing requirements [15]. Software artefacts capable of supporting *evolutionary changes* would bring long-term benefits, delaying phasing out [12].

The interest in software reuse is also increasing, as stakeholders are concerned with building software systems that are scalable, more reliable, less expensive and within shorter time-to-market [16]. Evidently, this requires performing *modifications for reuse* in multiple contexts and projects. A software artefact (e.g. component, module or architecture) could be easily reused if modifications and ripple changes are controlled; henceforth, stability is advantageous to effective software reuse [16].

The paramount importance of stability is evident in the industrial context, since it influences software quality, cost and longevity throughout the software life-time. As stakeholders and organisations are increasingly looking for software longevity, stability could be considered a primary criterion towards achieving longevity, and a fundamental property to sustain the whole system. 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.

1.2 Research Problem and Questions

Modern software systems are increasingly becoming complex, heterogeneous and pervasive. They tend to operate in environments undergoing unpredictable changes. Such challenges can have impact on the software lifetime and the quality of service provisioned. Stability is an essential property for long-lived software systems, as it indicates the capability to maintain service provision with expected qualities, accommodate maintenance and evolutionary changes, and unlock potentials for reuse [13]. 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.

Researchers in the field of software engineering have studied stability with respect to different software artefacts. 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). 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) [6]. Also, there have been many stability developments in other disciplines (e.g. control theory, dynamic systems theory) that could benefit software engineering for realising stability as a software property. This indicates the need for characterising the notion of stability in software engineering.

As architectures have a profound effect throughout the software life-span [17] [18], architectural stability tends to reflect on the success of the system in supporting and tolerating continuous changes in the long-term, while reducing the likelihood of architectural

drifting and phasing-out [15] [19]. Architecting for stability is becoming a necessity and a critical requirement for the longevity and dependability of modern software systems over time [20].

As modern software systems operate in unpredictable environments, self-adaptation has been motivated as a solution to achieve the necessary level of dynamicity, as well as to comply with the runtime changes, fluctuations in workloads and environmental conditions [21] [22] [23]. Self-adaptive architectures are expected to manage themselves following the principles of autonomic computing, to respond to changes in end-user requirements and the environment coping with uncertainty in runtime operation [24], for continued satisfaction of quality requirements under changing context conditions [25]. In this context, stability is a runtime property that has to consider a dynamic view of the world. This dynamic view shall not consider only intactness during design-time, but also the runtime behaviour, so that the system can be seamlessly adapted and ensure a constant provision of the intended services. Even though adaptation mechanisms have been widely investigated, runtime stability was not explicitly tackled [25] [26].

Architectures are used as the basis for self-adaptation (termed architecture-based self-adaptation) [27] [28]. A self-adaptive architecture is expected to perform adaptations that converge towards the quality of service objectives (adaptation goals) [25] [26] without performing unnecessary adaptations [22]. An unstable architecture will repeat the adaptation process indefinitely with the risk of not reaching the adaptation goals, or probably degrading other quality attributes [25] [26]. Such architectures would result in not provisioning the expected quality of service and consequent service violations, as well as inefficient behaviour when performing unnecessary runtime adaptations not converging towards quality objectives. Despite the influx of research in self-adaptivity, there is a general lack of systematic methods for evaluating the stability of self-adaptive architectures' behaviour.

To tackle behavioural stability of self-adaptive architectures, we argue that stability should cover both components of the self-adaptive architecture, that are the *managed system* and the *managing system*, i.e. the physical system and the adaptation controller [25] [26]. In the community of self-adaptive architectures, adaptations mechanisms focus on the quality attributes delivered, but the properties of adaptations and their implications on the architecture and their stability are widely ignored [29].

Scope of the thesis. The focus of this thesis is 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 [30]. Stability of software product lines and software ecosystems is not covered, as both are different in nature from software systems [31] [32] and require special consideration.

Problem Statement. There is a lack of clear characterisation of the term stability in the software engineering community and systematic treatment of stability for long-living software systems.

Research Questions. The thesis is concerned with the following research questions:

RQ1. How to characterise the notion of stability in software engineering?

RQ2. What are the primitives for realising and engineering stability for self-adaptive software architectures?

RQ3. How to analyse and model runtime behavioural stability for self-adaptive software architectures?

RQ4. What are the engineering practices (specifically design and evaluation) to support runtime behavioural stability for self-adaptive software architectures?

1.3 Research Methodology

The research aims to provide uniform characterisation and systematic support for engineering and reasoning about stability. The research attempts to capture the essence and nuances of the notion, by characterising the concept, analysing its primitives and dimensions, with the intent to set the grounds for this new era.

To resolve the research questions and achieve the research aims, this thesis adopts a classical research design methodology inspired by [33] [34]. The methodology is applied iteratively to guide the research process, as described below.

- *Problem identification.* To acquire knowledge about the problem domain, a survey of the state-of-the-art has been conducted, following the guidelines of systematic literature reviews. The survey helped in identifying the research landscape, characterise the notion, and analyse the gap in research. As the understanding of the problem domain is gained, the research direction converged to the runtime behavioural aspect of stability at the architectural level, which is the pivotal problem addressed in this thesis.

- *Objectives of the solution.* Driven by the identified problem, the proposed solution aims to systematically handle architectural stability as a behavioural aspect during the runtime operation of self-adaptive software systems.

- *Design and development.* For providing understanding of the concept, the thesis presents requirements for realising stability and design principles inspired by Control Theory. Further, a systematic approach for analysing and modelling stability is developed. As the survey findings revealed the lack of engineering practices in realising runtime behavioural stability, the thesis fundamentally improved and extended suitable mechanisms for realising stability.

- *Demonstration.* The thesis adopts the self-adaptive cloud architectures case for demonstrating the usefulness of the research. The cloud computing paradigm is a challenging example for stability, due to the dynamics, unpredictability and uncertainty of the operation environment, as well as the on-demand nature of service provision [35].

- *Evaluation.* The applicability of the proposed research is qualitatively evaluated using the cloud architectures case. As research in Cloud Computing is experimental in nature [36], experimental quantitative evaluation is conducted in a controlled environment using simulations and benchmarks.

1.4 Thesis Contributions

1.4.1 Thesis Roadmap

As stability has been considered by many research communities within the software engineering discipline that has widely-spread during the last decades, the state-of-the-art is first surveyed (Chapter 2). Findings have revealed that stability has been defined and treated in many different ways in the software engineering community. This motivated the need for a taxonomy to analyse the concept and related qualities in order to reach an agreement on how stability can be positioned as a first-class essential property. 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.

In light of such characterisation and findings in the literature, stability has been found treated across many software artefacts, such as code, design, architectures. Among these artefacts, the thesis further investigates stability at the architectural level. We argue that the "architecture" is the appropriate abstract level for understanding stability given the complexity and scale of modern software systems for many reasons. First, architectures have been recognised as a key asset in building complex software-intensive systems [6], and have a profound effect throughout their life-span. Second, architectures have been recognised as a key for software reuse [37], maintenance and evolution [15]. Third, architectures have a strong impact on the quality of service delivered and system quality attributes [37], as well as could 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 is becoming the norm [38] [39]. By that, stability at the architecture level is a key for modern software longevity.

The review has also revealed that stability was tackled from the structural aspect during the architecture design stage. Structural stability has been mainly concerned with the extent to which the architecture "structural design" remains intact without entailing large and disruptive modification. Meanwhile, an architecture that is not capable of providing the expected quality of service (i.e. behavioural requirements) during operation would result in service violations and increase the likelihood of drifting and phasing-out. In the context of self-adaptive software systems, a stable self-adaptive architecture is expected to keep the fulfilment of quality stable, while performing adaptations that converge towards these objectives and eliminating unnecessary ones.

With the vision of stability as an *architectural runtime property* that should cope with dynamics, this thesis studies the *behavioural aspect* of stability as a property that reflects on the performance of the architecture during runtime and needs to be observed

dynamically as a moving target during *operation*.

To this end, this thesis provides conceptual design principles inspired from Control Theory (which contributed to designing self-adaptive software systems) for capturing behavioural stability (Chapter 3). Based on such principles, the concept is further analysed to guide understanding of stability (Chapter 4). Guided by the analysis results, the thesis presents modelling behavioural stability (Chapter 5) and different engineering practices for realising stability, as the literature review has revealed that the lack of software engineering practices supporting the stability problem. These include a reference architecture design and requirements modelling (Chapter 6), as well as reasoning techniques for engineering stability-aware adaptations (Chapter 7). The thesis motivates the use of self-awareness computing for the benefit of stability, as it tends to be general enough to cover instantiations of self-adaptive environments. On another side, architecture evaluation approaches have been found limited to the architecture's structure at design-time. Thus, the thesis provides a systematic approach for evaluating architectural behavioural stability and developed a suitable tool for evaluation (Chapter 8). The thesis roadmap is shown in Figure 1.1.

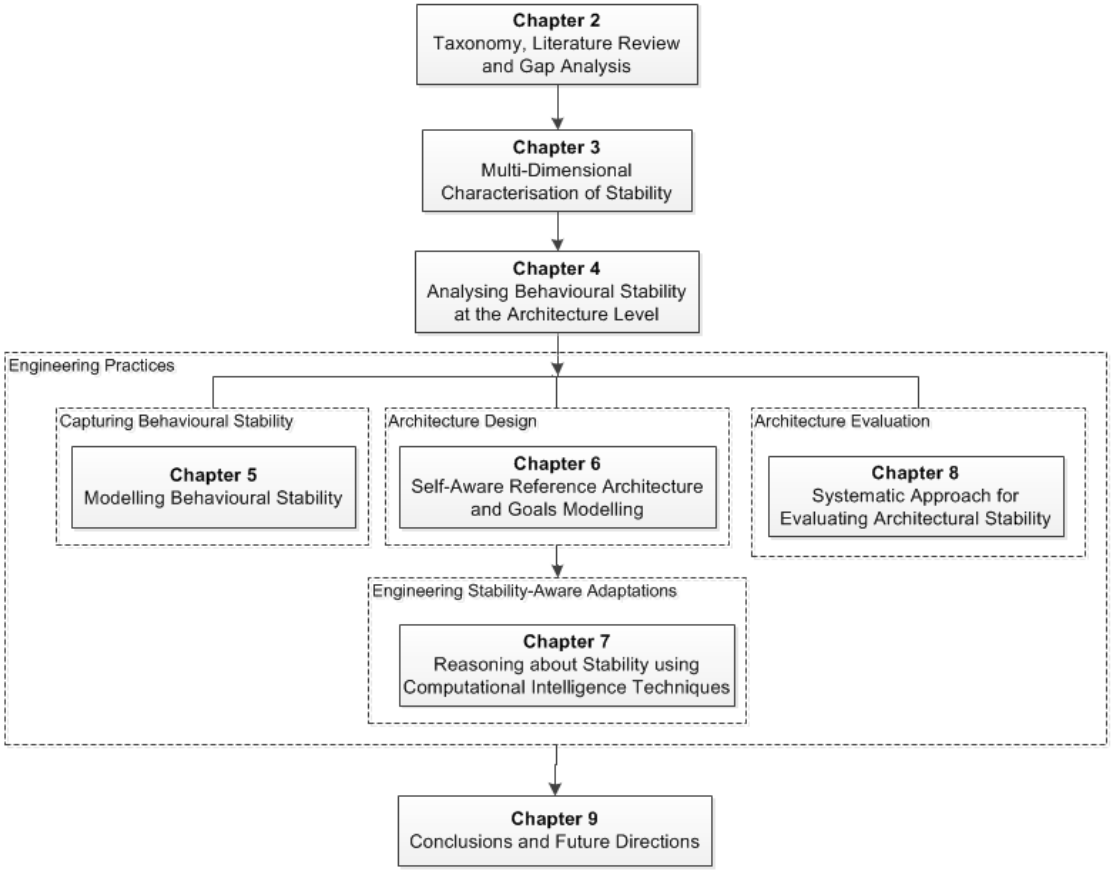

Figure 1.1: Thesis Roadmap

1.4.2 Summary of Contributions

The main contributions of this thesis are as follows.

- *Survey of the state-of-the-art on stability in software engineering.* A systematic literature review was conducted to survey the current research landscape and build the required understanding of the concept, related properties and practices. As stability has been found interpreted in many ways different ways, we proposed a taxonomy for characterising the concept. Focusing on the architectural level, the review closely surveyed related engineering practices. The survey has also identified the gaps in the literature and motivating the need for a new characterisation and engineering approaches, paving the way to contributions of the next chapters. This contribution partially addresses the research question *RQ1*.

- *Characterising and engineering the notion of stability.* Based on the taxonomy dimensions, the thesis proposes a multi-dimensional perspective for characterising and engineering stability as a software property. This perspective contributes to understanding the facets related to stability, advance the way of understanding the concept. As such, new requirements are identified. The thesis also draws designing principles for capturing the intended behaviour. This contribution is mainly concerned with addressing *RQ2* and partially addresses *RQ1*.

- *A novel approach for analysing and modelling stability.* Based on the stability design principles, the thesis proposes a novel methodology for the analysis and modelling of stability. The objective of the analysis is deeply understanding the intended behaviour, which will guide the course of the research. The proposed methodology to support architectural stability consists of three subsequent main phases: stability analysis, stability modelling, and stability runtime support, where the outcome of each phase is carried towards the next phase. *RQ3* is addressed by this contribution.

- *Architecture design-support principles for stability.* Drawing on the gap analysis identified in the review regarding the engineering practices related to stability, the thesis builds a reference architecture and goals modelling for stability. The proposed architecture design leverages self-awareness primitives that have been found suitable for handling trade-offs and time-related issues of stability attributes. The architecture incorporates quality self-management generic components and embeds a catalogue of architecture tactics within self-awareness capabilities. Stability goals modelling includes fine-grained and dynamic knowledge representation of the runtime goals, i.e. goals attributes necessary for enabling self-awareness and measures of goals satisfaction in relation to adaptation decisions. This contribution partially addresses *RQ4*.

- *Reasoning about stability using computational intelligence techniques.* The reference architecture is further developed by implementing computational intelligence techniques within self-awareness capabilities. The goal-awareness realises the symbiotic relation between the stability goals model and self-awareness component for managing stability goals during runtime. The time-awareness level implements an online learning technique for reasoning about stability in the long-run while learning from

historical information. Trade-offs between different stability attributes are managed at the meta-self-awareness level, using model verification of stochastic games using PRISM-games 2.0. This contribution partially addresses *RQ4*.

- *Systematic approach for evaluating architectural stability.* The evaluation approach addresses stability evaluation in the different phases of the software lifecycle, where the outcome of the design-time could be used for making architectural decisions or is taken forward for use in runtime evaluation during the operation phase. This contribution partially addresses *RQ4*.

- *Evaluation of applicability and experimentations.* For their runtime dynamics, self-adaptive cloud architectures case is used for evaluation throughout the thesis. The proposed analysis methodology, reference architecture and evaluation approach are qualitatively evaluated to show their applicability and added values. Using benchmarks, the thesis is quantitatively evaluated by simulation-based experiments. The quantitative results demonstrate the advantage of the proposed work in achieving behavioural stability in the long-run. This advantage is accompanied with computational overhead.

Related contributions include:

- *Systematic literature review on self-awareness in software engineering.* Grounded by the findings of the stability survey, novel extensions for designing stable architectures and more intelligence techniques for realising stability are needed. As such, the latest emerging *self-* property has been investigated for its computational intelligence primitives. The findings of the systematic literature review have revealed its effectiveness for stability, yet no explicit attempt has considered self-awareness in achieving stabilisation. This partially contributes to *RQ4*. A summary of the review findings is presented in Appendix B.

- *Systematic mapping study on managing trade-offs in self-adaptive architectures.* As the stability analysis revealed contradicting stability attributes, their trade-offs should be handled in a sensible way when making design and runtime decisions. Trade-off management has become a non-trivial and challenging issue during runtime operation. In addressing this challenge, a systematic mapping study was conducted to identify and analyse research work that explicitly addressed trade-off management for self-adaptive software architectures. The findings call for foundational work to analyse and manage trade-offs that can explicitly consider specific multiple quality attributes, the runtime dynamics, the uncertainty of the environment and the complex challenges of modern architectures. This partially contributes to *RQ4*. A summary of the findings is presented in Appendix C.

- *Symbiotic simulation environment for self-adaptive software systems.* The stability evaluation framework indicated the need for identifying evaluation tools. Discrete-event simulations are found to be one of the feasible tools to support design decisions and runtime operation. As such, a symbiotic simulation tool is designed to support the runtime evaluation process. This contribution partially addresses *RQ4*. Details are presented in Appendix D.

1.4.3 Publications

The publications arising from this thesis are:

- M. Salama, "Stability of self-adaptive software architectures," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Doctoral Symposium*, 2015, pp. 886–889. [1]

- M. Salama and R. Bahsoon, "Quality-driven architectural patterns for self-aware cloud-based software," in *IEEE 8th International Conference on Cloud Computing (CLOUD), Applications Track (acceptance rate 14%)*, 2015, pp. 844–851. [2]

- M. Salama and R. Bahsoon, "A taxonomy for architectural stability," in *31st ACM/SIGAPP Symposium on Applied Computing (SAC), Software Architecture: Theory, Technology, and Applications Track (SATTA)*, 2016, pp. 1354–1357. [3]

- M. Salama, R. Bahsoon, and N. Bencomo, "Managing trade-offs in self-adaptive software architectures: A systematic mapping study," 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. 249–297. [4]

- 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. [5]

- A. Elhabbash, M. Salama, R. Bahsoon, and P. Tino, "Self-awareness in software engineering: A systematic literature review," *(submitted for publication)*, 2017. [6] [7]

- M. Salama, R. Bahsoon, and P. Lago, "Stability in software engineering: Survey of the state-of-the-art and research directions," *(submitted for publication)*, 2017. [8]

- M. Salama and R. Bahsoon and R. Buyya, "Modelling and simulation environment for self-adaptive and self-aware cloud architectures," *(submitted for publication)*, 2018. [9]

- M. Salama and R. Bahsoon and R. Buyya, "A reference architecture and modelling principles for architectural stability based on self-awareness: Case of cloud architectures," *(submitted for publication)*, 2018. [10]

[1] This publication is part of Chapter 1.
[2] This publication is part of Chapter 6.
[3] This publication is part of Chapter 2.
[4] This publication is part of Chapter 7 and Appendix C.
[5] This publication is part of Chapter 4 and 5.
[6] This publication is part of Chapter 7 and Appendix B.
[7] This paper is equal contribution of the first two authors. The first author focused on the engineering practices and the second author focused on self-awareness concepts to assess their feasibility for engineering stability.
[8] This publication is part of Chapter 8.
[9] This publication is part of Chapter 8 and Appendix D.
[10] This publication is part of Chapter 6.

- M. Salama and R. Bahsoon and R. Buyya, "Architectural stability reasoning using self-awareness principles: Case of self-adaptive cloud architectures," *(submitted for publication)*, 2018. [1]

- M. Salama, R. Bahsoon, and P. Lago, "A framework for evaluating architectural stability," *(submitted for publication)*, 2018. [2]

1.5 Organisation of the Thesis

The rest of the thesis is structured as follows.

- **Chapter 2** reviews the state-of-the-art related to stability in software engineering. The aim is to present the required background and understanding of the notion, explore current engineering practices and identify gaps in the literature. This chapter is partially derived from [46], [42].

- **Chapter 3** discusses characterisation and engineering stability from a multi-dimensional perspective, based on the taxonomy proposed in the previous chapter. This chapter also discusses design concepts for capturing the intended behaviour. This chapter is partially derived from [46], [40].

- **Chapter 4** proposes a novel methodology for analysing architectural stability. The analysis model aims to capture stability dimensions, stakeholders' concerns for stability and related attributes. Representing stability attributes and their dependencies, the analysis supports understanding the intended behaviour. This chapter is partially derived from [44].

- **Chapter 5** proposes a methodology for modelling the architecture's intended behaviour. The modelling aims to support the control design principles, by understanding of the expected behaviour in comparison with the desired behaviour. The modelling accumulates the knowledge and performs runtime inference for reasoning about the architecture's behaviour on the long-run. This chapter is partially derived from [44].

- **Chapter 6** proposes design-support principles for stability. The design artefacts include a reference architecture and goals modelling capable of efficiently achieving stability objectives. The main purpose is to facilitate and guide the design of stable architectures for new systems and the improvement of developed systems with architectural stability. This chapter is partially derived from [48], [41].

- **Chapter 7** extends the reference architecture by implementing computational intelligence techniques in different self-awareness components. The proposed work includes algorithms for realising symbiotic relation between goal-awareness and runtime goals model, online learning technique and trade-offs management using stochastic games. This chapter is partially derived from [49], [45], [43].

[1] This publication is part of Chapter 7.
[2] This publication is part of Chapter 8.

- **Chapter 8** proposes a framework for conducting architectural stability evaluations. Architectural stability evaluation aims at enhancing design-time decisions and run-time operation, delaying the architecture drifting and phasing-out as a consequence of the continuous unsuccessful provision of quality requirements. This chapter is partially derived from [50], [47].
- **Chapter 9** summarises and evaluates the thesis contributions. Potential threats to validity related to the proposed work are discussed, as well as future work and possible extensions.

Chapter 2

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

We know next to nothing about virtually everything. It is not necessary to know the origin of the universe; it is necessary to want to know. Civilization depends not on any particular knowledge, but on the disposition to crave knowledge.

— George Will

2.1 Introduction

This chapter introduces background on the basic concepts adopted throughout the thesis, as well as self-adaptivity and self-awareness in software engineering. Then, the chapter reports on a systematic literature review on stability in software systems engineering. The survey aims to provide a comprehensive overview of the current state-of-the-art and connect knowledge on stability and related properties in software engineering.

Survey Approach. 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. 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.

Aims of the Survey. 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. The contributions of this chapter 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 to shed its relationship with other quality attributes and software engineering practices.
- an overview of the current state-of-the-art related to stability of the different software artefacts.
- a review of the software engineering practices supporting architectural stability.
- an analysis of current research gaps with respect to stability as a software property.

Organisation. This chapter is organised as follows. In section 2.2, we present the basic concepts. In section 2.3, we present background underlying the notion of stability, and briefly describe the survey method in section 2.4. In section 2.5, we present a taxonomy for characterising stability, and analyse the concept definitions in section 2.6. In section 2.7, we review the treatment of stability in software engineering, and in section 2.8, we present the research findings related to architectural stability. Gap analysis and related surveys are discussed in sections 2.9 and 2.10 respectively. The chapter is concluded in section 2.11.

2.2 Background

Before presenting our survey, we introduce the basic concepts and terms. To this aim, the following sections introduce the basic concepts (section 2.2.1) and the self-adaptive architecture domain (section 2.2.2).

2.2.1 Preliminaries and Basic Concepts

Software System. A software system comprises a set of software components, computer programs, procedures, rules (and possibly associated documentation and data) pertaining to the operation of a computing system or an information processing system that

satisfies an end-use function [51]. The *system boundary* is the common frontier between the system and its operating environment. The *function* of the system is "what the system is intended to do" [52]. The *behaviour* of such a system is "what the system does to implement its function" [52]. The *service* delivered by a system (in its role as a provider) is its behaviour as it is perceived by end-user(s) [52].

Software Lifecycle. The life cycle of a software system consists basically of the *development* and *operation* phases [52]. The development phase includes all activities to the decision that the software is ready for operation to deliver service, such as requirements elicitation, conceptual design, architectural design, implementation and testing [52]. 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 [52] [51]. 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 different cycles of maintenance and evolution stages till reaching the phase-out and close-down [15] [12] [52] [51]. During a maintenance cycle, minor defects are repaired, while the system functionalities and capabilities are extended in major ways in an evolution cycle [12].

Quality Attribute. We use the definition of the IEEE Standard for Software Quality Metrics [53], 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" [53].

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" [51]. This is determined by the state-changing operations the system can perform [51].

Software Architecture. The concept of software architecture has been defined in different ways in 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" [51]. This definition is in line with early [54] [55] and later definitions [56]. Software architectures provide abstractions for representing the structure, behaviour and key properties of a software system [55]. 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 [57] [58].

Architectural Structure. Architectural structure is "a physical or logical layout of the components of a system design and their internal and external connections" [51].

Architectural Style. An 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 [51] [58] [59]. An architectural pattern is described by its structure (what are the components) and its behaviour (how they interact) [60]. 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) [61] [62]. Functional requirements are implemented by the individual components, while the quality requirements are highly dependent on the organisation and communication of these components [63]. In this context, it is worth mentioning that we focus on the *architecturally-significant requirements*, as not all requirements have an equal effect on the architecture [64]. This special category of requirements, describing the key behaviours that the system should perform, plays the main role in taking architectural decisions and has a measurable effect on the software architecture [56]. 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 [64]. 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" [51]. The output of the architectural design phase is an architectural model that describes how the system is organised as a set of communicating components [63].

2.2.2 Self-Adaptive Software Architectures

Self-adaptivity is engineered to achieve the level of dynamicity and scalability necessary for modern and complex software systems, as well as to comply with the changes in components, fluctuations in workloads, and environmental conditions during runtime [21] [22] [23]. A self-adaptive software "evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible" [65] [24] [22]. Intuitively, a self-adaptive system is one that has the capability of modifying its behaviour at runtime in response to changes in the dynamics of the environment (e.g. workload) and disturbances to achieve its goals (e.g. quality requirements) [66]. Self-adaptive systems are composed

of two sub-systems: (i) the managed system (i.e. the system to be controlled), and (ii) the adaptation controller (the managing system) [25]. The managed system structure could be either a non-modifiable structure or modifiable structure with/without reflection capabilities (e.g. reconfigurable software components architecture) [25]. The controller's structure is a variation of the MAPE-K loop (Monitoring, Analysis, Planning, Execution - Knowledge) [25].

Self-adaptive architectures are expected to manage themselves following the principles of autonomic computing, to respond to environmental changes and prevent service provision violations [24]. Examples of adaptation strategies include architectural tactics, as mechanisms for better tuning, responding and achieving Quality of Service (QoS) attributes, such as response time, throughput, energy efficiency. Architectural tactics are inherently architectural decisions, with a measurable response, designed to support quality attributes subject of interest [56] [67]. For instance, tactics are designed for performance, greenability, availability, and reliability, e.g. horizontal scaling, vertical scaling and VMs consolidation [56] [68].

2.3 The Notion of Stability

The Latin origin "stabilitas" refers to both *firmness* and *steadfastness* [69]. 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" [69]. 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 the stable condition is, 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" [69] [70]. 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 [71] [72]. The *evolutionary* and *dynamic* stability have also been introduced in biology [73]. 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" [74].

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" [69] [70].

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" [69]. The basic definition is that "a bounded input produces a bounded response" [75]. Several notions of stability have been introduced in this area, such as Poisson, structural, exponential and asymptotic stability [76]. 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 [77].

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 [1] [78] [79]. 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 [1] [78] [79]. According to the classical notion of Lyapunov stability, small perturbations to the initial state of the system will affect its behaviour in small variations [1]. Intuitively, in control theory, a stable system is one that, "when perturbed from an equilibrium state, will tend to return to that equilibrium state" [77] [1]. In Optimal Control Theory [80], stability refers to "the continuous behaviour of optimal solutions under perturbations of the problem data", where bounded disruptions have bounded effects [81].

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" [75]. 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) [82], and would return the system to an equilibrium state following a perturbation [75].

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 [82]. The proposed scheduling algorithms have demonstrated that handling stability is subject to the algorithm and environment under consideration (e.g. real-time environment) [82]. 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) [75]. 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" [75].

Self-stabilisation was initially introduced by Dijkstra in the context of robust distributed algorithms [83]. This property ensures that the system autonomously recovers

and converges to legitimate behaviour in a finite time after any transient fault [83] [84] [85] [86]. Since Dijkstra's seminal work, recent work by Dolev et al. [87] [88] [89] [90] 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 [87] 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 [87]. The study of Schmid [91] 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.

2.4 The Survey Method

To identify the literature related to stability in software engineering, we conducted the survey following the guidelines of systematic literature reviews [92] [93]. The survey aims to address the following questions: (i) how stability can be defined and characterised as a software property? (ii) what is the current state of research on software stability? and (iii) 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 the first question is to identify these definitions, with the goal of getting a sound definition and characterisation of this quality property. The second question aims to provide the current state of research on software stability. In the third question, 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 new software paradigms and their dynamics, as well as 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 [94] –was used to complement the search process. As a result, we identified a set of 166 primary studies. The review protocol appears in Appendix A.

In the remainder of this chapter, we first present the taxonomy (section 2.5) defined to guide the review. We present the findings of the questions in section 2.6, 2.7 and 2.8 respectively.

2.5 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 [95] [96]. A structured representation for 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 [96]. Taxonomies are found essential to document and accumulate knowledge of software engineering phenomena too [97].

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.

Dimensions 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) [98] [21] [99] [100]. 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 2.1 gives an overview of the proposed taxonomy. Table 2.1 shows how the taxonomy answers the questions. The answers 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. A mapping between the data extracted from primary studies and the taxonomy dimensions is shown in Appendix A. The dimensions of the taxonomy and the resulting characterisation of stability are described below.

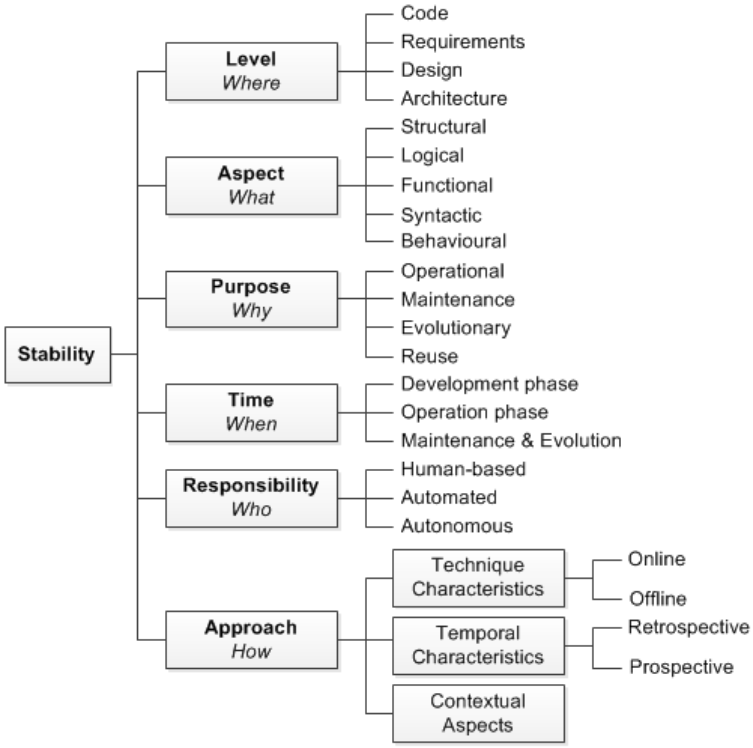

Figure 2.1: Taxonomy of Stability as a Software Property

Table 2.1: 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

- *Level (Where).* This dimension is concerned with 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 or requirements. In order to realise stability as a software property, it should be aimed at the different soft-

ware artefacts. This dimension sets out to locate the level at which this property is considered, which in turn determines *who* is responsible for 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 the literature, different aspects have been considered, varying between the physical aspect (related to equipment and physical resources, e.g. malfunction of a physical machine may put the system into an unstable state), structural (structure of the artefact) and 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 stability of the artefact throughout long-term modifications during software evolution, that is considering stability for an 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 detail, 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 [101].

- *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.

Discussion. 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 [30]. Technical and social aspects of the development process should also be considered. Knowledge (design, requirements and architecture knowledge) management, documentation and drift are influencing factors in extending the software longevity [102] [103] [104] [13]. The well-being of the development community highly affects the software engineering process [105]. Examples include changing organisational structure, the social communities in development [106] [107] [108] [109]. Another example is the stability of the maintenance process itself which has proven to affect the reliability of the maintained software [110]. 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.

2.6 Defining and Characterising Stability

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

2.6.1 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 (see Table 2.2). 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).

Table 2.2: Definitions of Stability in Software Engineering Literature[1]

Ref.	Definition	Dimensions of Stability		
		Level	Aspect	Purpose
[111]	"the resistance to the amplification/propagation of changes that has been made to a given program"	C	-	Mnt
[112]	"the resistance to the potential ripple effect that the program would have when it is modified"	C	St	Mnt
[112], [113]	"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
[112], [113]	"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
[114]	"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
[115]	"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	-	-
[116]	"the capability of the software product to avoid unexpected effects from modifications of the software"	-	-	-
[116]	Stability "characterises the sensitivity to change of a given system that is the negative impact that may be caused by system changes"	-	-	-
[117]	"a measure of how well it accommodates the evolution of the system without requiring changes to the architecture"	A	St	Ev
[118], [119]	"the ease with which a software system or a component can evolve while preserving its design as much as possible"	D	St	Ev
[120], [121], [122], [123], [19], [124], [125], [39]	"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
[126]	"the resistance to interclass propagation of changes that the design would have when it is modified"	D	L	Mnt
[127]	"the ease with which a software item can evolve while preserving its design"	D	St, L	Ev
[128]	"the degree of modification of the component"	C	-	Re
[129]	"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

Table 2.2 (*cont.*)

Ref.	Definition	Dimensions of Stability		
		Lev.	**Asp.**	**Purp.**
[130]	"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
[70]	"the ability of a software artefact to keep unchanged along with the time"	-	-	Ev
[70]	"the ability to adapt to changes by its flexible configuration mechanism"	-	-	Op
[70]	"the probability that a business model or a component remains stable in a given period of time"	-	St	Ev
[131]	"Design stability encompasses the sustenance of system modularity properties and the absence of ripple-effects in the presence of change"	D	St	Ev
[132]	"how easy or difficult is it to keep the system in a consistent state during modification?"	C	-	Mnt
[133], [134]	"the ability of the high-level design units to sustain their modularity properties and not succumb to modifications"	A	St	Mnt
[135]	"how well does the system avoid unexpected effects after a modification"	C	-	Mnt
[136]	"the ability of a module to remain largely unchanged when faced with newer requirements and/or changes in the environment"	C	Sy, St	Ev
[16], [137]	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
[138]	"robustness against input or code perturbations"	C	B	Op
[139]	"a quality characteristic that shows a software product's resilience to changes in the original requirements of the product"	-	-	Mnt, Ev
[139]	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
[140]	"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

[1]Within this table, we used the following abbreviations: C = code; D = design; A = architecture; R = requirements; St = structural; L = logical; F = functional; Sy = syntactic; B = behavioural; Op = operational; Mnt = maintenance; Ev = evolutionary; Re = reuse.

Analysing the definitions found in the literature, they partially covered the different abilities of stability (cf. section 2.3): (i) the ability to resist to (ripple effect of) changes (e.g. the definitions of [111], [112], [113], [139]), (ii) the ability to remain (largely) unchanged over time or when external conditions change (e.g. [114], [127], [129]), (iii) the ability to adapt to changes while remaining intact (to a big extent) (e.g [117], [118], [122], [136]), (iv) the ability to return to equilibrium state (within a defined time) when perturbed from that state ([132]), and (v) the ability to maintain a stated property value or fixed level of operation (within specified limits) under varying external conditions ([133], [138]). Some definitions encompass more than one ability, such as the definitions of [133] [134] covered the abilities to resist to changes and to remain unchanged.

Table 2.3 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. [128]) and precision (e.g. [117] [120]). Also, stability was defined in different ways by the same authors according to the perspective of their study, such as [112] and [114] for maintenance and evolutionary purposes, respectively.

Table 2.3: Mapping of Stability Notion, Taxonomy Dimensions and Definitions

Ability	**Purpose of Stability**	**Definitions**
(i) ability to resist to ripple effect of changes	Mnt, Ev	[111], [112], [113], [139]
(ii) ability to remain largely unchanged over time	Mnt, Ev	[114], [127], [129]
(iii) ability to adapt to changes while remaining intact	Op, Mnt, Ev	[117], [118], [122], [136]
(iv) the ability to return to equilibrium state when perturbed from that state	Op, Mnt, Ev	[132]
(v) the ability to maintain a stated property value or fixed level of operation within specified limits under varying external conditions	Op	[133], [138]

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. [111], [112]). 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.

2.6.2 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. The related quality attributes have been extracted from the primary studies, but when definitions were not found in the primary studies, we conducted separate searches to find how these concepts are defined and related to stability. Table 2.4 presents and compares quality attributes inter-related with stability and the respective dimension of stability.

Resilience. Resilience is defined as "the ability to successfully accommodate unforeseen environmental perturbations or disturbances" [141]. 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" [148], and "the persistence of the avoidance of failures that are unacceptably frequent or severe, when facing changes" [141] [142] [143] [29]. 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 [158] [141], 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" [141] [142] [143] [29]. 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" [144].

Trustworthiness. Trustworthiness is the "assurance that a system will perform as expected" [52]. Trustworthiness 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 [145]. A system is considered robust if it "retains its ability to deliver

Table 2.4: 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	[141], [142], [143], [144], [29]	Op, Ev
Trustworthiness	ability to perform as expected	dependability, reliability	[52]	Op
Robustness	ability to operate beyond normal operational conditions	resilience, dependability, reliability	[145], [146], [141]	Op
Reliability	ability to be available when required and behave as expected/ accept corrective actions effectively	dependability, fault-tolerance, maintainability	[116], [147]	Op, Mnt
Dependability	ability to deliver justifiably trusted services in spite of continuous changes	trustworthiness, reliability	[52], [141], [148]	Op
Maintainability	capability to be modified	modifiability, changeability	[116], [149]	Mnt
Modifiability	ability to make changes quickly and cost-effectively	maintainability, changeability, flexibility	[116], [150]	Mnt, Ev, Re
Changeability	ability to enable implementation of modifications	maintainability, modifiability, flexibility	[116]	Mnt, Ev
Flexibility	ability to be modified for use beyond the original design with acceptable effort	maintainability, modifiability, changeability	[51], [151]	Ev, Re
Adaptability	capacity to adjust to changes in the environment	sustainability, dependability, trustworthiness	[152], [153]	Op, Mnt, Ev
Evolvability	capacity to support adaptation and accommodate future changes in requirements in the long-term	sustainability	[154], [155]	Ev
Sustainability	capacity to preserve the function over an extended period of time and to be cost-effectively maintained and evolved	evolvability	[156], [157]	Op, Mnt, Ev

service in conditions which are beyond its normal domain of operation" [146] [141]. 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" [159], that could be behavioural stability.

Reliability. Reliability has earlier been concerned with "how well the software meets the requirements of the customer" [160] [7]. 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" [116] for a specified time [51]. According to the seminal work on the taxonomy of dependable and secure computing [52], 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 [7]. The previous definitions are interconnected with the stability abilities for operational purposes. According to the latest IEEE Recommended Practice on Software Reliability [147], code stability and release stability are considered measures of the 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 the maintenance setting.

Dependability. Dependability has been considered as "the ability of a system to provide dependable services in terms of availability, responsiveness and reliability" [148]. A widely adopted definition is "the ability to deliver services that can justifiably be trusted in spite of continuous changes" [52] [141]. 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" [52] [141]. The dependability attribute abstractly encompasses the trustworthiness attribute [161]. 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 [116], maintainability is one of the main characteristics of software, defined according to the standards as "the capability of the software product to be modified" [116]. It is divided into a set of attributes related to the ability to make specified modifications (analysability, changeability, testability and stability) [116]. Modifications may include corrections to handle errors, improvements or adaptations in response to changes in the environment and functional requirements [149]. 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

[150] [162]. 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 [150], while it was identified as one of the main quality characteristics in the ISO/IEC 9126 standards for software quality model [116]. 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 [116] as "the capability of the software product to enable a specified modification to be implemented". According to this standard [116], changeability reflects the ability of the software artefact to accept possible future changes, while stability is observed after the change has taken place [163].

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" [51]. Flexibility is mainly about future changes of software and is considered relative to these expected changes, similar to modifiability [151]. 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" [151].

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) [152] [153] [164] [165]. 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" [51]. 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 [154] [155]. The continuously changing stakeholders' requirements make evolvability an important software property to be explicitly addressed throughout the system's lifespan [155]. This property focuses mainly on the long-term evolutionary properties and changes without becoming progressively less useful [154].

Sustainability. Sustainability is defined as "the capacity to endure and preserve the function of a system over an extended period of time" [157]. Tough the concept of sustainability has usually been considered in the sense of green computing and associated with the ecological environment [8] as the capability of meeting the present needs without compromising future needs [156], a modern vision according to Lago et al. [157] 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" [156] and architectural sustainability is "the set of factors that promote an architecture's stability and longevity during system evolution" [13].

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 [52], trustworthiness in delivering confidential and trusted services, and resilience to security attacks and correlated faults [166] [148], or failures during operations [167]. 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 2.4 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 [142] [168]. Another example is dependability and resilience. While dependability is considered design-time attribute that deals with possible faults [143], as well as runtime attribute as previously mentioned in definitions [52] [141], resilience by definition is maintaining the same properties if evolution takes place in environmental factors [143]. Thus, resilience "encompasses the ability to resist and recover from changed environment, operational domains or requirements unknown at design-time" [143], 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 mean to achieve stability (such as flexibility), or an indicator for stability (e.g evolvability). As Bass et al. pointed out [4], these terms can be confusing and less meaningful without a concrete scenario, which is the approach we will adopt in defining and realising stability.

2.6.3 Related Software Engineering Practices

As mentioned above that related software quality attributes could be a mean or an indicator for 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 [112] [169] [6]. Stability is used in the maintenance process to indicate the accounting of ripple effects as a consequence of modifications [112]. Stability measures are used in conjunction with other factors affecting the maintenance process, to estimate maintenance costs and possible errors when generating maintenance plans [112].

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 [117] [122]. Such decisions are taken based in the long-term impact on stability when planning for possible evolution paths or in automated planning for evolution [170] [171]. According to the Lehman's laws of software evolution [172], 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 [173] [174], with a number of visible signs, such as performance degradation, design degradation, or quality reduction [175] [173]. It has been attributed in many ways [176], such as architectural/ design erosion and architectural drift [54] [177]. 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" [54]. 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 [173] [176]. The challenge is, then, to keep the architecture or design aligned throughout the system lifetime [178], 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 [175].

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 [37]. The purpose is to increase productivity and software reliability, as well as reduce development cost and time [37]. Stability is an important factor to consider both when building software artefacts to be reused later and when selecting the reusable artefact [179].

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 [180]. 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 [181]. 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 [21] [99] [182]. As inspirations were drawn from Control Theory in building adaptive systems, stability has been suggested as primary criteria for evaluation [66] [25]. 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" [25], i.e. the controlled variables are within a required range. This is characterised as the stability of the adaptation goal [25]. 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 [25] [26]. Even though adaptation mechanisms have been widely investigated, stability was not explicitly tackled [25]. 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 [182]. Besides, the adaptation process does not address the adaptation properties that affect the quality of adaptation, such as accuracy, settling time and resources overshoot [25] [26].

2.7 Stability in Software Engineering

In this section, we review stability in software engineering. First, the primary studies are classified and analysed based on the taxonomy described in section 2.5. 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.

2.7.1 Analysis Results of Primary Studies

The demographic and quantitative analysis results are shown in Appendix A. Below, we present analysis results related to correlating different stability dimensions and architectural stability.

2.7.1.1 Correlating Stability Dimensions

To analyse the current research state, we used the levels, aspects and purposes for stability as crosscutting dimensions, as shown in Figure 2.2. 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.

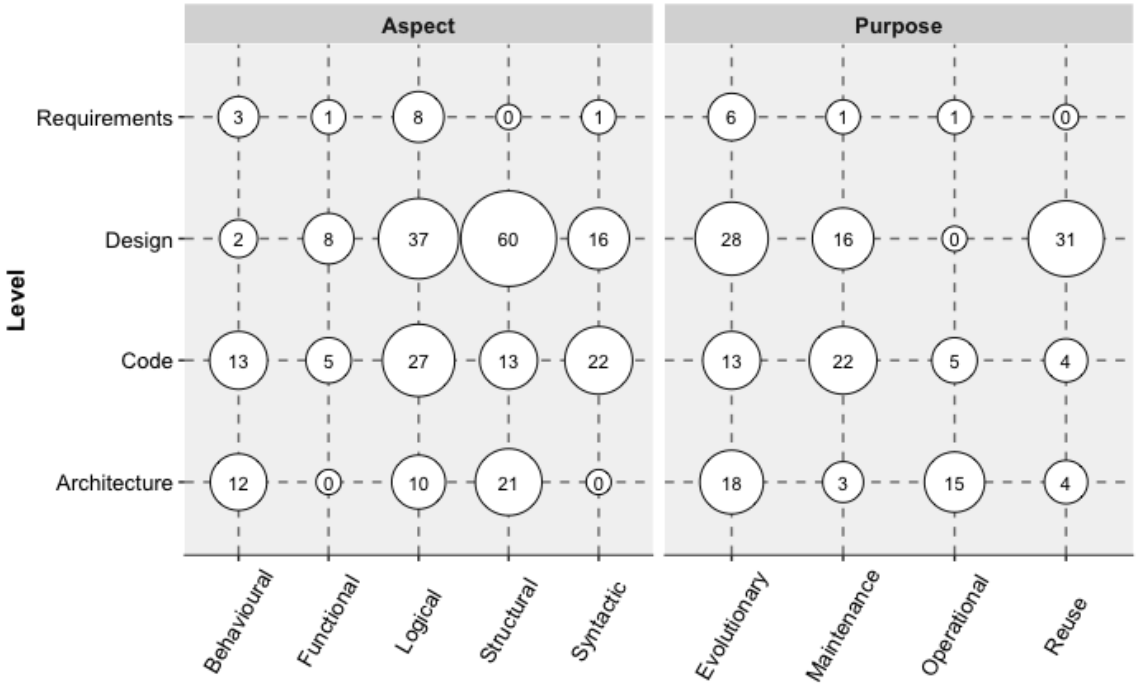

Figure 2.2: Correlation of Stability Levels, Aspects and Purposes

Correlating the different levels of stability and the phase when stability is considered (Figure 2.3), 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.

2.7.1.2 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 2.4. The references in each circle represent the studies related to both dimensions for different stability aspects, with the total number of studies appearing

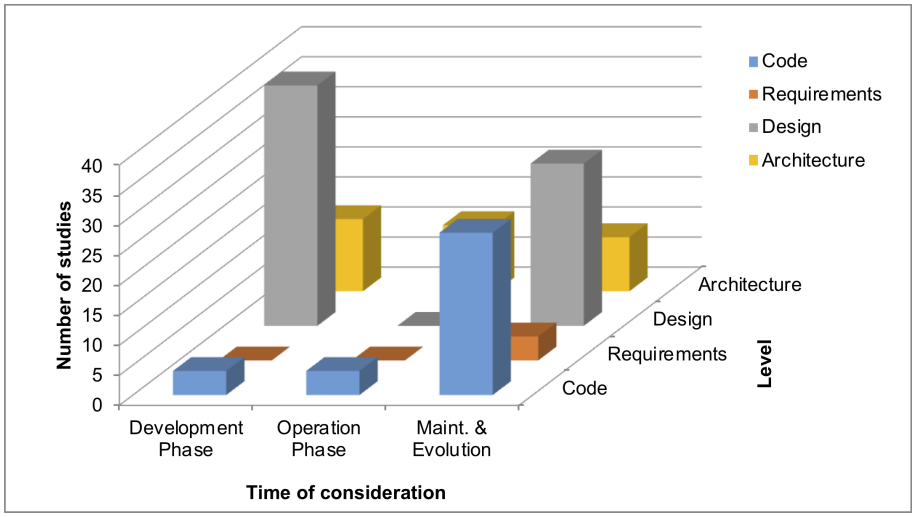

Figure 2.3: Correlation of Stability Levels and Time of Consideration

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 are discussed in section 2.8.

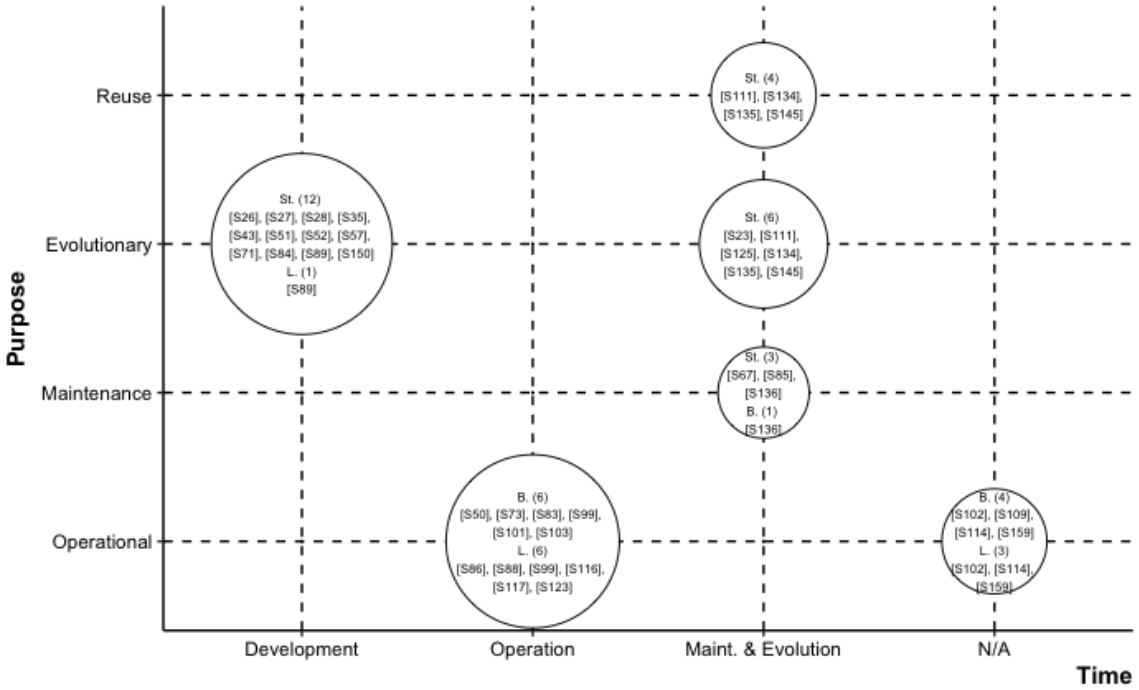

Figure 2.4: Systematic Map of Stability at the Architecture Level

2.7.2 Levels, Aspects and Purposes of Stability

In the Software Engineering discipline, explicit discussions about stability are traced back to 1977, where Soong [111] 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.

2.7.2.1 Code Level

Maintenance purpose. The earliest mention of stability is found at the code level of software programs [111], 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 [111].

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" [112]. 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 [112] [113].

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 [111] and [112] 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 [183]. The studies of Black [184] [185] 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 [186] 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 [132] have studied the mapping between stability —as one of the sub-characteristics of maintainability according to [116] —and code level properties (such as size of the system, duplication of code, unit complexity,

unit length, number of units and number of modules). Meanwhile, the survey of [135] 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. [187] have considered mapping source code metrics and maintainability sub-characteristics of the ISO standards [116]. Stability has been related to the unit interfacing and module coupling properties. The study of [188] 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 [189] 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 [190] 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 cloned code is found on average older than non-cloned code (not changed for longer) [191]. The studies [192] [193] 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 [194] [195] [196] 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 [197] 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 [198] 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 development, Bogart et al. [199] discussed the need for an awareness mechanism based on different stability indicators, such as historical and contextual, to assist developers in analysing stability of the changed code and evaluating the impact of changes.

Evolutionary purpose. In considering stability for *evolutionary* purpose, Li et al. [200] 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 [201], 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 [136], 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 [202] 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 [203] for measuring evolutionary stability of software artefacts at the binary level, such as version stability, branch stability, structure stability.

Stability has been studied as application- and domain-specific. Hou and Yao [204] 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. [205] 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 the 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 [149]. 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*) [206] [207].

The *functional* aspect of stability has been studied in [208] 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.

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" [16] [137], while it is reused in a project "if it is used in more than one context within the software system" [16]. In the exploratory study [16], 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 [137] focused on analysing the impact of code modularity and

composition on stability and reuse. The study of [128] 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.

Operational purpose. According to the latest IEEE Recommended Practice on Software Reliability [147], 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 from 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 [209]. 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 [209].

In the domain of multi-agent systems, Bracciali et al. [210] 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. [211] 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 [212]. 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 [138] to make threads stable during runtime, by allowing developers to write performance hints in their code for changing schedules when default ones are slow.

2.7.2.2 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 [213] [214] [215] [216]. According to the IEEE Recommended Practice for Software Requirements Specifications [115], 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 of stability requirements was found in [217], 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 [217]. Yet, the earliest and simplest measurement of require-

ments stability has been performed using the following equation [218]:

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

But this equation does not consider the changes occurring to requirements. The factors affecting the requirements stability have been analysed in [219], 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.

Evolutionary Purpose. According to the IEEE Standard of Measures to Produce Reliable Software [220] [221], the Requirements Maturity Index (RMI) has used changes from a previous release relatively to the current release as an indication of stability [222] [223]. These *retrospective* measurements are assessing requirements stability for *evolutionary* purpose.

The RMI was also used for estimating requirements maturity during the development phase [224] [225]. 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 [226] [227] 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).

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 [163]. The study provided evidence that certain crosscutting properties have negative effect on stability.

Operational Purpose. Focusing on the quality requirements during runtime, the study of [228] 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.

2.7.2.3 Design Level

Evolutionary Purpose. Stability has been widely studied as a design characteristic. The earliest research is the study of [114], 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" [114]. Kelly [130] 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. [229] have considered a system stable with respect to a set of anticipated evolutionary changes, where a bounded input (i.e. set of anticipated changes) should result in a bounded output (impact on the system). Meanwhile, authors in [230] 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 [231] 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 [181].

In the context of object-oriented design, Bansiya [232] [233] 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 [234] [235] extended Bansiya's work with an additional aggregated metric, which is the "relative-extent-of-change" metric. Grosser et al. [118] [119] 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. [236] 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. [200], 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 [237] [238] [239] for object-oriented systems developed using the agile software process. The work of Azar and Vybihal [240] 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 [241]. Meanwhile, Bouktif et al. [127] 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 [139], which allowed interpretations of the class stability.

In the context of aspect-oriented design, Greenwood et al. [131] 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 [242] have studied the aspect- and object-oriented versions of three evolving systems. Meanwhile, the *predictive* approach of [243] studied the correlation between crosscutting concerns and design (in)stability, in order to anticipate design decisions at early stages of software development.

Maintenance Purpose. The *prospective* approach of Yau and Collofello [114] 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 [126] [129] have adopted the same perspective of [114]. In [126], 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 [129], 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 [244], but no empirical evidence was provided. The relation between class stability (using the previous metrics) and maintainability has been experimentally studied in [245].

In the context of design patterns, the work of [140] 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 [238]) and software maintainability [246]. Using one metric for stability, the experiments showed variations in the correlation, but no direct causality was concluded.

Reuse Purpose. In the column series that appeared in the Communications of the ACM by Fayad [247] [248] [249], the concept of Enduring Business Themes (EBTs) [250] 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 [249]. By dividing the system into stable and unstable modules, Chiang [251] 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 [9]. The SSM has been employed in the context of software *reuse* to describe the core of a system, which generates a stable design that is extensible for software reuse [252] [253]. An implementation method for the BOs was proposed in [254] 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 [255], magnetic resonance image (MRI) visual analyser stable application [256], and for realising unified software engineering reuse [257].

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 [258] [259]. 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) [260], the classification pattern [261], AnyLog pattern [262], AnyTransaction pattern [263], AnyInformationHiding pattern [264], AnyCorrectiveAction pattern [265], the learning pattern [266], and the reputation analysis pattern [267].

Another stability pattern, called Stable Atomic Knowledge (SAK) pattern, has been introduced for representing domain-neutral knowledge to be reused in different domains [268]. 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 [269]. Also, an approach for identifying and reusing domain patterns has been proposed in [270]. A domain analysis method driven by stability has been proposed in [271], 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 [272]; and a semi-automated approach was proposed in [273]. Meanwhile, authors in [274] 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 [275].

In the context of component-based design, the correlation between the stability of

domain business models and components granularity (*structural*) has been studied in [70], 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 [276] [277]. 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 [278], 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 [279] [280]. 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 [281]. Stability-based adaptation aims at generating an adapter with the smallest bound satisfying stability [281].

Syntactic Aspect. Different aspects of stability for object-oriented design have been studied. The earliest studied is the *syntactic* aspect in [282], 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 [283] as factors for stability influencing software *maintenance* and *reuse*. Based on the number of methods in a class, Rapu et al. [284] 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. [285] 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.

Structural Aspect. The *structural* aspect of object-oriented design has been studied in [286], 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 [287], 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 about the correctness of this assumption

on Java interfaces has been conducted in [288]. The work of [289] 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 [290] 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 [14] [291] has aggregated the package level stability for measuring the structural design stability of open source systems.

Metrics. A metrics suite has been proposed in [292] 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.

2.7.2.4 Architecture Level

Evolutionary Purpose. Stability has been considered as the main criterion for assessing the long-term value of software architectures throughout their evolution [123]. The earliest discussion about stability at the architectural level is found in [117]. 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" [117]. The *retrospective* approach proposed by Jazayeri [117] aims at analysing how easily the evolution occurred over the successive releases of the software. Meanwhile, the *prospective* approach [39] 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" [122]. Tonu et al. [293] 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 [122], Aversano et al. [294] 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 [239]), 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 [179], 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 [295].

Also, an approach based on concern traces has been proposed for assessing and pre-

dicting architecture design (in)stability using information about early development stages for sustaining the architecture throughout the system evolution [296]. The effectiveness of concern assessment mechanisms to predict architecture (in)stability has been also studied in evolving architectures [297].

Nord et al. [298] 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.

Reuse Purpose. The stability metrics of [179] have been extended by [299] 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.

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" [133] [134]. 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 [133] [134].

Stability of architectural tactics, essential for realising architectural qualities and meeting quality requirements, has been studied in [300]. 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.

Operational Purpose. In the context of embedded systems, Rafiliu et al. [301] [302] 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. [303] 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 [304]. Meanwhile, authors in [305] have discussed a layered architecture for embedded systems capable of self-stabilising 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. [159] 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 [11] 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 [306] 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 an online statistical method. Focusing on the dynamic learning behaviour during runtime operation of adaptive systems, Yerramalla et al. [307] 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 [308], 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" [309]. A stable adaptation strategy is the one that "avoids oscillating behaviours and minimises the number of reconfigurations" (i.e. avoid unnecessary modifications) [310]. That is quantitatively measured by the total number of reconfigurations and the average time between consecutive reconfigurations [311]. The control-theoretic approach and adaptation strategies proposed in [311] [309] 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 reconfiguration 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 [312] 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.

2.7.3 Main Observations and Findings

Dimensions of Stability. As mentioned in section 2.6, 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 2.3). Meanwhile,

there exist a smaller number of studies focusing on the operational- and behavioural-related abilities (i.e. return to an 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 to 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 the 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 [112], where the changed program and its modules are compared, during maintenance, before and after changes for determining the effect of changes on stability [184]. 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 are [112], and the latter approach is [149].

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 [115] [220] [221]. 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 [228] 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 [313].

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 [232], aspect-oriented [131]).

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. [133] [134] [295]), i.e. the ability to accommodate changes while remaining intact. It has been retrospectively analysed over two or more versions of the software ([117]) or predicted for possible future changes ([122]). 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 as adaptive distributed systems ([309]), and embedded systems ([301], [302]).

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 artefact (code or design) and measures the interdependencies between modules/components [112] [114]. Such interdependencies reflect "the degree of probability that changes made to other modules could require corresponding changes to this module" [203], 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 [130], [136]), and using structural differences at the architecture level (architecture-level metrics) [203]. Meanwhile, the two types of metrics would complement each other; architecture-level metrics would be appropriate for measuring the 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 [203].

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. [314], [315], [316], [174]). Also, application samplers and simulators are needed for studying operational and behavioural stability in practice.

2.8 Engineering Practices Supporting Architectural Stability

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.

2.8.1 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 [317] [39]. The universal philosophy "design for change" [317] has been adopted in the early efforts for designing stable architectures [39]. 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 overbalance the cost of designing for change and the cost of changes as they occur [318] [39].

By that, an economic-based approach has been adopted to develop flexible architectures that will remain stable while the requirements are changing [318]. This required linking the structural decisions to the future value creation [39]. 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 [39]. 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 [120] [121] [123]. In details, the added value was attributed to "the flexibility of the architecture in enduring changes in requirements" [19]. 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 [120] [123] [19]. 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 [39].

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) [319], Architecture Tradeoff Analysis Method (ATAM) [320] [321], Cost Benefit Analysis Method (CBAM) [322] [323], behaviour analysis [324], the 4+1 view model [325], viewpoints [326], scenario-based analysis [327]. 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 [27] [328]. Different approaches have been discussed in the literature (e.g. [27], [329], [24]) 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 [66] [25].

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).

2.8.2 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 of accommodating future changes while remaining intact [39]. 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 [124].

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 [39]. The predictive evaluation aims at supporting the process of valuing the capability of a particular architecture design relative to the future changes [124]. 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? [124].

An early survey of design-time evaluation approaches [122] 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) [330], Attribute-Based Architectural Styles (ABAS) [331], Scenario-Based Architecture Reengineering [332], Quality-Attribute-Based Economic Valuation [333], and CHARMY for verifying architectural specifications [334]. These methods focused on evaluating architectural decisions in relevance to traditional quality attributes [39]. 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 [120] [121]. 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 [318] [122]. The major

[1]Further details about the critical relation between stability and these evaluation methods could be found in [122] [39].

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 in 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 [124]. 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 to 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. [297] 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 [297]. Authors in [133] [134] 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 [66] [25]. But stability has not been explicitly considered when evaluating adaptation policies at design-time. Meanwhile, special attention has been given to evaluating the 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. [335] 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 [168] 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" [336]. With respect to the adaptation controllers, authors in [337] [338] 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 in 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 to 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 [148] identified the challenges and opportunities for provisioning dependable and resilient cloud-based software services. The work of Ghosh et al. [339] 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 [143], the impact of environmental changes on resilience was quantitatively evaluated using Exploratory Data Analysis (EDA). The works of [340] [341] 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 [168], the resilience benchmarking presented in [142] has addressed the robustness and resilience issues.

In 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 [117] has considered architectural stability for the evolutionary purpose. The retrospective analysis aims at analysing how easily the evolution occurred, by examining consecutive releases of the software [117]. 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 [19]. 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 [122] [39].

Another retrospective approach is the metric-based approach proposed in [293]. 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 [293], 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 [120] 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 evolutionary purpose. Few studies have explicitly proposed metrics for architectural stability. Constantinou and Stamelos [179] [299] 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" [75].

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 [342] [142]. Authors in [342] 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 [66], 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 [66] 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 [66] 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 [25]. In this work, 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 [25]. In this context, stability is "the degree in that the adaptation process will converge toward the control objective" [25]. An unstable adaptation will repeat the action with the risk of not improving or even degrading the system to unacceptable states [25]. 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" [25].

Jiao [152] 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 [152] 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 [39], 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. [122] [39]). 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. [117]).

Generally, the retrospective evaluation is useful for *planned evolution*, to be used for evaluating how the next release of the software will be stable [19], 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 [39]. 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. Tough 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.

2.9 Gap Analysis

Based on the taxonomy (section 2.5), we identify the research gaps. 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 2.6.1, 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 to 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 2.7, 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.

- *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, it also comprised of behavioural aspects.

- *Stability metrics.* Tough 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 of 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 benchmarks explicitly devoted to 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 [343] and resilience benchmarking [142]. It can also benefit from the structure of established benchmarks [142]. 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.* Tough there exist systematic approaches for design and evaluation of stability (e.g. [120], [299]), 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 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 the industrial context.

- *Engineering approaches considering the different aspects of architectural stability for different purposes.* From the closer analysis of the architecture level (figure 2.4), 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 [344], or the provision of specific quality attributes (e.g. dependability and reliability) [345] [346] [347] [348]. 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) [349]. Architecture reasoning and design should also explicitly consider stability when translating requirements to an architecture solution [350]. There is also 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) [351], settling time (the amount of time the controller takes to achieve the adaptation goal) [352] [353]. Yet, other properties reflecting the quality of adaptation, i.e. how well the adaptation process converges towards the adaptation goal, are not explicitly considered [354] [25], while properties reflecting the behaviour of the controller have impact on the stability of the whole architecture [25].

2.10 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 [355] [356] [357], maintainability [358], reliability [359]. Other studies reviewed related engineering practices, such as software reuse [360], self-stabilisation [361], software ageing and rejuvenation [173]. In the area of software evolution, authors in [362] presented a review on the research of architecture evolution, and [363] presented a framework for classifying and comparing architecture evolution.

With respect to software architecture, there are many surveys, such as [364] [365]. Others focused on architecture related practices, such as architecture optimization methods [366], architecture design rationale [367], decision-making techniques for architecture design [368], analysis methods [369], evaluation methods [370], and methods that handle multiple quality attributes in architecture-based self-adaptive systems [371]. 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 lifespan) [372], architectural degeneration [373] and architectural decay [374]. With respect to architecture properties, authors in [375] presented a survey on reliability and availability prediction methods for software architectures, maintainability prediction in [358], and sustainability evaluation in [376].

2.11 Summary and Conclusion

This chapter has contributed to a systematic literature review of the state-of-art on stability in software engineering and has converged to closely look at stability in software architectures. We proposed a taxonomy for characterising the concept, analysed definitions found in the literature. Such characterisation paves the way for better understanding of the concept and consequently motivates future research directions. We discussed how stability was treated for the different software artefacts in the software engineering community. As architectures have a profound effect throughout the software lifetime, we closely reviewed the related engineering practices.

As long-lived software systems are becoming highly desirable, stability is the property to reflect such concerns, hence, could be considered a primary criterion towards achieving longevity, and a fundamental property to sustain the whole system. We envision that stability could be a new dimension to software properties, as it combines many related qualities and aspects. For instance, the behavioural aspect of stability is related to performance in the long-term. 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 intact-

ness) to a multi-dimensional concept, including many aspects (structural, behavioural). This survey could serve as a primary investigation for deeply characterising the notion of stability as a software property, to take it further towards handling the wider concept and the related challenges.

Chapter 3

Characterising the Notion of Stability in Software Engineering

True stability results when presumed order and presumed disorder are balanced.

— Tom Robbins

3.1 Introduction

The literature review findings have revealed that the notion of stability has been defined and characterised in many different ways and that the different dimensions of stability are not fully considered. Grounded on such findings, this has hindered the need for proper consideration of stability as a quality attribute that is strategically important for the longevity of software systems. Hence, we contribute to a working definition and a pragmatic view of stability based on the taxonomy dimensions (proposed in section 2.5). We further discuss a proposal for engineering stability as a software property. These contribute to understanding the facets related to stability, advance the way of understanding the concept, and presents a compilation of different understandings from the literature. As such, we identify new requirements and challenges that have been imposed for realising architectural stability. Focusing on the behavioural aspect of stability, we draw conceptual designing principles for capturing the intended behaviour.

Organisation. This chapter is organised as follows. Section 3.2 presents a working definition for stability. Section 3.3 discusses a multi-dimensional perspective for characterising and engineering stability as a software property. In section 3.4, we elaborate requirements for realising architectural stability. Section 3.5 sketches design principles for capturing behavioural stability. The chapter is concluded in Section 3.6.

3.2 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. Given the multi-dimensional, case- and context-specific nature of stability, we argue that a unique definition for 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) and the different dimensions of the taxonomy (level, aspect, purpose). 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 to 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 of keeping 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).

3.3 A Multi-Dimensional Perspective for Characterising Stability

In this section, we discuss our multi-dimensional perspective for characterising and engineering stability as a software property.

3.3.1 Dimensions of Stability

While we believe the literature covered important aspects of stability which are scattered, we argue that a multi-dimensional pragmatic view is required to offer a systematic way for practitioners and researchers to deal with stability as a software property. We integrated the taxonomy dimensions (section 2.5) to create a pragmatic view for stability, as shown in Figure 3.1.

Considering this view 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.

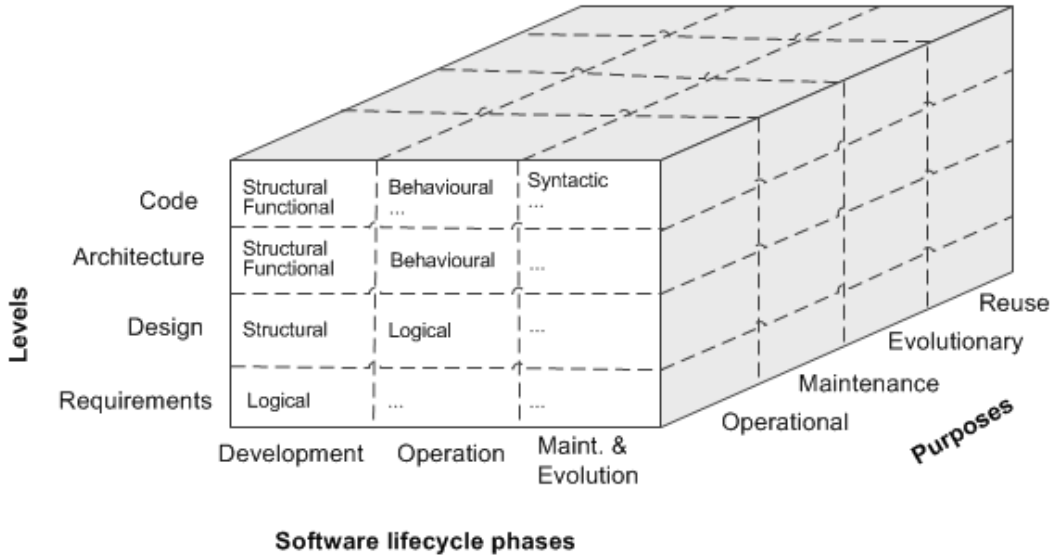

Figure 3.1: Dimensions of Stability as a Software Property

3.3.2 Engineering Stability as a Software Property

Our engineering proposal, illustrated in Figure 3.2, is based on the proposed working definition for the concept of stability, where we consider the different dimensions (5W+1H) of the taxonomy (section 2.5). Engineering stability as a software property requires addressing the following issues:

- *Stability analysis.* As stability is not an absolute property and given its different dimensions, this requires 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

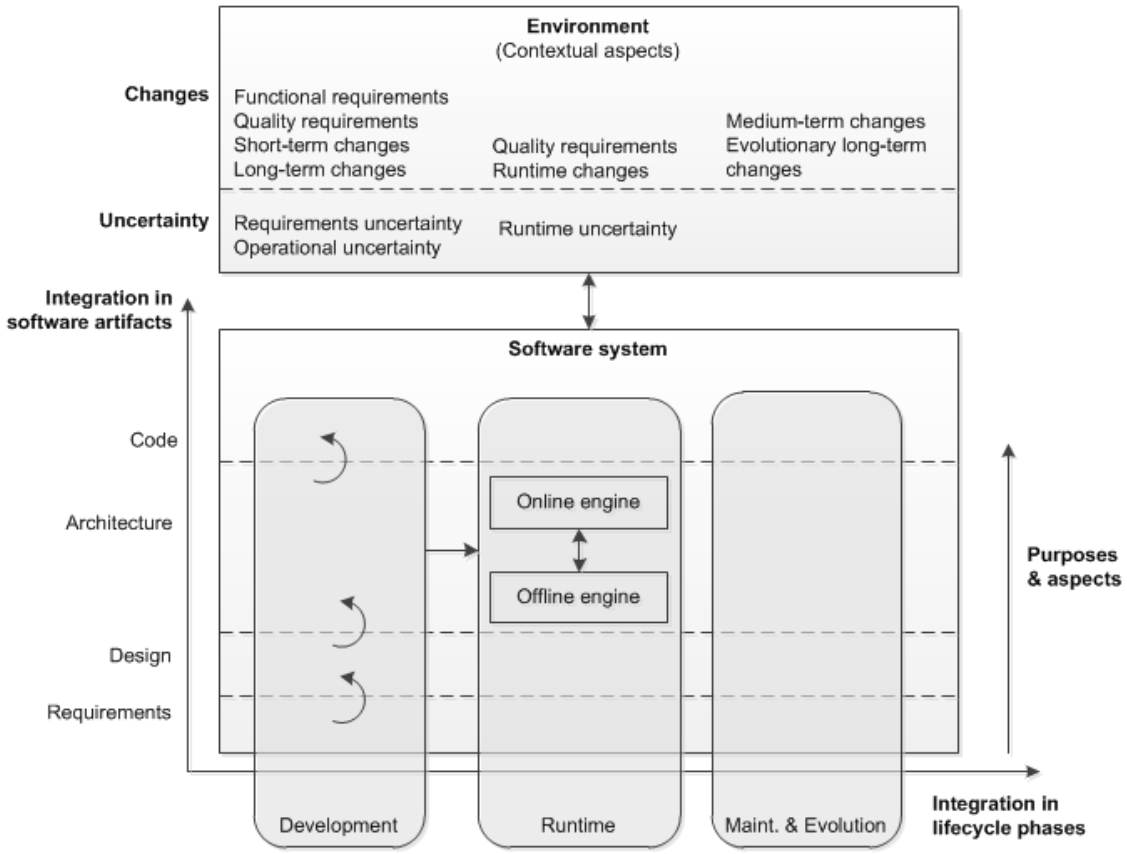

Figure 3.2: Engineering Stability as a Software Property

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 complementary. 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 in 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 the retrospective evaluation for evolution planning. Runtime evaluation approaches can vary between online and offline techniques.

- *Integration in the different software artefacts.*
  - *Design and architecture.* As architectures typically play a key role in achieving quality requirements [319] [63] [377] [378] and guiding the software evolution [17] [117], we can evidently agree that realising stability at the design and architecture levels should be based on the quality requirements subject to stability [319] [378] [379], where requirements are the key to long-term stability and sustainability [215] [380]. In other words, the outward requirements goal is concerned with what the system will accomplish for its end-users [61], which will be achieved by the architecture.
  - *Requirements engineering.* Realising stability should start in an earlier stage prior the design, i.e. in the requirements engineering phase [4], 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 [381] [19]. 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. 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 [382] [383]. Explicit relation between the requirements model and the architecture should also be present to consider the architectural stability [384] [381] [385] [39]. As runtime requirements engineering has the main role in monitoring the satisfaction of requirements during runtime [386], 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.

- *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 [387] to evaluate the risk, value perception and quantify the unpredictability.

– *Changes.* 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 [141]. 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) [141]. 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 [388]. 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.

– *Uncertainty.* Modern complex systems exhibit uncertainty from many sources, arising from the heterogeneity and dynamism of both the system itself and the operating environment [329] [22] [23]. Runtime uncertainty is associated with changes in workload [168], runtime requirements [389] [390] 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 [391] [392]. 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) [393], 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.

3.4 Requirements for Realising Stability at the Architecture Level

Engineering stability as a software property throughout the software lifecycle, along with the analysis of the current research status, have revealed new requirements for realising it. Description of these challenges is presented as follows (summarised in Figure 3.3)

[1]. We differentiate between challenges related to design-time and pre-deployment of the architecture, runtime while the architecture is under operation, and support tools.

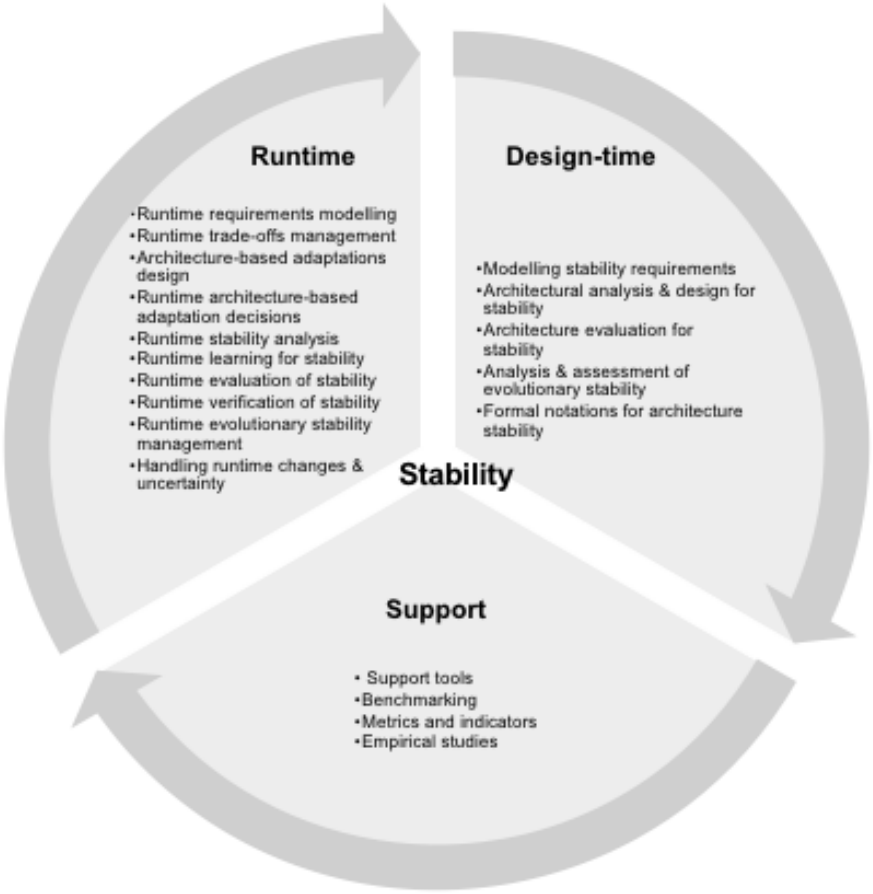

Figure 3.3: Requirements for Realising Architectural Stability

3.4.1 Design-time Requirements

- *Modelling stability requirements.* As discussed in section 3.3, requirements are the starting point for long-term stability and sustainability [215] [380]. 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 [313]. The second issue is predicting and modelling the changes in stability requirements that the systems are likely to experience during their lifetime [19], which requires a systematic way to predict the changes, quantify their likelihood [19] and their impact on the architecture. The third issue is the traceability of stability

[1]Though we discuss here requirements related to different aspects of architectural stability (e.g. behavioural and evolutionary), we address in the thesis onward the requirements related to the behavioural aspect.

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 [39]. 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 [381] [388], 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 [381] [384] [19]. 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 break-down easily when coping with increasing load demands [381] [19]. 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 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.

- *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 [394] [395]. 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 for managing, handling and rectifying uncertainty to achieve the long-term stability.

- *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 [349] and scalability [396] [397]. 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 [329] [387], 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 [387]. 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 [388]. 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 [124]. 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 [398] [155] [362]. 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 [362]. Long-term evolutionary changes, such as moving to a new paradigm or operating environment, can make use of evolutionary paths [362]. 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 [399] [400]. 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 [39], where an effective change impact analysis would assist in taking design decisions leading to stable architectures.

3.4.2 Runtime Requirements

- 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 [389] [386]. 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 [401]. 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 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 [402] [403]. 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) [25]. 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 [404] [405]. 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. [406] [407]). 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 [25], as the upper limit of the performance of the architecture is often set by stability considerations [1].

- *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 [354] [25]. 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) [351], settling time (the amount of time the controller takes to achieve the adaptation goal) [352] [353]. Meanwhile, runtime adaptations, if engineered with stability in mind, can render

benefits towards architectures intended behaviour. Runtime decisions should be seen as a continuous realisation and assessment of behavioural stability. More challenges are imposed by 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 [408], its paradigms (e.g. evolutionary computation, swarm intelligence) are promising to be applied for stability problem. Self-stabilisation of distributed systems [83] [84] [87] 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.

- *Runtime learning for stability.* The consideration of stability in runtime decisions should be complemented with online learning approaches and runtime dynamic impact analysis [409]. Such learning approaches are capable of using 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 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 a different method for the likelihood, magnitude and significance of the change [388]. At runtime, architectures can be simulated for testing the continuous fulfilment of stability requirements.

- *Runtime verification for stability.* Runtime verification is "the process of evaluating, while the system operates, whether it meets certain expected behaviour and

goals" [410] [411] [412]. Such quantitative verification is essential for self-* systems [413] [414] [415]. Given the uncertain operating environment of these systems, probabilistic model checking could be employed for continuous assurances of the intended behaviour [416].

- *Runtime evolution management for stability.* In situations where the costs and risks associated with shutting down and updating the system, runtime evolution in unavoidable [417] [24] [141]. In such case, evolutionary stability is becoming an important topic to be considered when performing runtime modifications in the architecture.

- *Managing runtime changes and uncertainty.* In considering runtime stability, changes in workload and runtime requirements should be considered [389] [168]. 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 [25]. Associated with the runtime changes is the uncertainty of these changes [335]. With the increasing complexity and heterogeneity of software systems, the uncertainty arising from the environment where the architecture is operating should also be considered [329]. 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.

3.4.3 Support-related Requirements

- *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 [343] [142] [52], based on their interlink with the stability concept. A stability benchmark can benefit from the structure of established benchmarks [142] but should additionally consider the various aspects of 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 [362] [145]. We suggest that metrics for stability would consider metrics from other inter-related attributes, such as resilience and dependability, as a base. 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 [362] [145]. Quantitative metrics would be the practical case for continuous evaluation of the architecture while the system is operating at runtime [418].

- *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 [39] [419]. 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.

3.5 Conceptual Design for Capturing Behavioural Stability

A fundamental question related to understanding the architecture's behaviour arises when engineering behavioural stability. This requires at first capturing the intended behaviour. Thus, we investigate and draw inspirations from the Control Theory discipline. The latter is one of the related disciplines that studied the notion of stability [77] [1] (refer to section 2.3) and has been widely used to incorporate self-adaptive capabilities into software systems [420].

Control Theory is mainly interested in systems behaviour and concerns itself with "means by which to alter the future behaviour of systems" [1], as such it has contributed in designing self-adaptive software systems [421]. In this context, a stable system is one that," when perturbed from an equilibrium state, will tend to return to that equilibrium state" [1]. To this end, we consider an architecture is behaviourally stable if it is able to fulfil the architecturally-significant requirements and when perturbed from its steadiness state, will tend to return to that steadiness state. Given the general definition of stability, behavioural stability for self-adaptive architectures is correlated with the quality of service subject to provision and encompasses the architecture's adaptations that tend to return the architecture to its stable state when perturbed.

Elements of Control Design. According to Control Theory principles [1], controlling (or stabilising) a system (or architecture)'s behaviour requires the following:

(i) an objective linked to the future state of the system, which is the desired behaviour (i.e. architecturally-significant requirements).

(ii) a set of possible actions for the system to be modified (i.e. adaptation actions).

(iii) means choosing the correct actions to achieve the desired behaviour (i.e. reasoning techniques).

Control Design Methodology. Putting forward the elements of control design, we set a possible design methodology for inspired by control design principles. Figure 3.4 shows the design concept. The main idea is based on a "model of the system" to compare the desired behaviour with the actual one and help in finding the optimal possible action (from the set of possible actions).

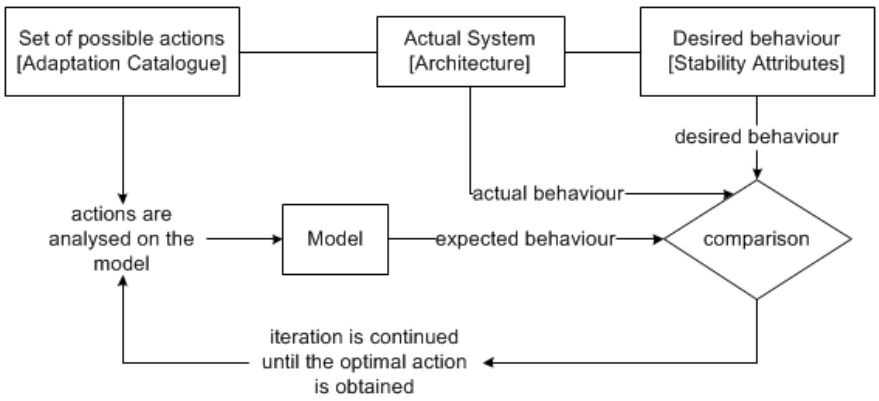

Figure 3.4: Control Design Methodology for Behavioural Stability (inspired from [1])

Given these pre-requisites and design principles, we are mainly concerned with: (i) understanding the architecture's desired behaviour (which we study in Chapter 4), (ii) modelling the behaviour (which we study in Chapter 5) (iii) designing architectures that exhibit certain behaviour (which we study in Chapter 6), and (iii) influencing or modifying the architecture to achieve the desired behaviour (studied in Chapter 7).

3.6 Summary

In this chapter, we discussed the characterisation of the stability notion based on the taxonomy dimensions. A multi-dimensional perspective for stability as a software property is discussed. The architectural level is taken forward, where related requirements are identified. Narrowing the scope to the behavioural aspect, the requirements related to this aspect are used to guide the research course of the thesis. Other requirements could serve to direct future research efforts in the community.

Further, we have sketched design principles inspired from Control Theory for capturing the architecture's intended behaviour subject to stability. Critically evaluating the work presented in this chapter, these concepts are employed to guide the research in the next chapters, where each contribution is evaluated separately, and reflective evaluation is discussed in Chapter 9.

Chapter 4

Analysing Architectural Stability

Stability leads to instability. The more stable things become and the longer things are stable, the more unstable they will be when the crisis hits.

— Hyman Minsky

4.1 Introduction

Guided by stability new perspective (discussed in Chapter 3), understanding the architecture's intended behaviour is essential for realising architectural stability. Yet the survey findings have shown inadequacy in understanding architectural behaviour and related practices. The challenge we address in this chapter is how to systematically analyse the runtime behavioural aspect of architectural stability. To address this challenge, we propose a systematic approach for analysing architectural stability, focusing on the runtime behavioural aspect. The analysis model aims to capture stability dimensions, stakeholders' concerns for stability and related attributes, in order to capture the intended behaviour subject to stability.

Contributions. The specific contributions of this chapter are as follows.

- We propose a model for analysing stability based on architectural concerns and viewpoints. Stability viewpoints frame the stakeholders' concerns for the system's behaviour along with the dimensions of stability that reflect the architecture type. Stability attributes are, then, defined to present the details of the intended behaviour needed to be kept stable.
- We describe a systematic approach for considering stability as an architectural property. The analysis approach aims at building the *stability qualitative model* that analyses and presents the intended behaviour.

- We apply the proposed approach to the self-adaptive cloud architecture case study. The analysis model has shown promising capability in exploring dimensions, concerns and attributes related to stability, and hence, drawing a comprehensive and explicit consideration of stability as an architectural property.

Organisation. This chapter is organised as follows. Section 4.2 elaborates the technical contribution of behavioural stability analysis. Section 4.3 presents our holistic approach for supporting runtime behavioural stability. In section 4.4, we apply our approach to the evaluation case study. Finally, related work is briefly discussed in section 4.5, and the chapter is concluded in section 4.6.

4.2 Stability Analysis

Stability is not an absolute property, it is treatment shall be approached using inputs from the architecture domain and intended behaviour. In particular, the architecture type (i.e. self-adaptive) and the application domain (e.g. mobile-, web-, cloud-based) have direct inputs to behavioural stability. As an example of behaviour, one architecture could be intended to keep the response time stable (as it is a crucial quality attribute for the end-users in the case of real-time systems), while energy consumption could be another critical requirement for stability. We argue that stability is a relative matter subject to the concerns of stability and the type of the architecture. Thus, stability should be considered relatively to these concerns. This calls for more expressive abstractions to represent the concerns and their related attributes subject to stability. The analysis aims to capture the relevant attributes that characterise stability concerns and stability dimensions, as well as their influence on each other's stability.

To consider the architecture type in the analysis, different components of the architecture should be considered. We view *stability dimensions* with respect to both the intended behaviour (e.g. quality of service for end-users) and the behaviour of the architecture components. The distinct dimensions allow considering both the intended behaviour and the architecture type in the analysis of behavioural stability of the architecture as a whole.

For analysing stability, we exploit one of the holistic reasoning methods for quality analysis in software architectures. In particular, we extend the "ISO/IEC/IEEE 42010 – Systems and software engineering - Architecture description" standards [422], as outlined in Figure 4.1). [1]

According to the ISO/IEC/IEEE 42010 [422], a system has one or more stakeholders, where each stakeholder has interest (i.e. concerns) for that system. Concerns are "those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders" [422]. Examples of concerns include quality of service, environmental regulations and economic concerns. We envision mapping the stability analysis to the well-known architecture related concept "architectural concerns" that refer to the requirements of different stakeholders [422].

[1] The figure uses UML notation of ISO/IEC 19501:2005, Information Technology — Open distributed processing — Unified Modelling Language (UML) Version 1.4.2.

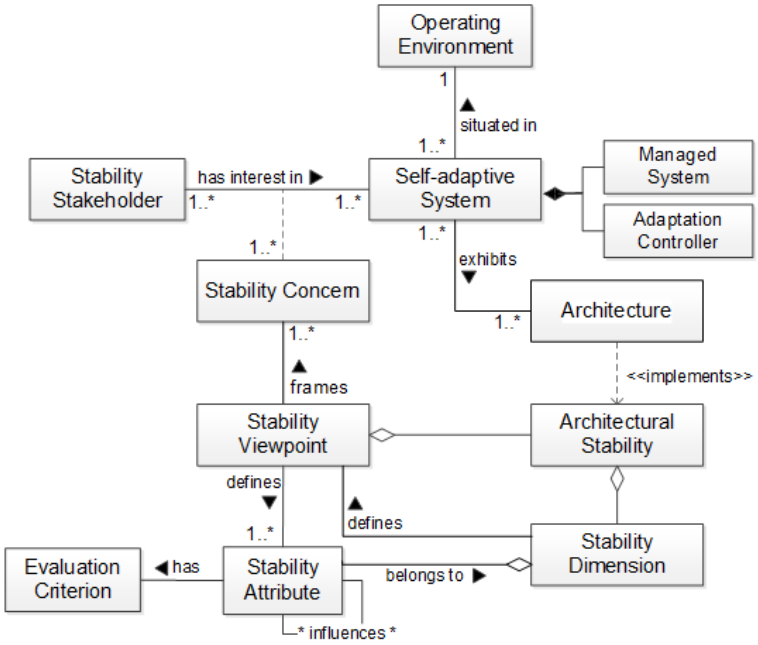

Figure 4.1: Architectural Stability Analysis Model

Considering stability, stakeholders' concerns for stabilising the architecture behaviour can be seen as architectural concerns or *stability concerns*.

Having different stakeholders, viewpoints have been introduced to support the modelling, understanding and analysis of software architectures for different stakeholders [423], delineating the architectural information that addresses stakeholders' concerns [424]. Architectural viewpoints refer to the conventions for constructing and using architectural representation addressing the requirements of different stakeholders [422] [425] [426]. Analysing stability from different perspectives can be seen as architectural viewpoints or *stability viewpoints*. We consider stability viewpoints as a model for framing stakeholders' concerns and representing architectural stability from different perspectives.

Realising runtime behavioural stability requires continuous provision of quality requirements. Following the approach of well-established architectural methods, which considers quality attributes [427] [320] [333] as the base for architectural analysis, we analyse stability in relation to the attributes that are required to be kept stable throughout the operation of the architecture, i.e. *stability attributes*. Attributes that are subject to stability are defined for different viewpoints reflecting stakeholders' concerns, including traditional quality of service attributes, which are the adaptation goals [25]). Since adaptations are motivated by the need of continued satisfaction of quality requirements, the analysis should also consider attributes of the adaptation properties [25], in order to reflect how adaptations converge towards adaptation goals.

Stability attributes are interdependent, i.e. may influence each other, either by supporting or by contradicting each other. So, architects should, for explicitly targeting stability, analyse the interdependency and correlation between different stability attributes (appearing as *influences* relation between stability attributes in Figure 4.1) and resolve related trade-offs.

While traditional architecture analysis considers dependencies and trade-offs analysis

between traditional quality attributes, such as performance and availability [428], stability analysis involves multiple viewpoints and related attributes, as well as analyses their interdependencies and trade-offs. These include not only traditional quality attributes but also adaptation properties that affect the architecture's behaviour for continuously satisfying quality requirements. Using the analysis model for identifying stability viewpoints, attributes and their dependencies explicit would help architects appreciate behavioural stability beyond traditional quality attributes.

4.3 Methodological Support for Analysing Behavioural Stability

The proposed methodology supports the initial analysis of stability as an architectural property. The outcome of this phase is the *stability qualitative model* that will contribute to the model structure for quantitatively modelling stability (presented in the next chapter). The proposed analysis methodology, illustrated in Figure 4.2, includes the following activities:

Step 1. *identify stability dimensions.* Dimensions for stability could be related to the end-users or the architecture itself. Other dimensions could be considered for the domain-specific application.

Step 2. *identify stability stakeholders.* Stability analysis entails architects to first identify the system's stakeholders that have interest in the system under consideration and hence input for stability.

Step 3. *identify stability concerns.* In this step, the stability interests and concerns of stakeholders are considered in order to build a well-balanced solution, as it is important to have a good understanding of the different concerns that the stability analysis should reflect.

Step 4. *derive stability viewpoints.* Stakeholders concerns are consolidated to derive stability viewpoints, in order to consider stability from different perspectives for building a stability solution relative to multiple concerns. The analysis also takes into consideration concerns from the components of the self-adaptive system (i.e. the managed system and the adaptation controller).

Step 5. *define stability attributes and their evaluation criteria.* Stability attributes are, then, defined for different viewpoints reflecting the stakeholders' concerns for stability. The set of stability attributes also includes attributes belonging to the adaptation properties, as one of the main stability dimensions for self-adaptive software architectures. Evaluation criteria can inform the choice of suitable metrics for assessing the fulfilment of these attributes. The choice of the metrics is highly dependent on the analysis, where the metric can be structural, behavioural, quantitative, qualitative, economic-driven in nature. Practitioners often utilise commonly used metrics. ISO standards documents [429], guidelines and quality models [53] [430],

white papers and benchmarks are among the credible sources for extracting these metrics. Systematic approaches could also be employed, such as goal-driven measurement [431] [218] and Goal Question Metric (GQM) approach [432].

Step 6. *extract interdependencies between stability attributes.* Interdependent quality attributes may influence one another. The dependencies between stability attributes are captured, in order to analyse how stabilising one attribute would affect the stability of related attributes.

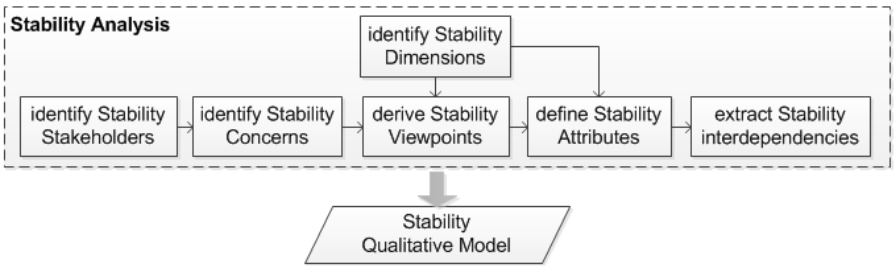

Figure 4.2: Architectural Stability Analysis Methodology

4.4 An Evaluation of Applicability

In this section, we show the applicability of the proposed approach using the self-adaptive cloud architectures case. We first describe the evaluation case study used throughout the thesis and introduce the architecture's domain (in section 4.4.1). We, then, present the step-by-step application of the approach (in section 4.4.2).

4.4.1 Architecture Domain

Cloud-based software architectures are a suitable example of dynamism, unpredictability and uncertainty [35]. The execution environment of cloud architectures is highly dynamic, due to the on-demand nature of the cloud. Cloud architectures operate under continuous changing conditions, e.g. changes in workload (number/size of requests), end-user quality requirements, unexpected circumstances of execution (peak demand) [3] [26]. The on-demand service provision in clouds imposes performance unpredictability and makes the elasticity of resources an operational requirement.

Due to the on-demand and dynamic nature of the cloud, there is an increasing demand for cloud services, where the realisation of quality requirements should be managed without human interventions. This type of architecture tends to highly leverage on adaptation (e.g. changing behaviour, reconfiguration, provisioning additional resources, redeployment) to regulate the satisfaction of end-users requirements under the changing contexts of execution [21] [26]. The self-adaptation process is meant to make the system behaviour converges towards the intended behaviour, i.e. quality requirements of the

end-users without SLA violation [26]. The purpose of adaptation is to satisfy the runtime demand of multi-tenant users, by changing configuration and choosing optimal tactics for adaptation. An unstable architecture will risk not improving or even degrading the system to unacceptable states [25]. In such case, there are more dynamics to observe, and stability is challenging with the continuous runtime adaptations in response to the perception of the execution environment and the system itself [26].

Further, the economic model of clouds (pay-as-you-go) imposes on providers economic challenges for SLA profit maximisation by reducing their operational costs [35]. Also, providers face monetary penalties in case of SLA violations affecting their profit, which push them towards stabilising the quality of service provisioned. With the rising demand of energy, increasing use of IT systems and potentially negative effects on the environment, the environmental aspect (in terms of energy consumption) has emerged as a factor affecting the software quality and sustainability [157]. While sometimes imposed by laws and regulations, decreasing energy consumption does not have only potential financial savings but also affects the ecological environment and the human welfare [157]. So, environmental requirements should be considered and traded off against business requirements and financial constraints [157].

4.4.2 Application of the Analysis Model

The outcome of the application of the analysis model is depicted in Figure 4.3. For simplicity, the information relating stability attributes with the stability concerns and their stakeholders is not included in the figure. The step-by-step application of the approach is detailed below.

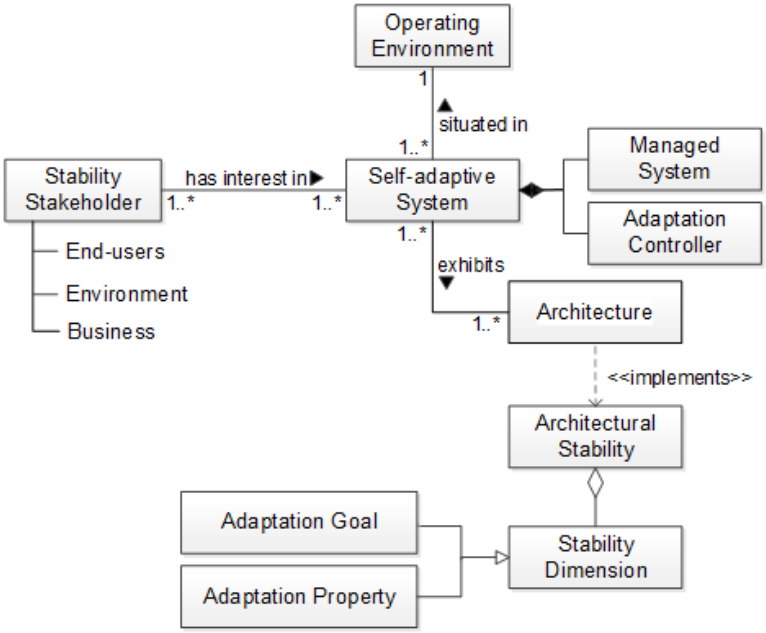

Figure 4.3: Evaluation Case: Application of the Stability Analysis Model

Step 1. *identify stability dimensions.* For the case of self-adaptive software, we identify two main stability dimensions, that are *adaptation goals* and *adaptation properties*. Both underlies the functioning of a self-adaptive system, that we intend to stabilise its architectural behaviour (i.e. the managed system and the managing system) [25] [26]. 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 [25] [26]. These two distinct dimensions allow considering both the quality requirements and the behaviour of the adaptation controller in the analysis of behavioural stability of the architecture as a whole.

Step 2. *identify stability stakeholders.* The main stakeholders that we consider are the end-users, the environment and the business.

Step 3. *identify concerns for stability.* The concerns for each stakeholder are listed as follows: (i) end-users' concern is the provision of QoS defined in their Service Level Agreements (SLAs), (ii) the environmental regulations are concerned with the energy consumption constraints, and (iii) the business is concerned with operational costs.

Step 4. *derive stability viewpoints.* Given the stability dimensions and the stakeholders' concerns, we identify the following viewpoints for stability: quality of service, environmental, economic and quality of adaptation viewpoints. The former three denote the adaptation goals, and the latter represents the adaptation property dimension. The quality of service viewpoint mainly covers the quality requirements of end-users. The environmental viewpoint covers aspects related to energy consumption and savings [157] [433]. The economical viewpoint is related to the business concerns about monetary operational cost.

Step 5. *define stability attributes and their evaluation criteria.* Based on the stability viewpoints, we define related attributes. Stability attributes could, then, include traditional quality requirements specified in end-users SLAs. Here, we consider performance (measured by response time in milliseconds), and throughput (measured by the number of completed requests per second). For the environmental aspect, we use the greenability attribute [157] [433] measured by energy consumption in kWh. For the economic constraints, we define the operational cost by the cost of computational resources (CPUs, memory, storage and bandwidth). Regarding the adaptation properties, we consider the settling time - that is the time required by the adaptation system to achieve the adaptation goal [25]. In order to capture the negative impact of adaptation on the system's behaviour, we consider the overhead of adaptation, measured by the frequency of adaptation cycles to achieve the adaptation goals [22] [25]. The analysis results of this step are illustrated in Figure 4.4.

Step 6. *extract interdependencies between stability attributes.* Dependencies between stability attributes are defined based on the architect's domain experience, as depicted in Figure 4.5. For example, performance and greenability could contradictorily affect each other, i.e. stabilising performance shall demand more computational resources

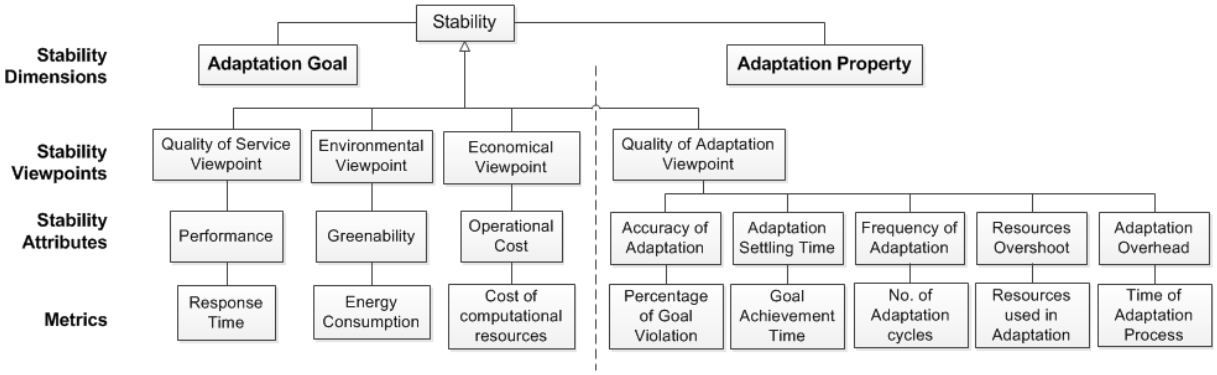

Figure 4.4: Evaluation Case: Stability Analysis Results

that consume more power and eventually have a negative effect on stabilising greenability. Meanwhile, greenability and operational cost could support each other, i.e. decreasing the usage of computational resources for saving power consumption would, in turn, decrease the operational costs.

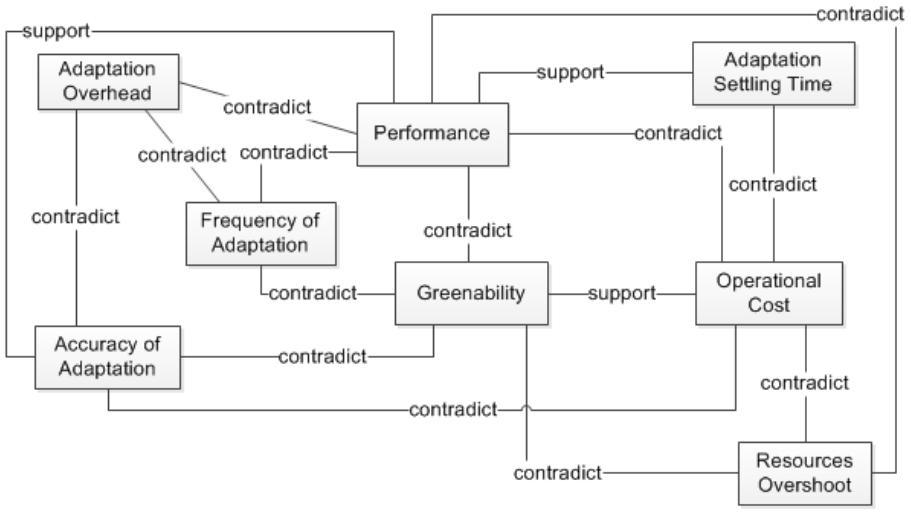

Figure 4.5: Evaluation Case: Stability Attributes Dependencies

4.4.3 Discussion

The proposed analysis model is generic enough to be applied to architecture-centric software systems. As an example, the components of the self-adaptive software (i.e. managed system and adaptation controller) could be replaced by domain-specific components of the architecture under evaluation. Domain-specific characteristics could further enrich the analysis as stability concerns and stability attributes, such as latency access for cloud federations [434]. As a general case, the stability analysis model could include any set of viewpoints and attributes subject to stability. Other stakeholders and stability dimensions could also be identified depending on the context, domain of the system and the

architecture type. On a wider perspective, the stability analysis approach could be applied to non-adaptive architectures for offline maintenance purposes. Architects can also employ the analysis model for making architectural decisions during design-time.

4.5 Related Work

The *architecture analysis* community has developed methods for predicting the quality provision of architecture design alternatives during design-time [369]. Examples include Scenario-based Architecture Analysis Method (SAAM) [327], Architecture Tradeoff Analysis Method (ATAM) [320] and quality impact analysis [344] which focused on traditional quality attributes, with no explicit focus on stability. Other studies focused on estimating system failures or predicting the probability that the system will perform its intended functionality aiming at reducing or eliminating failures [345] [346] [347] [348]. Architecture analysis methods cannot be used to support the runtime provision of quality requirements and their stability, given the uncertainty during operation and the automation and quantitative analysis required for runtime operation.

4.6 Summary

In this chapter, we presented a systematic approach for analysing behavioural stability. The analysis consolidates multiple stakeholders' concerns and architectural viewpoints for explicitly revealing attributes subject to stability, taking into consideration the software and architecture domain. The analysis introduced a qualitative model for representing the knowledge related to the attributes subject to stability and their dependencies. One feature of the analysis is making explicit consideration of the architecture type, domain and its environment while understanding the intended behaviour.

Chapter 5

Modelling Behavioural Stability of Architectures

Essentially, all models are wrong, but some are useful.

— George E. P. Box

5.1 Introduction

Following the control design principles (discussed in section 3.5), the desired behaviour captured by the Stability Analysis Model (the outcome of Chapter 4) should be modelled to support the control design principles, by understanding of the expected behaviour in comparison with the desired behaviour.

In this chapter, we propose a methodology for modelling architectural stability, focusing on the runtime behavioural aspect. Given the non-deterministic behaviour of the systems, modelling stability is based on probabilistic relational model for knowledge representation of stability multiple viewpoints and related attributes. The mathematical model leverages on the quantitative analysis of Bayesian networks for modelling dynamic impact and correlation assessment among stability attributes and analysing associated trade-offs. This approach can effectively conduct runtime inference to reason about stability attributes given the continuous runtime uncertain changes. Such reasoning improves the quality of adaptation for achieving the intended behaviour and supporting seamless operation. We show how the approach can be realised as an integral part of self-adaptive software systems runtime operation, as such the field of self-adaptive software can make a notable beneficiary of our contribution.

Contributions. The specific contributions of this chapter are as follows.

- We mathematically model the non-deterministic behaviour of stability attributes using probabilistic modelling. We present the interdependencies between stability attributes using probabilistic relational model. Based on that, we quantitatively

measure the strengths of dependence relations and sensitivity among stability attributes and construct the Bayesian network using observed data. With the help of Bayesian networks, we conduct runtime inference to measure the probable effect of stability attributes for reasoning about the whole architecture's behaviour under runtime uncertainty.

- We describe a systematic approach for considering stability as an architectural property. The approach complements the analysis approach (proposed in Chapter 4), and consists of two subsequent main phases: (i) stability modelling that captures the probabilistic relation between interdependent stability attributes by building the *stability quantitative model*, and (ii) runtime support which employs the model for runtime inference and reasoning about stability under runtime uncertainty.
- We apply the proposed approach to the self-adaptive cloud architecture case. We build the probabilistic model and conduct the experimental evaluation. The results show that reasoning about stability using the runtime inference has improved the adaptation decision and achieved the intended behavioural requirements with fewer violations.

Organisation. This chapter is organised as follows. Section 5.2 elaborates the technical contributions of behavioural stability modelling. Section 5.3 presents our holistic approach for modelling behavioural stability. In section 5.4, we apply our approach to the evaluation case study, followed by the experimental evaluation in section 5.5. Finally, related work is discussed in section 5.6 and the chapter is concluded in section 5.7.

5.2 Stability Modelling

Achieving runtime architectural stability for different viewpoints should involve a careful understanding of the relationship, impact, correlation and sensitivity among stability attributes, as well as handling potential conflicts between different viewpoints. Such attributes are non-deterministic given the uncertainty associated with the runtime operation. Uncertainty, affecting the architecture's operation, can be attributed to many facets, such as changes in workload, quality requirements, runtime goals, and the environment where the architecture is operating [435] [42]. Therefore, probabilistic modelling is appropriate for modelling stability given the runtime operational uncertainties, since deterministic analysis is limited when dealing with such operational uncertainties.

Modelling the correlation among stability attributes requires asserting changes to attributes' values, which could be informed by expert judgement or accompanied with careful assessment of the application domain. One potential problem, however, is that the analysis tends to be human-centred, subjective and can miss potential cases that are change-revealing, as such techniques rely on human judgement and sensitivity analysis. Furthermore, the analysis can be difficult to scale and handle in cases where more than one attribute can potentially change, or a higher number of attributes are under evaluation.

It worth noting that our method can complement existing architecture analysis and evaluation methods (e.g. Architecture Tradeoff Analysis Method (ATAM) [320] and quality impact analysis [344]) to provide automatic and probabilistic assessment, which replace and improve human assertions for attribute value and its likely influence on the trade-offs analysis and the choice of decisions. Probabilistic assessment is especially important for architectures that exhibit a high degree of uncertainty in their operation which is the case of self-adaptive systems.

Hence, we adhere to the Bayesian choice in the automated reasoning about stability during runtime for many reasons. As a consistent and complete representational tool, it is guaranteed to define a unique probability distribution over the network variables [436]. Also, the Bayesian network is a compact representation, as it allows one to specify an exponentially sized probability distribution using a polynomial number of probabilities [436]. The coherence of the Bayesian statistical inference is another important feature. By modelling the unknown parameters of the sampling distribution through a probability structure, i.e. by probabilistic uncertainty, the Bayesian approach authorises a quantitative discourse on these parameters [437]. The Bayesian approach is also known to be the only system allowing for conditioning on the observations, effectively implementing the Likelihood Principle and frequented optimality notions of Decision Theory [437].

5.2.1 Stability Probabilistic Model

Probabilistic modelling consists of two components: (i) the structure, often referred to as the qualitative model, and (ii) the parameters (i.e. conditional probabilities) referred to as the quantitative model [438]. For the former, we use Probabilistic Relational Models that are able to harness the expressive power of architecture analysis. For the latter, Bayesian networks feature the ability to quantitatively perform dynamic impact analysis and correlation assessment among stability attributes under runtime uncertainty [428]. Generally, Bayesian networks have proven to be ideally suited knowledge representations for reasoning and decision making under uncertainty [438], i.e. reasoning over probabilistic causal models under uncertainty [439]. Bayesian networks have been widely used for the modelling and analysis of uncertain phenomena which are known to be causally connected [440]. With the capability of representing probabilistic behaviours in a compact and intuitive way [345], Bayesian networks are applicable for domain areas with inherent uncertainty [438], which is applicable to the case of architecture's behaviour at runtime.

Stability attributes as the "knowledge" to be presented by Probabilistic Relational Models, as these attributes tend to vary during runtime. Probabilistic modelling, empowered by the quantitative analysis of Bayesian networks, aims to model the wide variations of probable values linked to stability attributes that we are interested in, as well as understand their likely ramifications on other attributes and their trade-offs under runtime uncertainty.

Our approach for modelling stability follows the formalism process of probabilistic relational models [438], that is suitable for representing and processing probabilistic knowledge of runtime behavioural stability. For each viewpoint, a probabilistic relational model is constructed using the stability attributes identified earlier in the analysis. The model represents the relation between the attributes of the viewpoint and interdependent at-

tributes. The approach for eliciting the model structure relies on the notion of cause-effect relations between the variables of the problem domain [438]. In practice, such relations are modelled using a graph of nodes representing the variables and links representing the cause-effect relations between the entities.

The construction of probabilistic networks usually proceeds according to an iterative procedure, where the set of nodes and the set of links are updated iteratively as the model becomes more and more refined [438]. Modelling causal dependence relations requires careful consideration, as sometimes it is not quite obvious in which direction a link should point [438]. In the case of architectural stability, we can rely on the architect's experience, subject matter experts and pre-experiments in defining the dependency relations between different stability attributes. Structure learning could also make use of data-driven approaches, where data could be acquired from pre-experiments and simulations. There exist different classes of algorithms for learning the structure of Bayesian networks, such as search-and-score and constraint-based algorithms [438]. Background knowledge of domain experts and architects can be specified in the form of constraints on the structure of the model.

Having the probabilistic relational model established, this defines the structure of the Bayesian network, where the elicitation of the quantitative information will take place. We use Bayesian networks to model the dynamic non-deterministic behaviour of stability attributes, that change with a range of values at runtime and tend to interfere with each other, collectively influencing the behaviour of the architecture.

A Bayesian network is a directed acyclic graph (DAG), where the nodes represent stochastic uncertain variables [441] [345], which are the stability attributes. The edges of the graph are the dependencies between the nodes, showing influential relations between the variables [441] [345]. The nodes' dependencies are specified qualitatively by the edges and quantitatively by the *conditional probability distributions*. The underlying joint probability is decomposed as a product of *local conditional probability distributions* (CPDs) associated with each node and its respective parents. The CPDs are represented as *node probability tables* (NPTs), which list the probability that the child node takes on each of its different values for each combination of values of related nodes.

Formally (following [442] [438]), a discrete Bayesian network $\mathcal{N} = (\mathcal{X}, \mathcal{G}, \mathcal{P})$ consists of a set of n discrete random variables (stability attributes) $\mathcal{X}$, a directed acyclic graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$, and a set of conditional probability distributions $\mathcal{P}$. Each variable $\mathsf{X}_i \in \mathcal{X}, \quad 1 \leq i \leq n$ is represented by a node v_i of $\mathcal{G}$ and has a finite set of mutually exclusive states $\mathsf{dom}(\mathsf{X}_i)$. The directed edges E of $\mathcal{G}$ specify assumptions of conditional dependencies between the nodes, where a directed edge from X_i to X_j is in E iif X_i is a parent of X_j. Each variable $\mathsf{X}_i \in \mathcal{X}$ has a conditional probability distribution $\mathsf{P}(\mathsf{X}_i|\mathsf{X}_{pa(v_i)}) \in \mathcal{P}$, that specifies the probabilistic dependence between the node v_i and its parents $pa(v_i) \in \mathsf{V}$.

Definition. A discrete Bayesian network $\mathcal{N} = (\mathcal{X}, \mathcal{G}, \mathcal{P})$ consists of

- a set of discrete random variables $\mathcal{X} = \{\mathsf{X}_1, ..., \mathsf{X}_n\}$
- a directed acyclic graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ with nodes $\mathsf{V} = \{v_1, ..., v_n\}$ representing the variables of $\mathcal{X}$ and directed edges $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$
- a set of conditional probability distributions $\mathcal{P}$ containing probability distribution $\mathsf{P}(\mathsf{X}_v|\mathsf{X}_{pa(v)})$ for each variable $\mathsf{X}_v \in \mathcal{X}$

The joint probability distribution of the Bayesian network $\mathcal{N}$ is obtained by the multiplicative factorisation of the joint probability distributions $\mathcal{P}$ over the set of variables $\mathcal{X}$ as represented by the chain rule of Bayesian networks:

$$\mathsf{P}(\mathcal{X}) = \prod_{v \in \mathsf{V}} \mathsf{P}(\mathsf{X}_v | \mathsf{X}_{pa(v)}) \tag{5.1}$$

The Bayesian network is constructed by computing *prior* probabilities, i.e. $\mathsf{P}(\mathcal{X})$ for all $\mathsf{X} \in \mathcal{X}$, collected from empirical data in order to get initial probability values.

Capturing dependency factors between stability attributes, the constructed Bayesian network for stability provides a powerful tool for reasoning and decision support, as it can be used to reason about the effect of stabilising a specific attribute on the stability of other attributes. By that, an adaptation action achieving the stability of the whole architecture's intended behaviour could be derived for multiple stability concerns, viewpoints and attributes. Also, behavioural stability could be estimated under changing runtime workloads.

5.2.2 Stability Runtime Inference

The Bayesian network model representation of a problem domain can be used as the basis for drawing inference and performing analysis about the domain, in order to support reasoning under uncertainty. Decision options and utilities associated with these options can be incorporated explicitly into the model, where the model becomes capable of computing expected utilities of all decision options given the information known [438]. Since a Bayesian network encodes all relevant qualitative and quantitative information contained in a full probability model, it is a well-suited tool for many types of probabilistic inference.

The Bayesian model is used to support reasoning about stability under runtime uncertainty, which requires dynamically computing the probability states of stability attributes given the runtime changes. That is the task of computing the *posterior* probability distribution of some variables of interest conditioned on some other variables that have been observed [438].

A Bayesian inference approach starts with the *priori* knowledge about the model structure. This initial knowledge, represented in the form of prior probability distribution gathered during the construction of the model, is updated to obtain *posterior* probability distribution over the model. By observing which states the nodes of the Bayesian network assume, known as *events*, we obtain the evidence ε for a subset of these nodes. With the help of evidence, we can compute the *posterior* marginals given a set of evidence ε, which are $\mathsf{P}(\mathsf{X}|\varepsilon)$ for all $\mathsf{X} \in \mathcal{X}$. If the evidence set is empty $\varepsilon = \phi$, then the task is to compute all prior marginals, i.e. $\mathsf{P}(\mathsf{X})$ for all $\mathsf{X} \in \mathcal{X}$.

Exploiting the independence relations induced by the structure of $\mathcal{G}$ and the evidence, let us consider the general case of computing the posterior marginal $\mathsf{P}(\mathsf{X}_i|\varepsilon)$ of a variable X_i given evidence ε. Let $\varepsilon = \{\varepsilon_1, ..., \varepsilon_m\}$ be a non-empty set of evidence over variables $\mathcal{X}(\varepsilon)$. For a non-observed variable $\mathsf{X}_j \in \mathcal{X}$, the task to compute the posterior probability distribution $\mathsf{P}(\mathsf{X}_j|\varepsilon)$ can be done by exploiting the chain rule factorisation of the joint

probability distribution (equation 5.1):

$$\mathsf{P}(\mathsf{X}_j|\varepsilon) = \frac{\mathsf{P}(\varepsilon|\mathsf{X}_j)\mathsf{P}(\mathsf{X}_j)}{\mathsf{P}(\varepsilon)}$$

$$= \sum_{\mathsf{X}\in\mathcal{X}\setminus\{\mathsf{X}_j\}} \prod_{\mathsf{X}_i\in\mathcal{X}} \mathsf{P}(\mathsf{X}_i|\mathsf{X}_{pa(v_i)}) \prod_{\mathsf{X}\in\mathcal{X}(\varepsilon)} \varepsilon_{\mathsf{X}} \tag{5.2}$$

for each $\mathsf{X}_j \notin \mathcal{X}(\varepsilon)$, where ε_{X} is the evidence function for $\mathsf{X} \in \mathcal{X}(\varepsilon)$ and v_i is the node representing X_i. By that, we can observe the state of all stability attributes, and hence the stability state of the whole architecture's behaviour, while the architecture is operating at runtime. The runtime inference is performed based on the Pearl's Message-Passing Algorithm [443] [444] [440].

5.2.3 Complexity Analysis of the Model

Given that the Bayesian analysis should be executed at runtime, this requires considering the complexity of the stability model.

The specification of conditional probability distribution $\mathsf{P}(\mathsf{X}_v|\mathsf{X}_{pa(v)})$ can be an intensive task, as the number of parameters grows exponentially with the size of $\mathsf{dom}(\mathsf{X}_{fa(v)})$. The complexity of the network is defined in terms of the family $fa(v)$ with the largest state space size $\|\mathsf{X}_{fa(v)}\| \triangleq |\mathsf{dom}(\mathsf{X}_{fa(v)})|$, where $fa(v) = pa(v) \cup \{v\}$. As the state space of a family of variables grows exponentially with the size of that family, a technique to reduce the complexity of Bayesian networks is to reduce the size of the parent sets $pa(v)$ to a minimum. This is, in fact, the case of the stability model, where the number of variables, i.e. stability attributes of each viewpoint, is limited. In such cases, estimating parameters from data could be a useful technique to simplify the intensive task of knowledge acquisition when operating at runtime.

While the Bayesian network is placed into operation, the model stores probability distributions and calculates various marginal distributions subject of interest [445]. So, it is important to understand the storage capabilities of the network. Given that the variables are discrete and have a finite state space, to fully specify the model, we need to elicit $\mathsf{P}(\mathsf{X}_v)$ — which is the marginal probability mass function of X_v together with the conditional mass function $\mathsf{P}(\mathsf{X}_v|\mathsf{X}_{pa(v)})$ — of each of the variables conditioned on each possible configuration of values of its parents that might occur. The practical difficulty appears when the number of different configurations of parents, and hence the number of probability vectors that need to be elicited, is extremely large. In the case of a Bayesian network for a stability viewpoint, there is one variable subject of stability X_1, and its dependant variables $\{\mathsf{X}_2, \mathsf{X}_3, ..., \mathsf{X}_n\}$. If the number of possible stability values of X_1 is m_1 and for X_i is m_i, $i = 2, 3, ..., n+1$, then the number of probabilities we need to elicit $\mathsf{P}(\mathsf{X}_1)$ is $m_1 - 1$. And to elicit all the conditional tables $\mathsf{P}(\mathsf{X}_i|\mathsf{X}_{pa(v_i)})$ we need $m_i - 1$ for each possible stability value. Summing these, we have the total number of probabilities

that need to be elicited, as follows:

$$m_1 \left\{ \sum_{i=2}^{n+1} (m_i - 1 + 1) \right\} - 1 \tag{5.3}$$

which is practically feasible, due to the structure of the Bayesian network. Also, storing stability values in ranges, rather than single values, is useful for reducing the complexity of the stability model. For instance, response time is to be considered as ranges of 1-5, 5-10 ms. instead of multiple single values.

Considering the complexity of runtime inference, though probabilistic inference is an NP-hard problem in general, the complexity is polynomial in the number of variables of the network when the Bayesian network a singly connected graph [444] [438]. This is valid in the case of stability models, where we have one stability attribute directly connected to its dependent attributes for each stability viewpoint.

5.3 Methodological Support for Modelling Behavioural Stability

Our methodology for modelling architectural stability consists of two subsequent main phases: (i) stability modelling, and (ii) stability runtime support. For each step, we identify the human-based efforts required for the qualitative analysis and potential automated tools to be used in the quantitative modelling. The approach is illustrated in Figure 5.1.

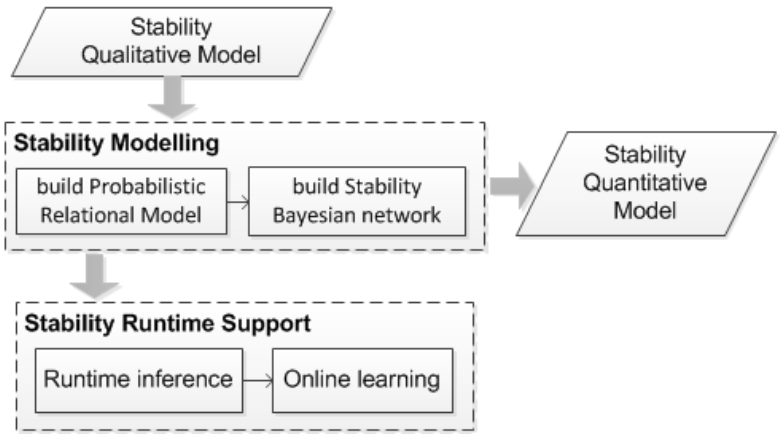

Figure 5.1: Stability Modelling Methodology

Phase 1: Stability Modelling. This phase uses the outcome of the *stability qualitative model*. In this phase, the stability model is built, and stability attributes are quantitatively assessed. This phase includes the following activities:

Step 1. *build the probabilistic relational model.* For each viewpoint, a probabilistic relational model is built, based on the attributes dependencies identified in the last step of the stability analysis. Each probabilistic relational model, representing the relations between the attributes of a viewpoint, defines the structure of the Bayesian network.

The problem of inducing the structure of the Bayesian network is NP-complete, thus, heuristic methods are considered appropriate. Building the probabilistic relational models should go through an iterative process by the architects, domain experts and subject matter experts. This could be complemented with mechanisms and tool support to facilitate the adoption of the method. Examples include tools for documenting architecture knowledge and detecting patterns of use where similar problems could exhibit similar modelling, as well as platforms for sharing experiences, guidelines and recommended practices [446].

Step 2. *build the stability Bayesian network.* The Bayesian network is built for quantitatively modelling the interdependency impacts of different stability attributes. Bayesian network specifies the strengths of interdependencies and correlations between different attributes, using probability theory and preference relations quantified by the utility associated with these attributes. This task is inducing the Bayesian network for modelling stability by fusion of observed data and domain experts' knowledge is undertaken. Building the Bayesian network could leverage on operational pre-experiments and/or simulations of the system.

The outcome of this phase is the *stability quantitative model* that will be used for reasoning about stability during runtime. The model provides a basis for what-if analysis covering probable runtime behaviour that ranges from likely to extreme scenarios.

Phase 2: Stability Support at runtime. The stability quantitative model is used during runtime for estimating probable variations in stability attributes and associated trade-offs under the dynamically changing workload. This, consequently, improves the quality of decision making under runtime uncertainty for achieving the intended behaviour of the architecture and supports seamless runtime operation of the system. This phase is the continuous runtime process of:

Step 1. *conducting the runtime inference.* During runtime, posterior probabilities are obtained using the Bayesian network that allows measuring the probable effect of stabilising different stability attributes and their impact on each other. The posterior probabilities, contributing to the adaptation decision making, help in improving the quality of adaptation and ensuring the stability of quality attributes and hence the architecture intended behaviour.

Step 2. *performing online learning.* When the system is operating, new cases arise, and it is recommended to learn from these cases, assuming that the structure of the Bayesian network will remain unchangeable [447] [438]. The conditional probabilities are dependent on the context and operation environment which change dynamically. The situation may also be that the simulation results used to extract the conditional probabilities do not reflect accurately the actual runtime workloads. This calls for online learning and updating the conditional probability distributions of the Bayesian network to reflect the real world, e.g. reasoning about quality requirements satisfaction as the system evolves dynamically [448] and learning for adaptation [449]. [1]

[1]The online learning algorithm is not discussed in this chapter, but we introduce it in our method for the purpose of completeness. An online learning algorithm is proposed in Chapter 7, and our future work will focus on integrating this algorithm with the Bayesian analysis (as discussed in section 9.3).

The runtime support for stability can be conducted online while the system is operating, by embedding the Bayesian analysis into the adaptation controller. The runtime inference and online learning can also be conducted through *symbiotic simulation* along with the adaptation controller. Symbiotic simulations shall run close to the physical system, benefiting from real-time measurements from the actual system, and provide feedback to the system [450] [451] [452]. The results of the simulation shall be used for taking adaptation decisions autonomously during runtime by the adaptation controller (managing system). While the former approach can achieve effective immediate results, it can place extra computational overhead onto the system while running. Conversely is the case of the latter approach. A balanced solution would be conducting the inference online (using a threshold prior to violation) and employing the symbiotic simulation for online learning.

5.4 An Evaluation of Applicability

In this section, we show the applicability of the proposed approach using the self-adaptive cloud architectures case described earlier in section 4.4.1.

5.4.1 Building the Stability Model

Below, we present the step-by-step application of the approach.

Step 1. *build probabilistic relational model.* Based on the interdependencies between stability attributes (from the analysis results presented in section 4.4.2), we deduce the relational model for each stability viewpoint. For instance, the probabilistic relational models related to the stability viewpoints (quality of service, environmental and quality of adaptation viewpoints) are shown in Figure 5.2, 5.3 and 5.4 respectively. The quality of service model could be read as follows: stabilising the performance would affect the stability of related attributes that are greenability, operational cost and quality of adaptation attributes. Regarding the environmental model, such representation reflects that stabilising the energy consumption would affect the stability of performance, operational cost and some quality of adaptation properties.

Step 2. *build stability Bayesian network.* To quantitatively measure the dependency factors between stability attributes and getting the prior knowledge to build the stability Bayesian network, we conducted the pre-experiments described in section 5.4.2, and the results of the stability model are presented in section 5.4.3.

5.4.2 Pre-experiments Setup

The pre-experiments were conducted using the evaluation tool, testbed configuration and architecture configurations described below.

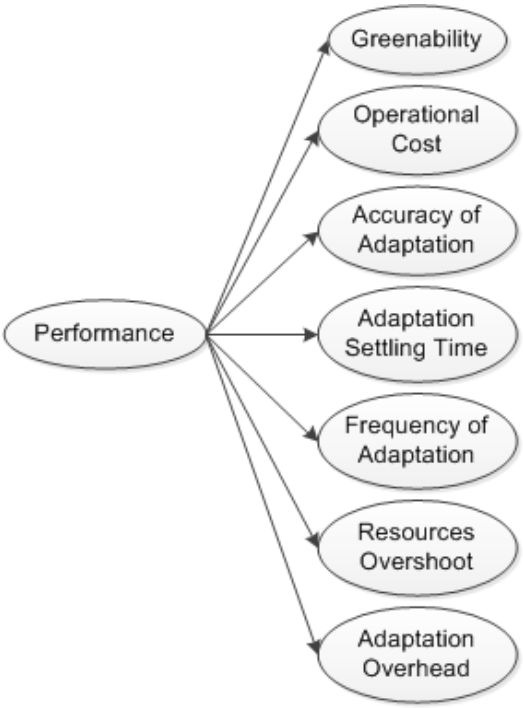

Figure 5.2: Evaluation Case: Stability Relational Model for the Quality of Service Viewpoint

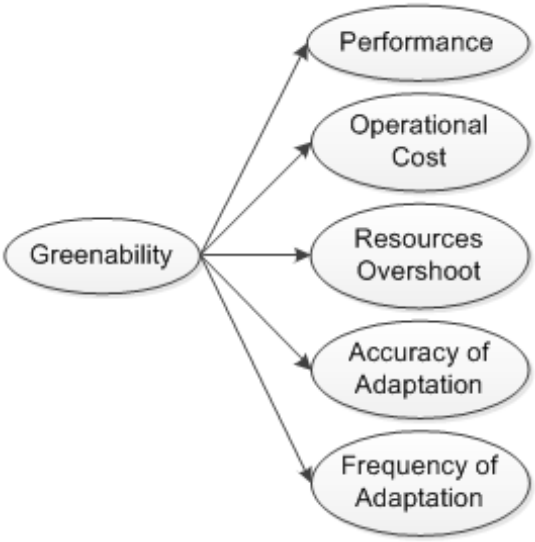

Figure 5.3: Evaluation Case: Stability Relational Model for the Environmental Viewpoint

5.4.2.1 Evaluation tool

As research in Cloud Computing is experimental in nature [36], it is almost practically not achievable to get highly scalable configurations and constant extensive workload to conduct repeatable and scalable experiments. Therefore, as most researchers do, we resort to simulation-based evaluations [36] [5], so that repeatable and scalable experimentation is manageable. However, the simulation results can be used to guide the application of

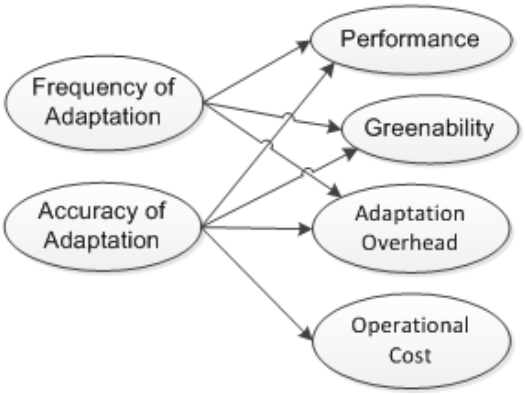

Figure 5.4: Evaluation Case: Stability Relational Model for the Quality of Adaptation Viewpoint

the selection approaches in of real-world scenarios.

To this end, we implemented the architecture of a self-adaptive cloud node, extending the widely adopted *CloudSim* simulation platform for cloud environments [5]. Our evaluation tool implements a foundational adaptation controller of the IBM architectural blueprint for autonomic computing, that is the MAPE feedback loop [453] [25], implemented as monitor, detector, adaptation engine and adaptation executor components. The simulation was built using Java JDK 1.8 and was run on a 2.9 GHz Intel Core i5 16 GB RAM computer.

5.4.2.2 Testbed configuration

The cloud architecture is configured with maximum capacity of 1000 hosts (physical machines/ server). The configuration of each server is 2 x Xeon X5675 3067 MHz, 6 cores and 256 GB RAM. The frequency of the servers' CPUs is mapped onto MIPS ratings: 3067 MIPS each core [454] and their energy consumption is calculated using power models of [454]. The maximum capacity of the architecture is 1000 hosts.

The characteristics of the virtual machines (VMs) types correspond to the latest generation of General Purpose Amazon EC2 Instances [455]. In particular, we use the m4.large (2 core vCPU 2.4 GHz, 8 GB RAM), m4.xlarge (4 core vCPU 2.4 GHz, 16 GB RAM), and m4.2xlarge (8 core vCPU 2.4 GHz, 32 GB RAM) instances. The operational cost of different VMs types is 0.1, 0.2 and 0.4 $/hour respectively. Initially, the VMs are allocated according to the resource requirements of the VM types. However, VMs utilise fewer resources according to the workload data during runtime, creating opportunities for dynamic consolidation. The testbed configuration of the experiments is shown in Table 5.1.

5.4.2.3 Architecture Configuration

The Catalogue of Architectural Tactics. We defined the catalogue of architectural tactics to fulfil the quality attributes subject to stability. Table 5.2 lists the tactics

Table 5.1: Testbed Configuration

Configuration	
Max. hosts	1000
Host type	IBM x3550 server
Host Specs	2 x Xeon X5675 3067 MHz, 6 cores, 256 GB RAM
VMs type	General Purpose Amazon EC2 Instances
VMs Specs	m4.large: 2 core CPU 8 GB RAM, m4.xlarge: 4 core CPU 16 GB RAM, m4.2xlarge: 8 core CPU 32 GB RAM

and their definitions. We have based this work on the description tactics by Bass et al. [4]. The tactics include: (i) horizontal scaling (increasing/decreasing the number of physical machines), (ii) vertical scaling (increasing/decreasing the number of virtual machines or their CPU capacities), (iii) virtual machines consolidation (running the virtual machines on less number of physical machines for energy savings), (iv) concurrency (by processing different streams of events on different threads or by creating additional threads to process different sets of activities), (v) dynamic priority scheduling (scheduling policy is implemented, where the scheduler handles requests according to a scheduling policy), and (vi) energy monitoring (providing detailed energy consumption information).

Adaptation Rules. Adaptation rules are defined as such tactics related with quality attributes. Adaptation rules are summarised in Table 5.3.

5.4.2.4 Pre-experiments Settings

We conducted the pre-experiments using the testbed configuration described above and the initial deployment of 10 running hosts. We run the pre-experiments for 300 time intervals, each interval is of 200 seconds, in order to get sufficient data for building the Bayesian network. In each time interval, we generate a random number of requests, and the length of each request randomly varies between 1,000 and 20,000 Million Instructions Per Second (MIPS).

To measure the stability ranges for different viewpoints, we configured the architecture to take adaptation actions to stabilise specific attributes within different ranges, by setting this attribute as the single adaptation goal. The adaptation controller selects an adaptation tactic from the tactics catalogue based on the adaptation rules, in order to achieve the quality requirement within the desired range. We, then, measured the impact of such stability actions on the stability of related attributes.

Table 5.2: Catalogue of Architectural Tactics

No.	Tactic	Description	Object	Limits	Variations
1	Vertical scaling	increasing the number of virtual machines (VMs) or their CPU capacities	VMs	maximum CPU capacity of hosts running in the datacenter	+1, 2, 3,... VMs or increase the CPU capacity of running VMs
2	Vertical de-scaling	decreasing the number of virtual machines (VMs) or their CPU capacities	VMs	minimum one running VM	+1, 2, 3,... VMs
3	Horizontal scaling	increasing the number of running hosts	Hosts	maximum number of hosts in the datacenter	+1, 2, 3,... hosts
4	Horizontal de-scaling	decreasing the number of running hosts	Hosts	minimum one running host	-1, 2, 3,... hosts
5	VMs consol-idation	shut down hosts running least number of VMs and migrate their VMs to other hosts	Hosts, VMs	minimum one running host and one VM	-1, 2, 3,... hosts
6	Concurrency	processing different streams of events on different threads or by creating additional threads to process different sets of activities	datacenter scheduler	maximum CPU capacity of hosts running in the datacenter	single, multiple threads
7	Dynamic scheduling	scheduling policy is implemented, where the scheduler handles requests according to a scheduling policy	datacenter scheduler	maximum number of running hosts and VMs	earliest deadline first scheduling, least slack time scheduling, single queueing, multiple queueing, multiple dynamic queueing

Table 5.3: Adaptation Rules

Tactic	Related Quality Attributes	Priority
Dynamic scheduling	response time	1
Concurrency	response time	2
Vertical scaling	response time	3
Horizontal scaling	response time	4
VMs consolidation	operational cost, energy consumption	1
Vertical de-scaling	operational cost, energy consumption	2
Horizontal de-scaling	operational cost, energy consumption	3

5.4.3 Results of the Stability Model

Samples of the Bayesian networks for different viewpoints when stabilising their attributes for one range are shown in Figure 5.5, 5.6 and 5.7. For instance, to capture stability from the quality of service viewpoint, we run the architecture with the adaptation goal of

having the performance response time stable for different ranges. Figure 5.5 shows the Bayesian network for the quality of service viewpoint when response time is stabilised for a range of 10-15 ms. The impact of such stability actions is, then, measured on the related quality attributes, i.e. energy consumption, operational cost, adaptation settling time and overhead. The attributes selected for stability, i.e. performance, is indicated by probability = 1 for the range of 10-15 ms. With respect to the environmental viewpoint, Figure 5.6 shows the probabilities of impacts of this viewpoint when the energy consumption is stabilised in the range of 50-75 kWh. The energy consumption range 50-75 kWh is indicated with probability = 1. Then, the probabilities of related attributes are calculated.

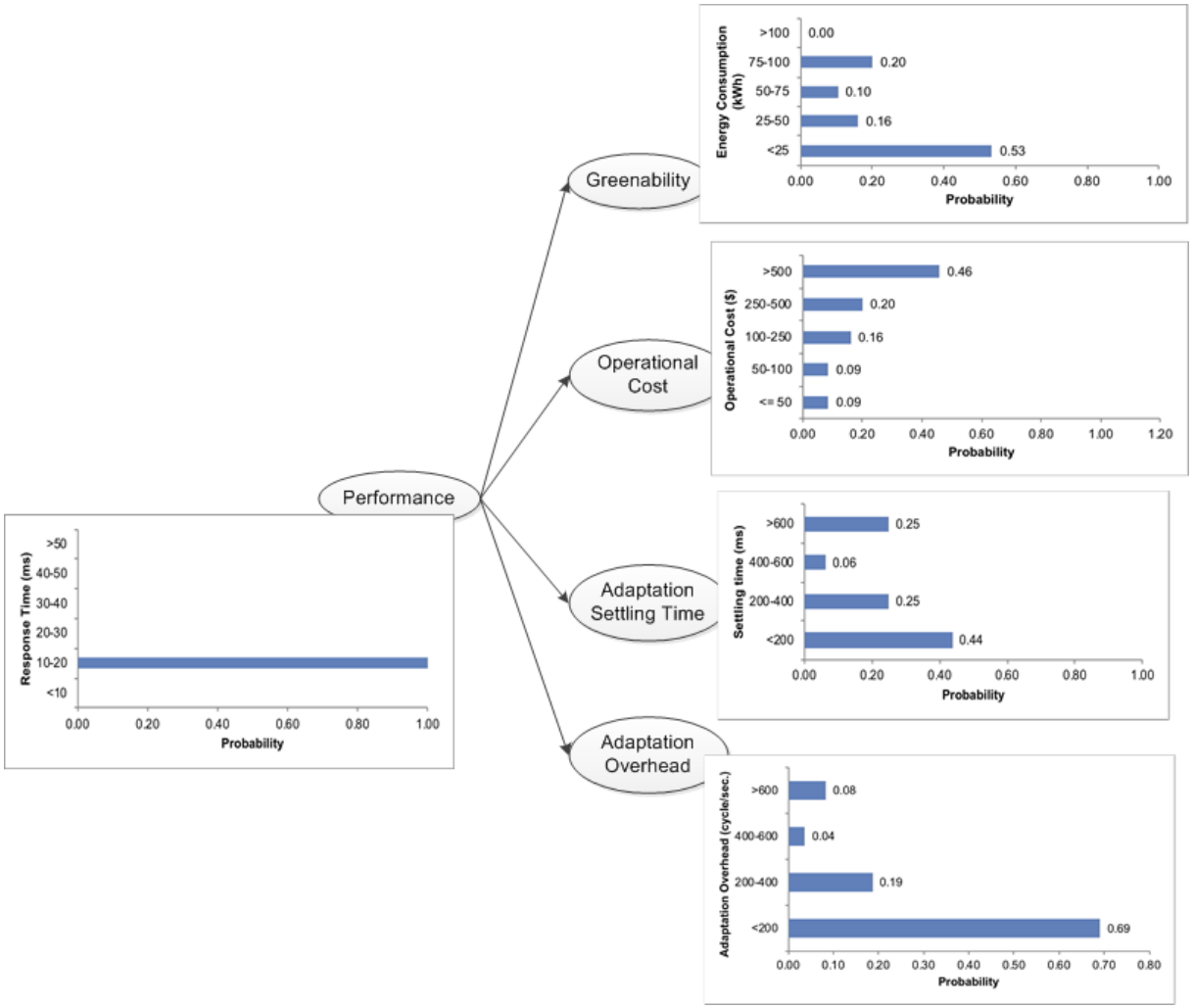

Figure 5.5: Evaluation Case: Stability Bayesian Network for the Quality of Service Viewpoint

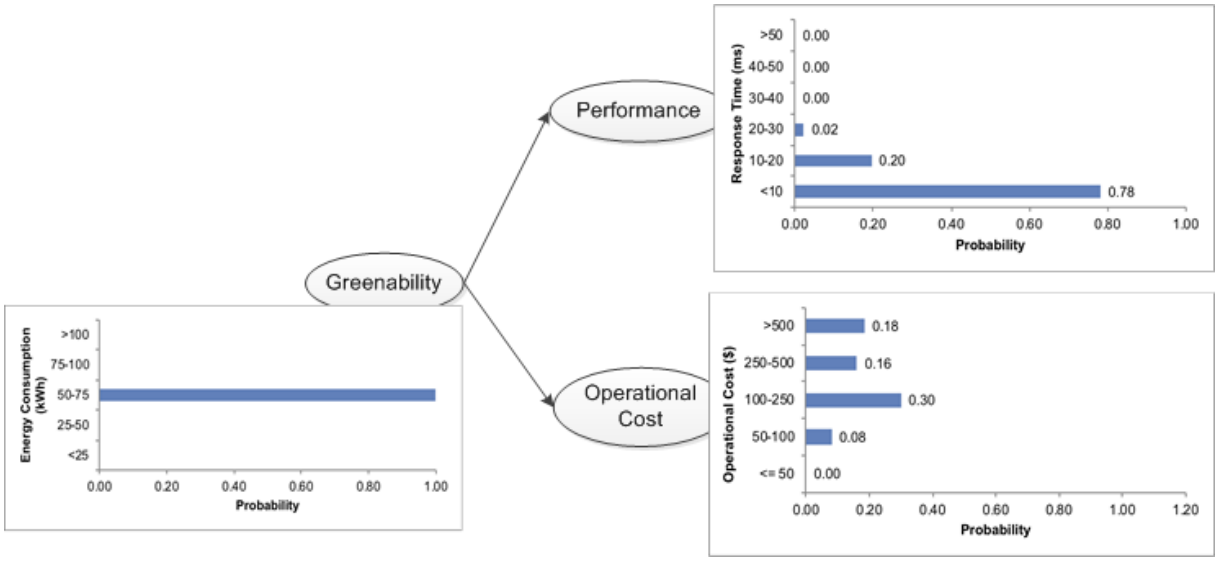

Figure 5.6: Evaluation Case: Stability Bayesian Network for the Environmental Viewpoint

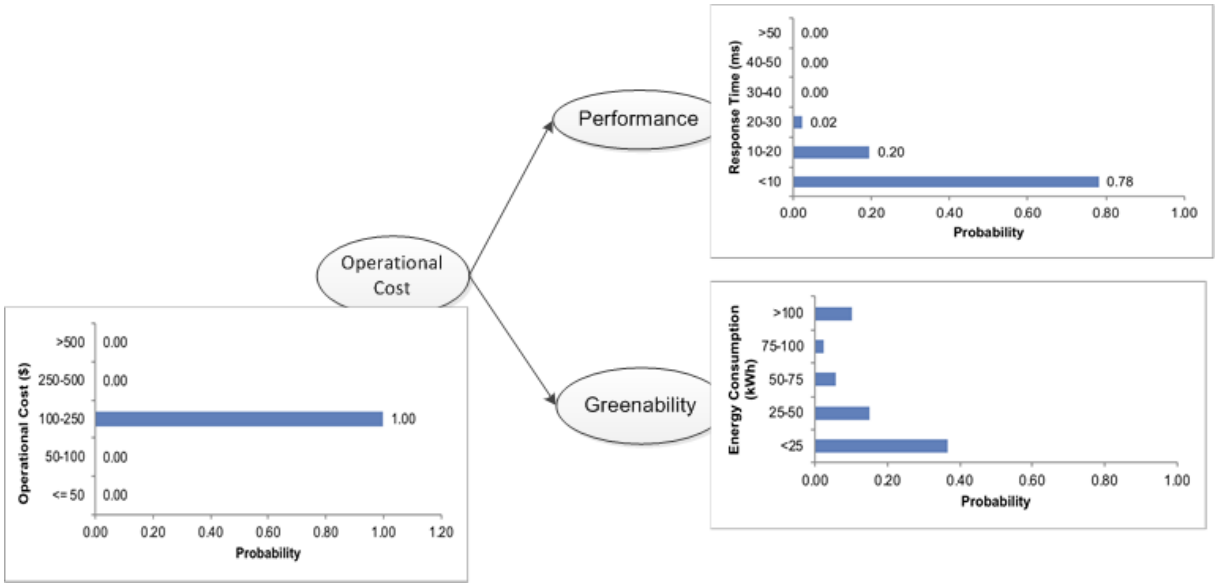

Figure 5.7: Evaluation Case: Stability Bayesian Network for the Economical Viewpoint

5.5 Experimental Evaluation

The main objectives of the experimental evaluation are to examine the quality of service delivered and the quality of adaptation when employing the stability model during runtime and to assess the associated runtime overhead.

5.5.1 Experiments Setup

The experiments were conducted using the simulation tool and testbed configuration described above in section 5.4.2.1 and 5.4.2.2 respectively. The initial deployment of the experiments is: 30 hosts running 30 VMs (10 x m4.large, 10 x m4.xlarge, 10 x m4.2xlarge). Initially, the VMs are allocated according to the resource requirements of the VM types. However, VMs utilise fewer resources according to the workload data during runtime, creating opportunities for dynamic consolidation.

5.5.1.1 Benchmarks

We used benchmarks to stress the architecture with highly frequent changing demand and observe stability goals. To simulate runtime dynamics, we used the *RUBiS* benchmark [456] and the *World Cup 1998* trend [457] in our experiments[1].

The *RUBiS* benchmark [456] is an online auction application defining different services categorised in two workload patterns: the browsing pattern (read-only services, e.g. BrowseCategories), and the bidding pattern (read and write intensive services, e.g. PutBid, RegisterItem, RegisterUser). For fitting the simulation parameters, we mapped the different services of the *RUBiS* benchmark into Million Instructions Per Second (MIPS), as listed in Table 5.4.

Table 5.4: Types of Service Requests

Service Pattern	**S#**	**Service Type**	**Required MIPS**
browsing only	1	read-only	10,000
bidding only	2	read and write	20,000
mixed with adjustable composition of the two service patterns	3	70% browsing, 30% bidding	12,000
	4	50% browsing, 50% bidding	15,000
	5	30% browsing, 70% bidding	17,000

To simulate a realistic workload, we generated the number of requests based on the *World Cup 1998* workload trend [457]. The fluctuation of the workload is depicted in Figure 5.8. To fit within the capacity of our testbed, we compressed the trend in a way that the fluctuation of one day (=86400 sec) in the trend corresponds to one time instance of 864 seconds in our experiments, and varied the number of requests proportionally. This setup can generate up to 700 parallel requests during one time instance, which is large enough to challenge stability.

[1] inspired by the earlier work of Chen et al. [3] [458] on self-adaptive and self-aware cloud architectures.

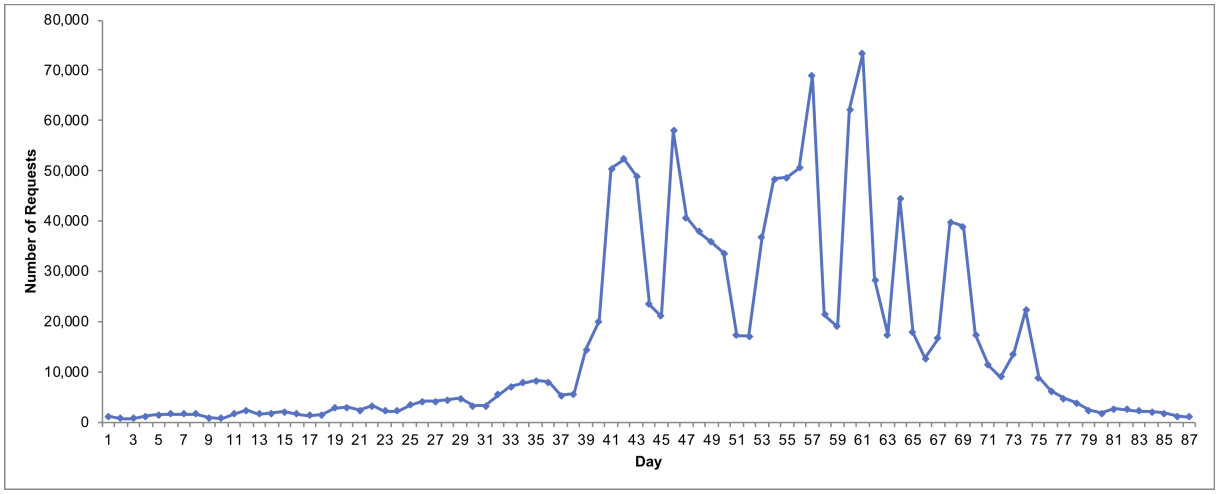

Figure 5.8: The *World Cup 1998* workload trend

5.5.1.2 Stability Goals

We performed the runtime analysis based on the following stability goals defined in Table 5.5. The objectives are defined to be challenging. The choice of the weights is hypothetical for the purpose of showing the applicability of the work. The weights reflect the importance of stability attributes, as such the end-users related attributes are given the highest weight, followed by the environment and the cost. Adaptation tactics are chosen according to the stability attributes. The runtime adaptation options and adaptation rules are described in section 5.4.2.3 and 5.4.2.3 respectively.

The architecture was configured to select adaptations (by order of preference), informed by the stability analysis in one experiment and in another one without stability analysis, which we note as stability-based adaptations and conventional adaptations respectively. We, then, examined the quality of service provisioned and closely observed the quality of adaptation in both cases.

Table 5.5: Settings of Stability Goals

Attribute	**Description**	**Weight**	**Metric**	**Objective**
Performance	Response time	0.50	ms	25
Greenability	amount of energy consumed for operating hosts	0.20	kWh	25
Operational cost	cost of computational resources (CPUs, memory, storage, bandwidth)	0.20	$	50

5.5.2 Results of Stability Goals

The average of response time, energy consumption and operational costs are depicted in Figure 5.9, 5.10 and 5.11 respectively. Generally, the adaptations informed by stability

analysis have achieved better performance in stabilising the three attributes. In more details, as shown in Figure 5.9, the response time achieved by the stability case has varied with different service types within the objective, while the average of both cases in very close.

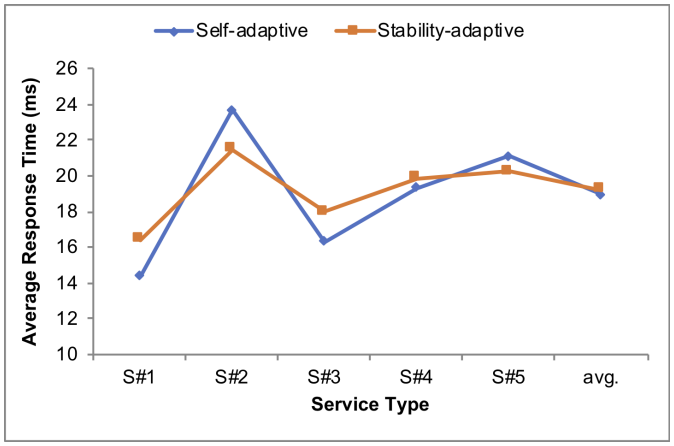

Figure 5.9: Average Results of Response Time

Regarding energy consumption, the stability-adaptive was capable of consuming less energy varying with the processing requirements of each service type. This reflects the minimal number of PMs running (resources overshoot). Similar to energy consumption and following the same trend, operational costs (reflecting the number of VMs running) was better achieved by stability-adaptive. This is due to performing adaptations that are capable of keeping stability goals for longer periods.

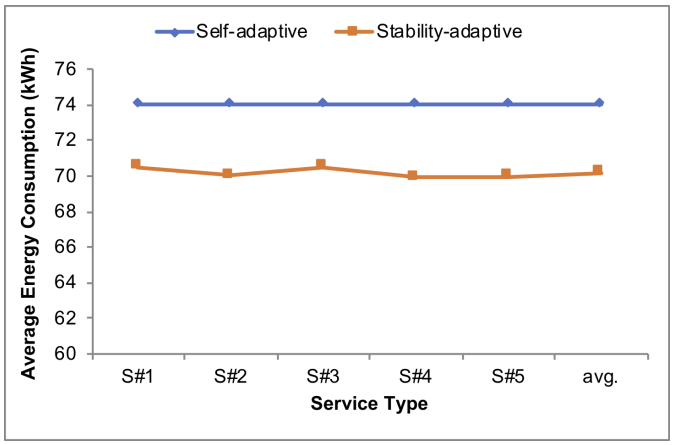

Figure 5.10: Average Results of Energy Consumption

5.5.3 Results of Adaptation Properties and Overhead

To observe the accuracy of adaptation, we closely examined the achievement of all service requests and we calculated the percentage of requests achieved without response time violations. As shown in Figure 5.12, the accuracy of adaptations is better achieved by

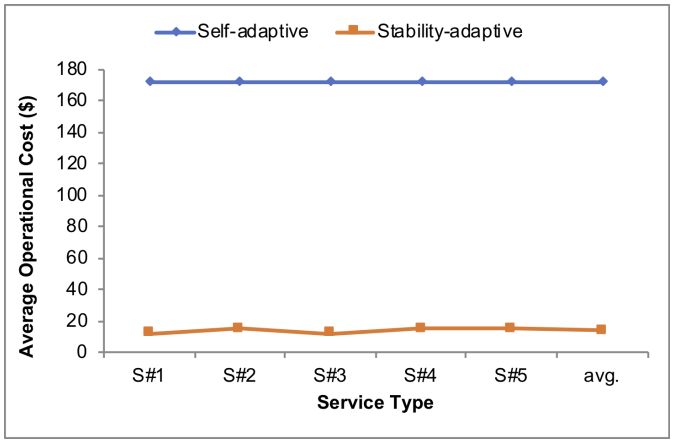

Figure 5.11: Average Results of Operational Cost

the stability-adaptive analysis. This is due to taking into consideration the quality of adaptation properties in the analysis.

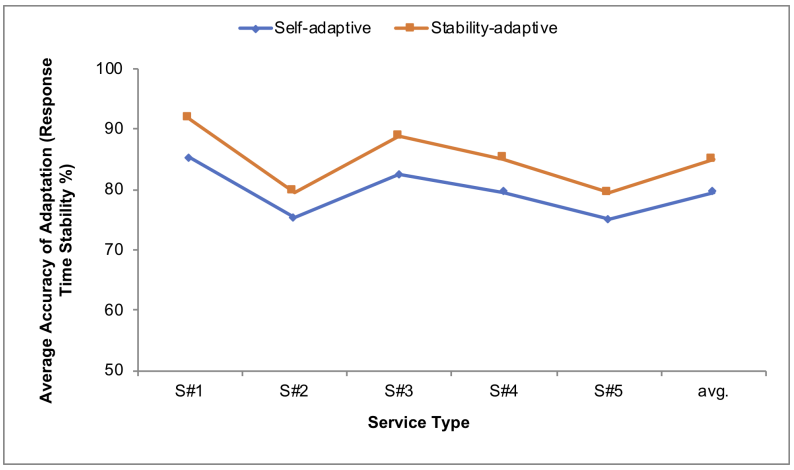

Figure 5.12: Average Results of the Accuracy of Adaptation

Given the direct impact of the frequency of adaptation on architectural stability, we evaluate the number of adaptation cycles taken by each capability. As shown in Figure 5.13, the frequency is much less in stability-adaptive case, due to making adaptation decisions that can persist longer and avoid unnecessary adaptations. Meanwhile, the self-adaptive was performing the same number of cycles for all service requests, given the same fluctuation of the workload.

We also evaluated the adaptation overhead by calculating the total time spent by the architecture in the adaptation process. Figure 5.14 shows the overhead of each service type and their average. With fewer adaptations frequency, the overhead is on average minimised by the stability-adaptations compared to the self-adaptive.

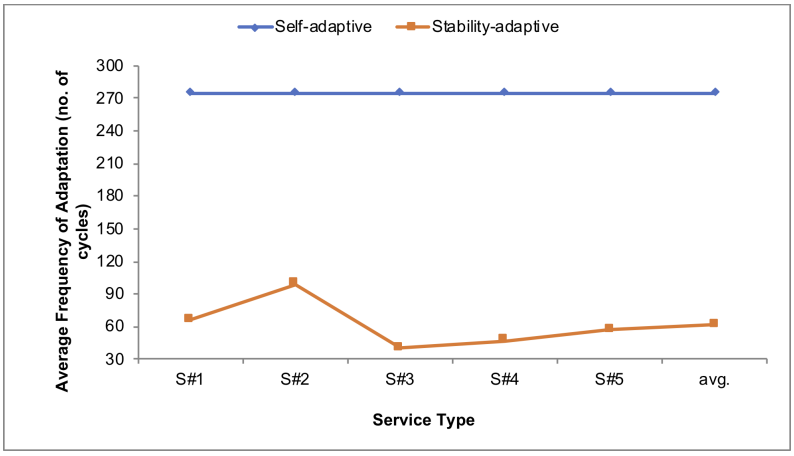

Figure 5.13: Average Results of the Frequency of Adaptation

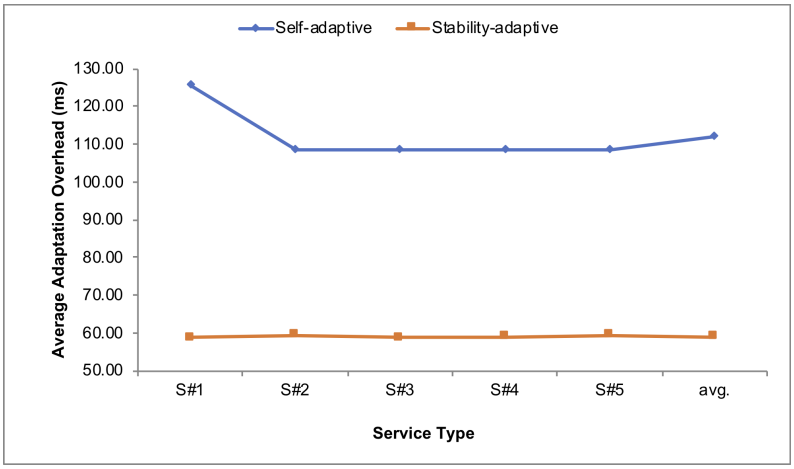

Figure 5.14: Average Results of Adaptation Overhead

5.5.4 Complexity and Runtime Overhead

Considering the complexity and overhead of Bayesian analysis, the performance of the Bayesian network is directly related to the number of nodes [445]. In the case of stability model, the number of stability attributes of each viewpoint and their parents is limited. Thus, we can claim that running the stability model during runtime is not an overhead on the system, compared with the expected benefits when achieving stability of the architecture.

With respect to the storage requirements of the Bayesian network, let us consider the case of the environmental viewpoint. There is one variable subject of stability X_1, i.e. greenability, and its dependent variables (performance, throughput, cost) $\{\mathsf{X}_2, ..., \mathsf{X}_{n+1}\}$, $n = 3$. If each attribute can take 10 possible ranges for stability, then we have 279 probabilities (from equation (5.3)), to be elicited in the stability model. For the case of the economical viewpoint, where cost is the variable subject of stability with 4 dependent variables, we have 359 probabilities to store in the stability network, which is just about practically feasible.

Meanwhile, the state space of the variables will grow exponentially, as the Bayesian network will accumulate knowledge during runtime. As such growth will affect the per-

formance of the analysis during runtime, one possible technique could be prioritising the stability viewpoints. The highly-prioritised viewpoints could be kept running online, while the less prioritised could be considered offline using symbiotic simulations and adaptation decisions will be taken forward online. Such approach will limit the overhead of running the analysis at runtime while preserving the benefits of the stability analysis.

5.5.5 Discussion

Probabilistic Relational Models for different stability viewpoints have provided a natural representation for capturing the semantics of dependencies between different attributes subject to stability. The modelling allows reasoning about a stable state for the architecture that satisfies multiple attributes essential for stability. Such consideration would prevent SLA violations, excessive runtime adaptations, and consequently architecture drifting or phasing-out. Whereas, the Bayesian networks have presented a quantification of the dependence relations strengths and the preferences for runtime decision, as well as provided quantitative evaluation for reasoning under uncertainty.

Capturing dependency factors that affect the attributes subject to stability, the Bayesian networks for different viewpoints provide a powerful decision-support tool; it can be used to measure and predict the effect of an adaptation action on stabilising a specific attribute on other interdependent attributes. The knowledge obtained from the model can also provide insights for runtime adaptations that are linked to stabilising multiple qualities. This also prevents unnecessary adaptations that could lead to instability.

Conducting stability inference for self-adaptive architectures, as part of their runtime operation, ensures more effective and efficient adaptations that contribute to the continuous fulfilment of quality requirements and eliminate SLA violations. As the objective of self-adaptivity is to seamlessly manage the runtime quality requirements and their trade-offs, the stability model allows verifying to what extent the adaptation actions are able to converge towards their goals, i.e. the quality of adaptation. Combining the adaptation properties with the adaptation goals in the process would result in a more efficient self-adaptive system. Compared with adaptations not informed by stability analysis, even with multi-objectives optimisation, stability analysis would ensure the constant provision of these requirements with fewer violations, while the former might result in frequent unnecessary adaptations leading to instability.

Integrating the stability model into the adaptation process of such dynamic architectures provides valuable support for reasoning about the adaptation decision and the operation of the architecture during runtime. The reasoning aims to satisfy not only the adaptation goals but also to ensure the constant provision in addition to the adaptation properties of the controller. The optimal of adaptations required tend to fulfil multiple stability properties and converge quickly towards adaptation goals. Such optimality of adaptation decisions can lead to the desired stable state. However, it is possible to reach and maintain stability by reaching sub-optimal stages. Henceforth, the problem would be what is the range that can keep the system stable, which could vary between minimum sub-optimal to optimal. Yet, the results are sensitive to the analysis step and the accuracy of data used to build the model.

Our method for modelling stability can make use of "sensitivity analysis" [459] [438],

in order to test the extent to which small perturbations to the inputs of the model, i.e. entries of the conditional probability distributions, can affect the stability of the whole architecture. Two types of sensitivity analysis could be performed in probabilistic models: (i) evidence sensitivity analysis, in which how the result of an evidence is sensitive to the variations in the set of evidence, and (ii) parameter sensitivity analysis, in which how the result of an evidence is sensitive to the variations in a parameter of the model. The sensitivity analysis could be easily embedded in the steps of building the stability model.

Regarding the proposed methodological support, the level of automation generally varies between steps. For instance, the qualitative analysis depends on the human capabilities (stakeholders' input and architects' decision), which is different from the automated reasoning during runtime. Though extensive efforts have been taken to ensure re-reproducibility of the method by providing systematic guidance, this would be subject to further empirical studies to determine the practicality of the method, where factors, such as availability of information, stakeholders' experience would be examined. Though, we believe that the presented case study for evaluation exemplifies the working procedure of the approach and reflect the potential usability.

5.6 Related Work

The *runtime behavioural stability of software architectures* was not explicitly tackled as an architectural property in the literature to date, to the best of our knowledge, with the exception of [340]. This study investigated the instability of service-oriented architectures, focusing on the instability of three attributes: performance, response time and communication delays. Though this work could be considered partially tackling the stability of the architecture's behaviour (performance characteristics) during runtime, the explicit focus of this work was on dependability and resilience, not explicitly considering stability as an architectural property.

Considering self-adaptive architectures, the *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) [351], settling time (the amount of time the controller takes to achieve the adaptation goal) [352] [353]. Yet, properties reflecting the quality of adaptation, i.e. how well the adaptation process converges towards the adaptation objective, are not explicitly considered [354] [25]. Meanwhile, properties reflecting the behaviour of the controller have impact on the stability of the whole architecture [25].

5.7 Summary

In this chapter, we presented a systematic approach for analysing and modelling behavioural stability. The stability analysis, based on architectural concerns and viewpoints, introduced a qualitative model for representing the knowledge related to the attributes subject to stability and their dependencies. One feature of the analysis is making ex-

plicit consideration of the architecture type, domain and its environment. For modelling stability, we employed probabilistic relational models that capture the correlations between stability attributes of different viewpoints. Bayesian networks are, then, used for quantitatively calculating probability distributions of the impact of stabilising specific attributes on interdependent attributes, as well as reasoning about stability under runtime uncertainty. The approach enables the continuous realisation of the intended behaviour; hence it could be used to take preventative actions for maintaining the required QoS and preventing violations that could lead to penalties and reputation damage.

Chapter 6

Reference Architecture and Goals Modelling for Stability

All fine architectural values are human values, else not valuable.

— Frank Lloyd Wright

6.1 Introduction

Grounded on the survey findings (Chapter 2), there is a lack of engineering practices explicitly addressing architectural stability. Meanwhile, achieving behavioural stability for long-living software calls for stability planning starting in an early development stage, i.e. in the requirements engineering and architecture design phase [4], where stability 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 [381] [19]. Hence, designing for a potentially stable architecture can be probed at design-time 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. Stability requirements should be modelled as goals at an abstract level, then technically refined to a fine-grained level that can be allocated to single components [382] [383]. Explicit relation between the requirements model and the architecture should also be present to consider the architectural stability [384] [381] [385] [39]. This will result in having the necessary runtime actions to keep the architecture stable, more effective and less cost in the long-term.

Even though architecture design has been widely investigated and derived from quality attributes [427], 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 requirements modelling and architecture design. To address this challenge, we propose a reference architecture and requirements modelling for stability based on the self-awareness computing [460]. The main purpose is to facilitate and guide the design

of stable architectures for new systems and the improvement of developed systems with architectural stability.

Contributions. The main contributions in this chapter are as follows.

- We propose a reference architecture for stability. The architecture leverages on the principles of self-awareness and self-expression —that have recently emerged in the field of software engineering as a mechanism to seamlessly improve the quality of runtime adaptations, the fulfilment of runtime requirements and the management of complex dynamic trade-offs [460]. The proposed architecture incorporates quality self-management generic components and embeds a catalogue of architecture tactics within self-awareness capabilities. Such architecture would take adaptation decisions for better tuning, responding and achieving stability goals.

- We present runtime goals modelling for stability featuring novel extensions for the Runtime Goal Models [461] —that is based on the Goal-Oriented Requirements Engineering (GORE) —in order to enable efficient use of self-awareness and self-expression in achieving stability goals. The extensions include finer-grained and dynamic knowledge representation of the runtime goals, i.e. goals attributes necessary for enabling self-awareness and measures of goals satisfaction in relation to adaptation decisions.

- We apply the reference architecture and model to the case of cloud architectures, where the continuous satisfaction and provision of quality requirements without SLA violations in the highly dynamic operating environment are challenging. Experimental results have shown that the proposed design artefacts have improved the stability in delivering the quality of service goals.

The proposed design-support artefact would assist architects and practitioners in planning for stability, as well as designing stable and long-living systems. Such design-support would increase the efficiency of the architecture runtime operation, preventing the architecture from drifting and phasing-out as a consequence of the continuous unsuccessful provision of quality requirements. As reference architectures refer to "a special type of software architecture that has become an important element to systematically reuse architectural knowledge" [462], the reference architecture makes it possible to more systematically design stable architectures.

Organisation. This chapter is organised as follows. Section 6.3 and 6.4 elaborate the technical contributions on reference architecture and goals modelling respectively. In section 6.5, we instantiate the architecture to show the applicability of our work, followed by experimental evaluation in section 6.6. We discuss related work in section 6.7 and concluded the chapter in section 6.8.

6.2 Background

In this section, we present an overview on self-awareness (section 6.2.1) and Runtime Goal Models (section 6.2.2) on which we base our work.

6.2.1 Self-Awareness and Self-Expression

As self-adaptive software systems are increasingly becoming heterogeneous with dynamic requirements and complex trade-offs [463], engineering self-awareness and self-expression is an emerging trend in the design and operation of these systems. Inspired from psychology and cognitive science, the concept of self-awareness has been re-deduced in the context of software engineering to realise autonomic behaviour for software exhibiting these characteristics, with the aim of improving the quality of adaptation and seamlessly managing these trade-offs [464] [2].

The principles of self-awareness are employed to enrich self-adaptive architectures with awareness capabilities. As the architectures of such software exhibit complex trade-offs across multiple dimensions emerging internally and externally from the uncertainty of the operation environment, a self-aware architecture is designed in a fashion where adaptation and execution strategies for these concerns are dynamically analysed and managed at runtime using knowledge from awareness.

Self-aware architecture style is defined based on a *self-aware node* unit [2] [3]. A self-aware computational node is defined as a node that "possesses information about its internal state and has sufficient knowledge of its environment to determine how it is perceived by other parts of the system" [464] [2]. A node is said to have *self-expression* capability "if it is able to assert its behaviours upon either itself or other nodes, this behaviour is based upon a nodes sense of its personality" [460]. Different levels of self-awareness, called capabilities, were identified to better assist the self-adaptive process [460] [2] [3]:

- *Stimulus-awareness*: a computing node is stimulus-aware when having knowledge of stimuli, enabling the system's ability to adapt to events. This level is a prerequisite for all other levels of self-awareness.
- *Goal-awareness*: if having knowledge of current goals, objectives, preferences and constraints, in such a way that it can reason about it.
- *Interaction-awareness*: when the node's own actions form part of interactions with other nodes and the environment.
- *Time-awareness*: when having knowledge of historical information and/or future phenomena.
- *Meta-self-awareness*: the most advanced of the self-awareness levels, which is awareness of own self-awareness capabilities.

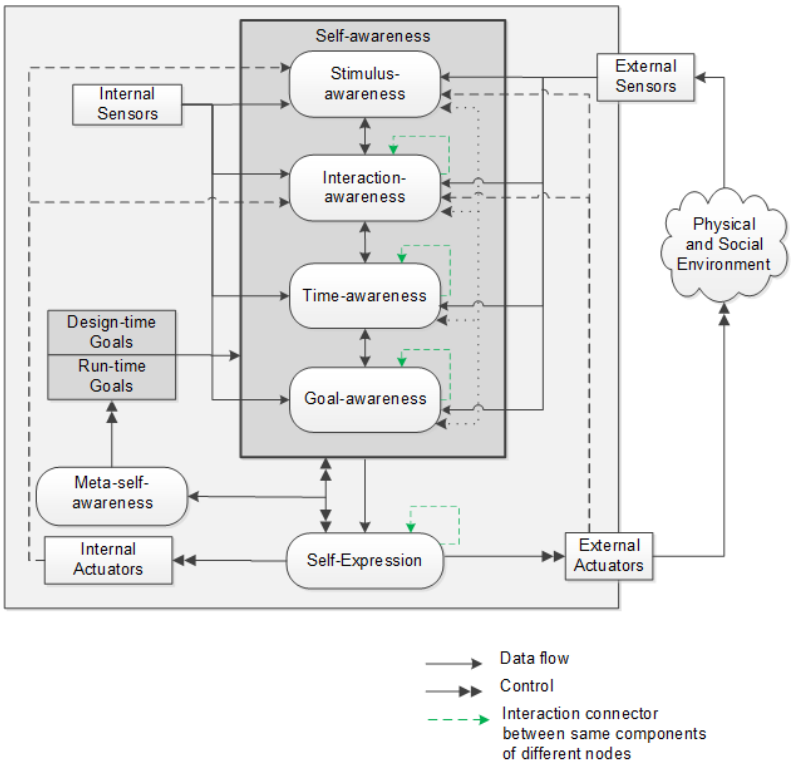

Figure 6.1: Self-Aware Computing Node (re-drawn from [2] [3])

6.2.2 Runtime Goal Models

Goal-oriented requirements engineering (GORE) has become a widely used paradigm for elicitation, modelling, analysis and reasoning of systems requirements [465] [466]. Goals are objectives to be achieved by the system under consideration [382], i.e. prescriptive statements of intent whose satisfaction requires the cooperation of different components in software and its environment [381] [383]. Goals range from high-level to fine-grained technical prescriptions that can be assigned as responsibilities to single components [383] [382]. Goals, thereby, provide a rationale for requirements and allow tracing low-level details back to high-level concerns [382].

Runtime Requirements Models —denoting requirements models that are used at runtime —have a key role to support monitoring requirements satisfaction and the consequent adaptations during runtime. Runtime Goal Models, extending design-time goals, were proposed to analyse the runtime behaviour of a system with respect to the satisfaction of requirements and consequently refine the goals specification model, its assumptions and operationalisation decisions [461] [467] [468]. Runtime goals were employed in self-adaptive software catering for uncertainty [469].

6.3 Self-aware Reference Architecture for Stability

As the architecture design plays an essential role in delivering the quality requirements [4], architectural behavioural stability is directly related to the intended behaviour of the

architecture. As an example of behaviour, one architecture could be intended to keep the response time stable (as it is a crucial quality attribute for the end-users in the case of real-time systems), while throughput could be a critical requirement attribute to be kept stable for another architecture. Having the architecture's intended behaviour stable, by assuring the delivery of some quality attributes, is highly desirable.

The reference architecture is based on an architecture pattern enriched with self-awareness and self-expression components, quality self-management components and catalogue of architectural adaptation tactics for achieving the intended behaviour. Self-awareness capabilities are employed to safeguard the stability of these attributes, where the selection of the appropriate tactic leading to stability will be performed during runtime by the awareness capabilities. Incorporating the tactics, as adaptation actions to meet the quality requirements, will improve and enrich the quality of self-expression, i.e. the adaptation actions taken during runtime [41]. Such reference architecture allows instantiation of different patterns suitable for different software domain applications interested in stability.

Achieving such stable behaviour requires adaptation actions to cope with the runtime changes. Adaptability is known to be the current routine to consider various "ilities" –subject to stability –when architecting systems [470]. Architecting for adaptability is meant to make adaptability part of the architecture design reviews, by creating a catalogue of adaptability-enhancing design tactics [470]. As such, our reference architecture is enriched with a catalogue of architectural tactics as adaptation actions designated to fulfil quality attributes subject to stability. The architectural pattern is also enriched with quality self-management capabilities, in order to achieve the desired behavioural stability [41].

We envision that self-awareness and self-expression are the most convenient capabilities for realising behavioural stability. The self-awareness capabilities, embedded in the architecture pattern, own the necessary knowledge for achieving stability and keeping the stable state. For instance, the stimulus- and goal-awareness could provide knowledge about stability goals relevant to the system. The time-awareness could help with the historical information and/or future phenomena about achieving stability.

To design the reference pattern for achieving stability attributes, we follow the general quality scenario presented at [4] to formally capture stability requirements. The general scenario, illustrated in Figure 6.2, is described as follows:

1. The Stability Monitor (*source of stimulus*) monitors changes in stability attributes (*stimulus*) during runtime and collect relevant data.

2. The architecture pattern (*artefact*) is responsible for realising stability. Stimulus-awareness is responsible for detecting violations (or possible violations as per threshold) in stability attributes and notifying the self-awareness component to consider adaptation action. The self-awareness responds by selecting an architectural tactic from the Tactics Catalogue, embedded in the pattern, to meet stability requirements and accordingly perform the adaptation actions.

3. The Self-expression component is, by its turn, responsible for composing the tactic (*response*) and instantiating it as an adaptation action.

4. The *response*, after the execution of the tactic, is measured by the Architecture Evaluator which in turn feeds the different levels of awareness to take further actions if needed and keep history.

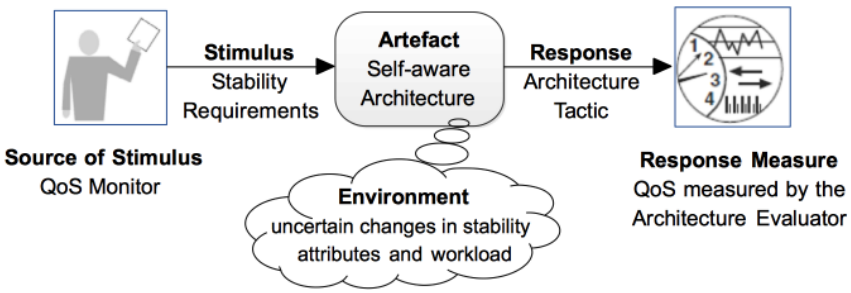

Figure 6.2: General Scenario for Designing Stability-driven Pattern (adopted from [4])

6.3.1 Quality/Tactics Self-management Generic Components

The reference architecture aims at supporting the process of architecture design for stability. Figure 6.3 illustrates the architecture pattern with self-awareness capabilities and tactics generic components. To achieve the envisioned quality self-management capability, the generic components added within self-awareness capabilities are:

- *Stability Monitor component*: responsible for monitoring changes in workload and stability attributes during runtime.
- *Tactics Catalogue*: a catalogue of runtime tactics designated to achieve different quality attributes subject to stability. As stimulus-awareness is the base of all self-awareness capabilities, the catalogue of architectural tactics is embedded at the stimulus-awareness component.
- *Tactics Rule Manager*: embedded at the stimulus-awareness level, it defines *if-condition-then-action* adaptation rules, where the conditions are stability requirements and the actions are response tactics. Rules include priorities for tactics to reflect the order of executing tactics (e.g. vertical scaling is used first before horizontal scaling for faster response and less cost).
- *Adaptation Engine*: could be seen as a more complex version of the Tactic Rule Manager present in different levels of awareness. A goal-oriented adaptation engine uses knowledge about design-time and runtime goals available at the goal-awareness component to make decisions about tactic selection in line with the system's current goals. Interaction-oriented adaptation engine contributes to the selection of the adaptation decision according to runtime conditions of the other nodes in the interacting environment where the node is collaborating.

- *Adaptation Trainer*: helps in improving the selection of the adaptation decision using historical information. Historical data, received from the Stability Monitor and the Architecture Evaluator, include tactics responses under different runtime conditions to improve the quality and accuracy of adaptation in the future.

- *Adaptation Manager*: in the meta-self-awareness level, is responsible for managing trade-offs between stability attributes during runtime and switching between different behavioural strategies in the interaction-, time- and goal-awareness capabilities. The dynamic selection of the appropriate tactic at runtime is performed based on the reasoning about the benefits and costs of selecting a tactic based on a certain level of awareness in order to meet stability attributes while managing trade-offs between them.

- *Tactic Executor*: responsible for managing the process of tactic composition and execution during runtime at the self-expression level. In more details, it makes instructions about the composition and instantiation of the components required to execute the tactic, and the actual execution of the tactic components and connectors during runtime.

- *Architecture Evaluator*: evaluates the response after executing of the tactic, and feeds the different levels of awareness to take further actions if needed and accumulate historical information.

6.3.2 Designing Stability-driven Architecture Patterns

We discuss how the reference architecture could be instantiated. A variety of patterns could be designed using different combinations of self-awareness capabilities, so that the pattern used when designing the software would include capabilities relevant to the software requirements [41]. This could follow the methodology for designing self-aware and self-expressive systems proposed in [3]. We use the set of self-aware and self-expressive patterns of [3] and [41] that are Basic Pattern (P_1), Basic Information Sharing Pattern (P_2), Coordinated Decision-making Pattern (P_3), Temporal Knowledge Aware Pattern (P_4), Temporal Knowledge Sharing Pattern (P_5), Goal Sharing Pattern (P_6), Temporal Goal Aware Pattern (P_7), Temporal Goal Sharing Pattern (P_8), Meta-self-aware Pattern (P_9), as examples of different possible combinations. The generic components added in different self-aware patterns are summarised in Table 6.1.

6.4 Runtime Goals Modelling for Stability

In this section, we present the finer-grained knowledge representation of our proposed *SAwGoals@run.time* for modelling runtime stability goals.

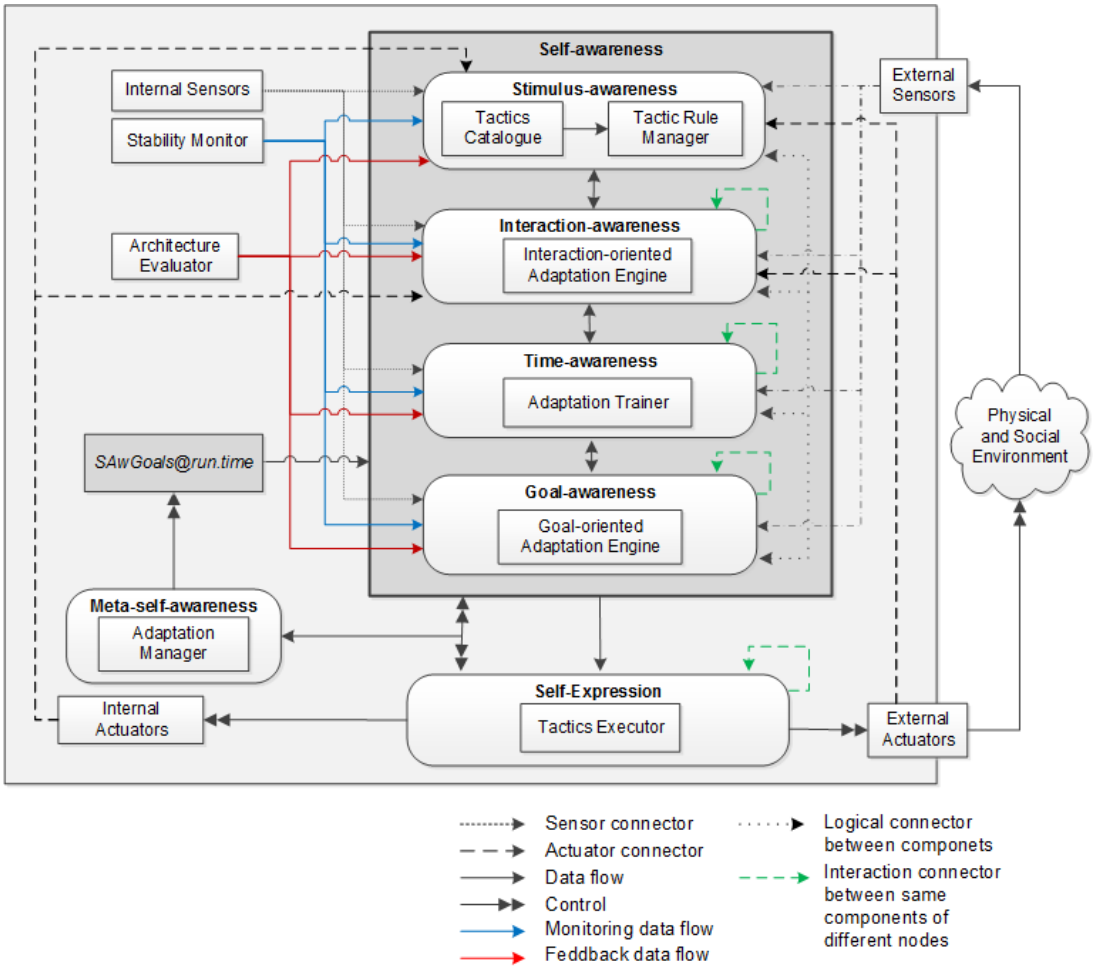

Figure 6.3: Reference Architecture Pattern with Tactics Generic Components

Table 6.1: Variations of Stability-driven Architecture Patterns

Component	Patterns								
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
Stability Monitor	√	√	√	√	√	√	√	√	√
Tactics Catalogue	√	√	√	√	√	√	√	√	√
Tactics Rule Manager	√	√	√	√	√	√	√	√	√
Goal Adaptation Engine	-	-	-	-	-	√	√	√	√
Interaction Adaptation Engine	-	√	√	-	√	√	-	√	√
Adaptation Trainer	-	-	-	√	√	-	√	√	√
Adaptation Manager	-	-	-	-	-	-	-	-	√
Tactic Executor	√	√	√	√	√	√	√	√	√
Architecture Evaluator	√	√	√	√	√	√	√	√	√

6.4.1 Runtime Goals and Self-Awareness

We propose enriching the architecture pattern with *SAwGoals@run.time* component (as illustrated in Figure 6.3). As runtime goals drive the architecture in reasoning about adaptation during runtime [471], *SAwGoals@run.time* extends the GORE model to suit

the needs of self-awareness capabilities and stability requirements. The objectives of the proposed modelling are: (i) fine-grained dynamic knowledge representation of stability goals to enable efficient use of the different levels of self-awareness, (ii) monitoring the satisfaction of stability goals and the performance of tactics, (iii) better informed decision of the optimal tactic for realising architectural stability, and (iv) continuous accumulation of historical information to update the knowledge for future learning using time-awareness.

We refine the Runtime Goal Models with fine-grained dynamic knowledge representation that reflects self-awareness needs for new attributes of the goals, operationalisation, tracing down to architecture and runtime satisfaction measures. Specifically, additional runtime behavioural details relevant to different levels of self-awareness are integrated, such as node information for interaction-awareness, and trace history for time-awareness, as well as information about the execution environment in different time instances. Operationalisation of stability attributes is realised by self-expression, through runtime tactics which are defined within the proposed model. The model would better operate in the presence of historical information about the ability of operationalisation decisions. In the case of instantiation, it is imperative that the designer considers what-if analysis, simulation or scenarios to test the suitability of the choice. Models which rely on decision-making under uncertainty can also be sensible to employ. Given relevant information about goals and the operating environment, conflict management between goals during runtime is handled by meta-self-awareness capabilities.

The proposed *SAwGoals@run.time* overcomes the limitations of GORE with respect to self-awareness and self-expression as follows:

- *Goal Attributes.* Operating different levels of self-awareness requires detailed information about the goals during runtime. Such information should include attributes about the interacting node, time instance, the execution traces, the adaptations and their performance to satisfy the goal, as well as the operating environment. For instance, information about goals from other nodes and adaptations taking place in the operating environment is required for the interaction-awareness level. Having this information for different time instances would form historical information useful for the time-awareness level to improve the accuracy of adaptation.

- *Goal Operationalisation.* Operationalisation is performed at the self-expression level using Runtime Goal Model operationalisation, as follows. For operationalising stability attributes, we extend the Runtime Goal Model to introduce alternative of runtime tactics, designed to stabilise and operationalise changes in stability goals at runtime. QoS provision under runtime uncertainty could be handled using alternative operationalisation strategies/ tactics designated for various quality attributes [56] [67]. For instance, self-aware systems encounter during runtime uncertain changes in stability goals due to the changing workloads and size of jobs from users with different SLAs. Runtime tactics designed for performance, like vertical and horizontal scaling, are candidate artefacts for handling stability goals, from which self-awareness can select the optimal handling tactic. The extent to which goals are satisfied is subject to the choice of the tactic.

- *Conflict Management.* As the system encounter operationalisation decisions during runtime for multiple goals, conflicts are likely to exist. Conflict management

in dynamic environments exhibits numerous uncertainties and trade-offs requiring intelligent strategies for negotiating conflicts, prioritising and reconciling decisions. Conflict management, through active negotiation, can rely on information related to the historical performance of the tactics in meeting the goals. Negotiation is continuously live in the self-aware system, as such: once reconciliation is reached and a decision is taken, a *trace* of the decision is monitored for its ability to satisfy the goal and possible dependencies. This information can feed into subsequent cycles of negotiation, with the objective of better resolving conflicts the system learns through self-awareness.

6.4.2 Runtime Goals Knowledge Representation

Runtime goals in *SAwGoals@run.time* are defined along with an execution trace and traced to runtime tactics for operationalisation. A **Runtime Goal** (e.g. performance) $G \in \mathcal{G}$, where $\mathcal{G}$ is the set of goals in a self-aware and self-expressive node. A goal is defined by the following attributes:

- *Unique identifier id* of the goal G.
- *Definition.* formally and informally defining the goal and its satisfaction in an absolute sense.
- *Node identifier* N, the unique identifier of the self-aware node responsible for realising the goal.
- *Weight* w to consider the priority of the goal.
- *Metric* M a measurable unit (e.g. response time measured in milliseconds) that can be used to measure the satisfaction of the goal while the system is running.
- *Objective* $f(G)$ defines the measures for assessing levels of the goal satisfaction with respect to values defined in SLAs of different end-users (e.g. objective for performance are response time 15 ms and 25 ms for dedicated and shared clients).
- *Set of tactics* $T(G) \in \mathcal{T}$ to be used in case of violation of the goal. The goal semantic is the set of system behaviours, i.e. runtime tactics, that satisfy the goal's formal definition.

A **Runtime Tactic** $T \in \mathcal{T}$ (e.g. vertical scaling) is defined as follows:

- *Unique identifier id* of the tactic T.
- *Definition* includes the description and informal definition for when to apply the tactic and how to execute it.
- *Object* in the architecture in which the tactic is executed (e.g. VMs).
- *Pre-condition* defines the current condition of the operating environment in which the tactic could be applied.

- *Limits* defines the minimum and maximum limits of the architecture for executing the tactic (e.g. the maximum number of servers).
- *Functionality* defines how the tactic should be executed.
- *Post-condition.* This characterises the state of the operating environment after applying the tactic.
- *Variantions of the tactics* includes different forms or possible configurations for applying the tactic (e.g. earliest deadline first scheduling, least slack time scheduling).

A **Runtime Goal Instance** $G(n, t_i)$ is an instance of the runtime goal G in the self-aware node n at a certain time instance t_i, and is defined as follows:

- *Client* c issuing the service request r.
- *Objective* denotes the quality value defined in the SLA of the client c.
- *Tactic* T and its configuration executed as an adaptation action to satisfy the goal.
- *Actual value* v denotes the degree of satisfaction achieved after the execution of the tactic T that is measured by the Architecture Evaluator.
- *Set of environment runtime goals* G_e, that are the goals from other self-aware nodes n_x running at the same time instance t_i with which the node n is interacting, where $G_e = \{G_1(n_1, t_i), G_2(n_2, t_i), ..., G_x(n_x, t_i)\}$.
- *Set of environment runtime tactics* T_e, that are the tactics taking place at the same time instance t_i in the environment, where $T_e = \{T_1, T_2, ..., T_x\}$ for $\forall\ G \in G_e$.

For each goal G, *change tuples* are created at different time instances t_i to form the history of this goal $\mathsf{H}(G)$ for keeping record of the goal satisfaction and related tactics performance over time. This history shall be used by time-awareness to reason about adaptation actions in the future.

6.5 An Evaluation of Applicability

In this section, we show the applicability of the proposed work through the case of cloud architectures described in section 4.4.1.

6.5.1 Application of the Reference Architecture

We created the architecture of a cloud node using the reference architecture to perform stability-driven adaptations, as illustrated in Figure 6.4. To this end, this architecture should dynamically perform architecture-based adaptation, which would use the knowledge available at different levels of awareness in choosing optimal tactics to meet stability requirements during runtime. The instantiated architecture pattern embeds different

awareness components, with exception of the interaction-awareness, to focus of a single cloud node with no interaction with other nodes. To focus on the evaluation of the proposed architecture, only the goal-awareness components is enabled in these experiments. Other awareness components are used in the next chapter for reasoning about stability.

The architecture embeds the catalogue of architectural tactics (defined in section 5.4.2.3) to fulfil the stability goals. Architectural tactics are defined in the Tactics Catalogue component. Adaptation rules (listed in Table 5.3) are embedded in the stimulus-awareness component. Monitors for stability attributes are implemented in the Stability Monitor component. Components necessary for checking possible violation of stability attributes are implemented in the stimulus-awareness component, e.g. SLA Violation Checker and Green Performance Indicator. The scheduler component of the scheduling tactic was embedded into the stimulus-aware. Management components of tactics were configured into the Tactic Executor for running the tactics, e.g. auto-scaler.

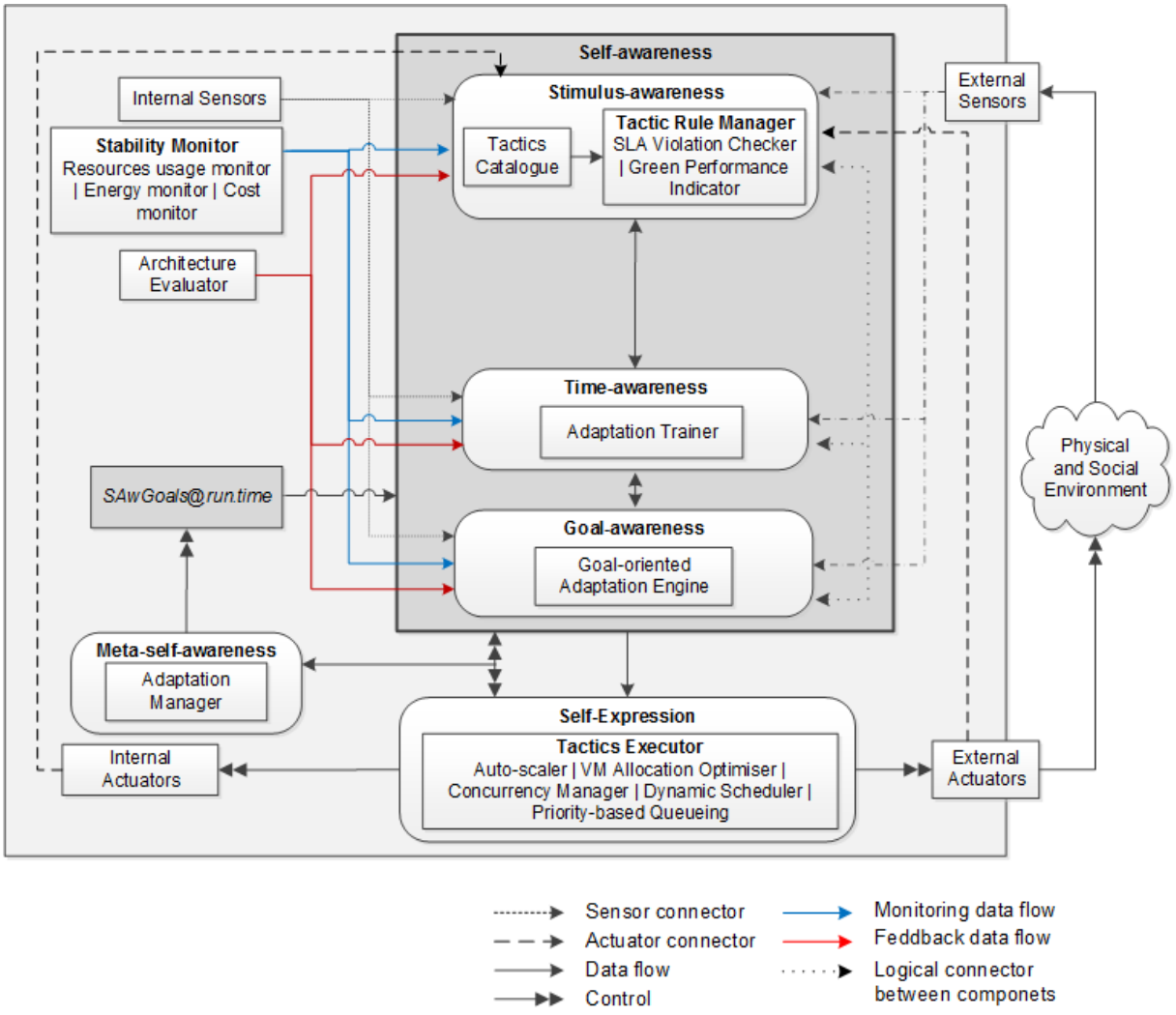

Figure 6.4: Evaluation Case: Application of the Reference Architecture

6.5.2 Application of the Goals Model

We define, hereunder, stability goals and runtime tactics determined above using our runtime goals modelling. Then, we provide an example of a runtime goal instance.

Stability goals Performance and QualityOfAdaptation are dedined as follows.

Goal Achieve [Performance]
Informal Definition
For every request received, the request processing should be accomplished within the performance parameters defined in the SLA of the client issuing the request.
Formal Definition
$\forall$ r:Request, c:Client
ExecuteRequest (r) $\Rightarrow \Diamond \leq$ c.SLA(ResponseTime)
Node identifier n_1:Self-awareArchitectureNode
Weight $w = 1.0$
Metric ResponseTime: Request $\rightarrow$ Time
def: the duration of processing request starting from client submitting the request till submitting the response back to the client
Objective
ResponseTime = ResponseTime $\leq$ c.SLA(ResponseTime)
Tactics T_1: VerticalScaling
T_3: HorizontalScaling
T_6: Concurrency
T_7: DynamicScheduling

Goal Achieve [QualityOfAdaptation]
Informal Definition
Any quality attribute should not be worse than 20% of the threshold in SLA for more than 300 seconds.
Formal Definition
$\forall$ r:Request, c:Client
QualityAttributes(r) $\Rightarrow \Diamond_{5min} \leq 20\%$ c.SLA(QualityAttributes)
Node identifier n_1:Self-awareArchitectureNode
Weight $w = 0.7$
Metric QualityAttribute: Request $\rightarrow$ Time
def: the quality attributes of processing requests should not be worse than 20% of the threshold in the client SLA for more than 300ms.
Objective
ResponseTime = ResponseTime $\Diamond_{300sec} \leq 20\%$ c.SLA(ResponseTime)
Tactics T_1: VerticalScaling
T_3: HorizontalScaling
T_6: Concurrency
T_7: DynamicScheduling

Runtime tactics VerticalScaling and VMsConsolidation are dedined as follows.

Tactic VerticalScaling
Unique identifier T1
Informal Definition
increase the number of VMs or their capacities
Object VMs
Pre-condition
$TotalCPUcapacity$ of running VMs $\leq$
$TotalCPUcapacity$ of hosts running in the datacenter
Limits $max(TotalCPUcapacity)$ of hosts running in the datacenter
Functionality
increaseCPUCapacity(vm: VM) $\vee$
increaseCoresNum(vm: VM) $\vee$
runNewVM()
Post-condition Waiting Requests are migrated to the new VM
Variations $T_{1.1}$: increase CPU capacity of 1 running VM
$T_{1.2}$: increase the number of cores of 1 running VM
$T_{1.3}$: add 1 VM to running VMs

Tactic VMsConsolidation
Unique identifier T4
Informal Definition
shut down hosts running least number of VMs and migrate their VMs to other hosts
Object Hosts, VMs
Pre-condition $number$ of hosts running in the datacenter ≥ 2
Limits min 1 host running in the datacenter $\wedge$
min 1 VM running
Functionality
migrateVMs(host: Host) $\vee$
shutdown(host: Host)
Post-condition Requests are migrated to VMs
Variations $T_{4.1}$: shutdown 1 host

An instance of the Runtime Goal Performance is defined as follows.

Goal $G_1(n_1, t_i)$
Client c:Client
Request r:Request
Objective ResponseTime$_c$ = ResponseTime(r) $\leq 15ms$
AdaptationAction $T_{1.3}$
SatisfactionDegree $v = 14ms$
Environment Runtime Goals $G_e(t_i) = \{G_1(N_2, t_i), G_1(N_3, t_i), ...\}$
Environment Runtime Tactics $T_e(t_i) = \{N_2.T_{4.1}, N_3.T_{1.3}\}$

6.6 Experimental Evaluation

The main objective of the experimental evaluation to examine the stability goals and assess associated overhead when using the instantiated architecture and goals modelling in comparison with a foundational self-adaptive architecture (described in section 5.4.2.1).

6.6.1 Experiments Setup

To conduct the experimental evaluation, we implemented the instantiated architecture using the widely adopted *CloudSim* simulation platform for cloud environments [5]. The benchmarks and testbed configuration of these experiments are as described in section 5.5.1.1 and 5.4.2.2 respectively. The initial deployment of the experiments is: 10 hosts running 15 VMs (5 x m4.large, 5 x m4.xlarge, 5 x m4.2xlarge). Initially, the VMs are allocated according to the resource requirements of the VM types. However, VMs utilise fewer resources according to the workload data during runtime, creating opportunities for dynamic consolidation.

We set the runtime goals model with stability attributes from the stability analysis results (section 4.4). Regarding the quality of adaptation, we challenge the experiments with the settling time (the time required by the adaptation system to achieve the adaptation goal to assure stable provision of attributes) and the accuracy of adaptation (how well the adaptation converges towards adaptation goals) [25]. The stability objectives and weights are hypothetical to stress the architecture, as defined in Table 6.2.

Table 6.2: Settings of Stability Goals

Attribute	**Description**	**Weight**	**Metric**	**Objective**
Performance	Response time	0.50	ms	25
Greenability	amount of energy consumed for operating hosts	0.20	kWh	25
Operational cost	cost of computational resources (CPUs, memory, storage, bandwidth)	0.20	$	50
Settling time	time required by the adaptation system to achieve stability goals	0.10	ms	
Accuracy of adaptation	how close adaptation goals are met within given tolerances	0.10	%	

6.6.2 Results of Stability Goals

We report, first, on the average of stability goals at each time interval. The average response time results for service types 1 and 2 are depicted in Figure 6.5. As shown in the figure, the self-aware architecture was able to result in more stable response time compared with the self-adaptive architecture in both service types, while the latter caused violation in response time in early time intervals when the peak workload started. At the same time, the self-aware architecture was also capable of stabilising the operational cost for longer time intervals than the self-adaptive architecture. It is worth noting that stabilising response time with energy consumption and cost at the same time is very challenging in case of peak workload, that is why the self-aware architecture had some violations during the highest peak load.

Next, we report the results of stability goals on average 30 runs of the experiments total results in Table 6.3. The average response time of all requests for each service type

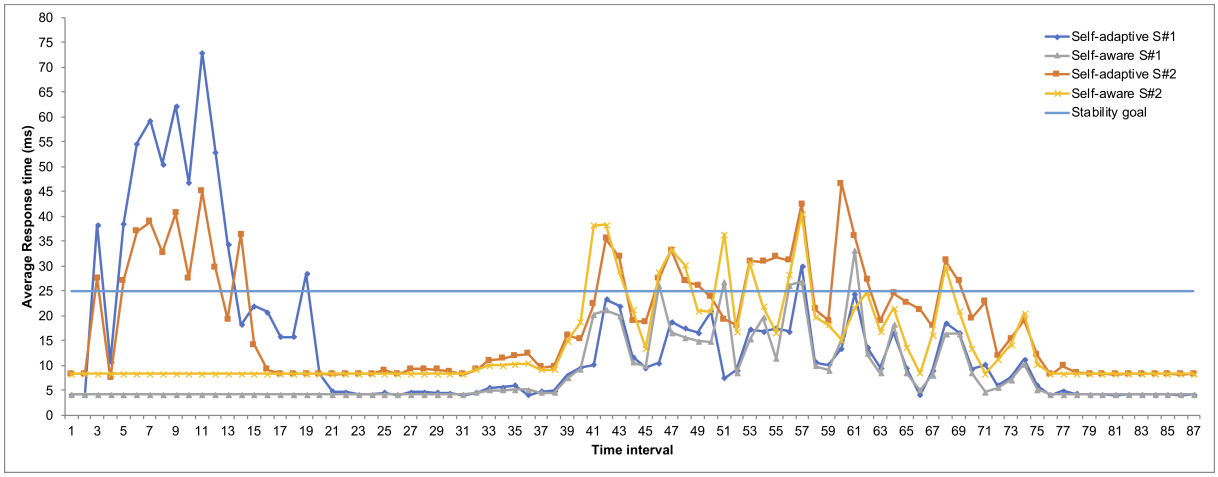

Figure 6.5: Average Response Time of Service Types 1 and 2 during Time Intervals

is much better achieved by the self-aware architecture (average 62.85 ms compared to 20.02 ms). This came with the price of higher operational cost (168.94 $ vs. 205.84 $) and adaptation overhead (as shown below). In such case, the self-aware architecture considered keeping the response time without violations while not fully stabilising the cost within the constraint, as the response time weight is higher. Meanwhile, the difference in response time is much bigger than the difference in cost and energy. While the energy consumption in self-adaptive architecture was less (24.96 kWh), the self-aware architecture was capable of keeping it within the stability goal (28.52 khW).

Table 6.3: Average Results of Stability Attributes

Stability Attributes	**S#**	**Architecture Pattern**	
		Self-adaptive	Self-aware
Response time (ms)	1	73.73	16.00
	2	63.49	22.92
	3	58.41	18.56
	4	58.90	21.10
	5	59.74	21.54
	avg.	62.85	20.02
Energy consumption (kWh)	1	23.80	28.52
	2	25.33	28.52
	3	25.02	28.52
	4	25.33	28.52
	5	25.33	28.52
	avg.	24.96	28.52
Operational cost ($)	1	137.18	193.44
	2	184.08	230.88
	3	159.02	199.38
	4	180.66	199.68
	avg.	168.94	205.84

6.6.3 Results of Adaptation Properties and Overhead

Table 6.4 shows the average results of adaptation properties. The accuracy of adaptation is shown in terms of the violation percentage in response time for all service types, and the settling time is shown as total time periods where the response time was violated. As shown in the table, the percentage of violations in response time is slightly higher in the case of self-aware architecture in all service types. But regarding the total periods of time where the response time was violated, the self-aware architecture was capable of keeping it much less than the self-adaptive architecture. For instance, response time violation in the case of service type 1 was 12.96% and 14.23% for the self-adaptive and self-aware architectures respectively, while the total time of violations was 11232 sec compared to 4320 sec. Meanwhile, the self-aware architecture violations were less for service types 2 and 4, with better settling time. This reflects the higher quality of adaptations and tactics selection.

With respect to the associated adaptation overhead (calculated by the time spent in the adaptation process), the average overhead of self-aware architecture is 251.62 sec on average of all service types, compared to 164.90 sec of the self-adaptive architecture. Yet, the difference in response time is much bigger than the difference in overhead (compared with Table 6.3).

Table 6.4: Average Results of Adaptation Properties

Adaptation Property	**S#**	**Architecture Pattern**	
		Self-adaptive	Self-aware
Accuracy of adaptation (%)	1	12.96	14.23
	2	26.44	24.40
	3	15.78	17.19
	4	20.90	20.91
	5	27.51	28.31
	avg.	20.72	21.01
Settling time (ms)	1	11232	4320
	2	24192	9504
	3	13824	5184
	4	19872	6048
	5	19008	8640
	avg.	72921.60	6739.20
Adaptation overhead (sec)	1	157.80	247.40
	2	168.50	260.60
	3	162.70	249.00
	4	167.40	249.60
	5	168.10	251.50
	avg.	164.90	251.62

6.6.4 Discussion

The proposed architecture with its generic components to embed runtime tactics have successfully instantiated many tactics for different quality and stability attributes and

enriched the self-aware patterns with self-management quality capabilities to meet the changing workload and stabilise quality requirements during runtime. Applied to the evaluation case, the hypothetical case has shown the potential to deliver values on the following attributes:

- *Efficiency.* The ability to incorporate a range of tactics for different stability attributes into the patterns diversify the catalogue space from which the adaptation actions could be selected and implemented during runtime to meet stability requirements under a dynamic workload.
- *Ease of application and use.* The structure of the tactics for different quality attributes was embedded efficiently within the generic components of the reference architecture. Their interaction specification also took place within the process flow while taking advantage of the self-awareness knowledge available from different self-awareness levels.
- *Multiple uses.* The generic approach for instantiating the architecture allowed featuring different combinations of self-awareness capabilities. Thus, incorporating tactics approach could be used in any of these patterns according to the requirements of the system, without unnecessary overhead caused by self-awareness components.

Generally, the proposed architecture and goals modelling for stability have proven feasibility when embedding tactics for different stability attributes. The proposed architecture tends to diversify the possible adaptation actions to be taken during runtime. The quantitative evaluation has proven the ability of the architecture and goals model to efficiently realise stability and enhance the quality of adaptation.

6.7 Related Work

In this section, we discuss related work in the context of architecture patterns and goals modelling.

6.7.1 Architecture Patterns and Tactics

A large body of research in architecture design has yielded the development of approaches for incorporating and using tactics in the context of software architectures. For instance, a systematic approach for building software architecture that embodies quality requirements using architectural tactics has been proposed [472] [473]. Other efforts focused on tactics for certain quality attributes, such as modifiability tactics [474], performance tactics [475]. Others tackled the application of tactics, such as analysing the application of tactics [476] and recommendation [477]. But stability has not been explicitly considered as a property in designing software architectures.

The self-adaptive architecture community has developed in the area of quality management. For instance, the Rainbow framework [478] was proposed to support such adaptation, where strategies in the adaptation engine are architectural tactics. A framework

for evaluating quality-driven self-adaptive software systems was proposed using a set of metrics to evaluate quality attributes and adaptation properties [25]. While literature has widely covered the incorporation of tactics in the context of software architectures, yet till recently, architecture patterns and tactics for self-adaptive and self-aware software have received little attention, as to the best of our knowledge [479] [3]. A reference architecture for self-adaptive software has been proposed based on reflection [462], but designing for stability with self-awareness has not been tackled yet.

6.7.2 Goals Modelling

Related work, geared towards runtime requirements modelling, are "models@run.time" and "self-explanation".

Models@run.time rethinks adaptation mechanisms in a self-adaptive system by leveraging on model-driven engineering approaches to the applicability at runtime [468]. This approach supports requirements monitoring and control, by dynamically observing the runtime behaviour of the system during execution. Models@run.time can interleave and support runtime requirements, where requirements and goals can be observed during execution by maintaining a model of the requirements in conjunction with its realisation space. The aim is to monitor requirements satisfaction and provide support for unanticipated runtime changes by tailoring the design and/or invoking adaptation decisions which best satisfy the requirements. Meanwhile, authors in [480] proposed a goal-oriented approach for systematically building architecture design from system goals.

In the context of self-adaptive systems, self-explanation was introduced to adaptive systems to offer interpretation of how a system is meeting its requirements, using goal-based requirements models [390]. Self-explanation focused mainly on explaining the self-adaptive behaviour of the running system, in terms of satisfaction of its requirements, so that developers can understand the observed adaptation behaviour and garner confidence to its stakeholders. Authors in [481] have theoretically revisited goal-oriented models for self-aware systems-of-systems. Goal models were also introduced as runtime entities in adaptive systems [482] and context-aware systems [483].

Though there has been growing research in runtime requirements engineering in the context of self-adaptive software systems, yet these models and approaches have limitations in enabling the newly emerged self-properties, i.e. self-awareness and self-expression. To the best of our knowledge, there is no research that tackled goals modelling for self-aware and self-expressive software systems, as well as realising the symbiotic relation between both.

6.8 Summary

In this chapter, we presented a reference architecture for architectural stability, using a generic approach for incorporating architecture tactics and QoS self-management components in self-aware architecture patterns. The approach is based on providing the self-aware patterns with a catalogue of architectural tactics designated to fulfil different

stability attributes. The stability-based adaptation will be performed during runtime by the awareness capabilities available in different patterns. Using the case of cloud architecture, quantitative experiments have proven enhancements in achieving stability and quality of adaptation using the reference architecture and goals modelling for stability.

Chapter 7

Reasoning about Architectural Stability

In questions of science, the authority of a thousand is not worth the humble reasoning of a single individual.

— Galileo Galilei

7.1 Introduction

Achieving behavioural stability for long-living software calls for more intelligent reasoning about stability on the long-run. We propose reasoning about stability based on self-awareness principles. Even though self-* properties have been widely investigated, no explicit attempt has considered self-awareness in achieving stabilisation. The latest emerging paradigm has proven effectiveness in managing trade-offs and deal with uncertainties. Benefiting from self-awareness primitives, self-adaptive architectures are enriched with goal-, time- and meta-self-awareness capabilities for managing stability goals, stability learning and managing associated trade-offs. We embed different computational intelligence techniques in self-awareness components for reasoning about stability.

Contributions. In more details, the main contributions are as follows.

- *Goal-awareness for managing stability goals.* With the typical key role of architectures in achieving quality requirements [319] [63] [377] [378], we can evidently agree that realising stability at the architecture level should be based on the quality requirements subject to stability [319] [378] [379], where requirements are the key to long-term stability and sustainability [215] [380]. We implement algorithms for realising symbiotic relation between runtime goals model (proposed in section 6.4) and self-awareness.

- *Time-awareness using online learning.* Stability learning is essential for achieving stability in the long-term by learning from historical information. We propose a learning technique based on Q-learning, a reinforcement learning technique that can

handle problems with stochastic transitions while learning how to act optimally in a controlled Markovian context [484] [485] [486]. Time-awareness is, then, capable of taking adaptation decisions converging towards stability by learning from historical information about adaptation actions and stability states.

- *Meta-self-awareness for managing trade-offs using model verification of stochastic games.* Achieving a stable state for the architecture requires an explicit trade-offs management between different quality attributes, so that the adaptation process converges towards runtime goals given runtime uncertainty. We build a runtime approach for managing trade-offs based on automatic verification of stochastic multi-players games (SMGs) using PRISM-games 2.0 [487] [488]. The approach allows reasoning about possible adaptations for multiple attributes on the long-run.

Organisation. The rest of this chapter is organised as follows. In section 7.2, we elaborate the technical contributions on self-awareness techniques for stability reasoning. Section 7.3 discusses experimental results of the evaluation case. We discuss related work in section 7.4. Section 7.5 concludes the chapter.

7.2 A Self-Awareness Assisted Framework for Reasoning about Architectural Stability

We employ the different awareness capabilities for reasoning about architectural stability. The goal-awareness embeds the symbiotic relation between the self-awareness component and runtime goals (discussed in section 7.2.1). The time-awareness implements an online learning algorithm to assist in making adaptation decisions leading to stability using historical information (discussed in section 7.2.2). The meta-self-awareness is assisted by probabilistic game-theoretic approach for managing trade-offs between different stability goals (discussed in section 7.2.3).

7.2.1 Goal-Awareness for Managing Stability Goals

As end-users' requirements change during runtime, there is a need to maintain the synchronisation between the goals model and the architecture [471]. We envision enriching the proposed architecture patterns and goals modelling by incorporating the symbiotic relation between runtime goals and self-awareness capabilities. The symbiotic relation promises more optimal adaptations and better-informed trade-off management decisions. It aims to keep the runtime goal model "live" and up-to-date, reflecting on the extent to which adaptation decisions satisfied the goal(s). The symbiotic relation, illustrated in Figure 7.1, is realised during runtime as follows.

1. Goals are defined and modelled in the *SAwGoals@run.time* component, with fine-grained knowledge representation relevant to the different levels of awareness.

2. Having goals information fed to the self-awareness component, a better-informed adaptation decision would be taken based on the learning of time-awareness and the runtime environment of interaction-awareness capabilities.

3. The selected tactic is executed by the self-expression component.

4. The execution trace is, then, fed back to the goals model to be kept in the log of the goal history.

5. The goal satisfaction is evaluated by the Architecture Evaluator component to be logged in the goal history.

6. The goal history is used, in turn, by time-awareness at the next time instance when selecting the appropriate tactic.

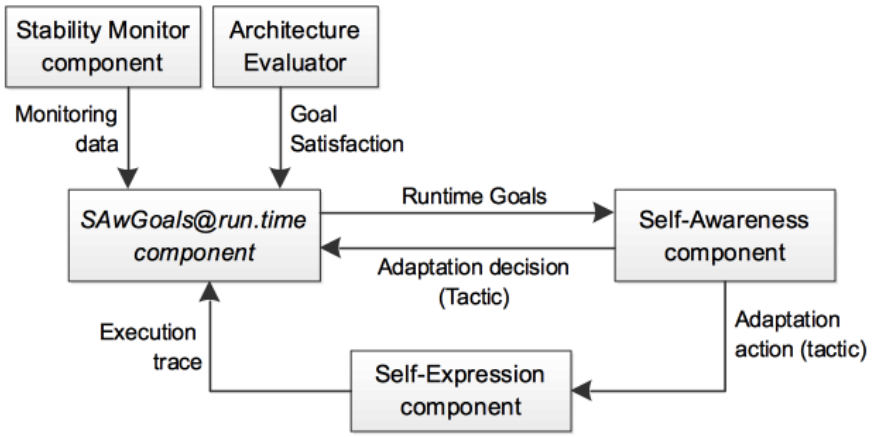

Figure 7.1: Symbiotic Relation between Runtime Goals and Self-Awareness

7.2.1.1 Algorithms for Realising Symbiotic Relation Between Runtime Stability Goals and Self-awareness

To realise the symbiotic relation, we provide algorithms to process the Runtime Goal Instance (Algorithm 7.3) and construct the Goal History (Algorithm 7.2).

Algorithm 1: Processing Runtime Stability Goal. This algorithm is launched to process the Runtime Goal Instance $G(n, t_i)$ at time instance t_i.

Algorithm 7.1 Process Runtime Stability Goal

```
1: procedure ProcessGoal(G_i = (G_id, N_id, t_i))
2:     get ObjectiveFunction(client c)
3: QoSMonitor:
4:     get MonitoringData(G)
5: Self-awarenessComp:
6:     if violation(G) then
7:         Identify set of possible tactics T(G)
8:         if TimeAwareness is enabled then
9:             get goal hisotry H(G)
10:        end if
11:        select tactic T_x ∈ T(G)
12: Self-expressionComp :
13:        execute tactic T_x
14:        get ExecutionTrace τ(G_i)
15: ArchitectureEvaluator:
16:        get GoalSatisfaction v(G)
17:    end if
18: end procedure
```

Algorithm 2: Constructing Stability Goal History. This algorithm constructs a change tuple for the goal G at each time instance t_i. Each change tuple records a log of the objective function, goals from the environment, set of tactics executed in the environment, the tactic executed, the execution trace and the goal satisfaction measure. These change tuples would form the goal history over the different time instances.

Algorithm 7.2 Construct Stability Goal History

```
1: procedure ConstructHistory(Goal G = (G_id, N_id))
2:     for each t_i do
3:         log time instance t_i
4:         log ObjectiveFunction(client c)
5:         log executed Tactic T_x
6:         log ExecutionTrace τ(G)
7:         get GoalSatisfaction v(G)
8:         if InteractionAwareness is enabled then
9:             log Environment Goals G_e = {G_1(n_1, i), G_2(n_2, i), ..., G_x(n_x, i)}
10:            log Environment Tactics T_e = {T_1, T_2, ..., T_x} for ∀ G ∈ G_e
11:        end if
12:    end for
13: end procedure
```

7.2.2 Time-Awareness for Stability Online Learning

Learning from historical information about adaptation actions and stability states, time-awareness is capable of taking adaptation decisions converging towards stability. We use a form of model-free reinforcement learning technique, that is "Q-learning". The technique does not require a model of the system (i.e. priori knowledge and computational demands) and can handle problems with stochastic transitions with the capability of learning how to act optimally in a controlled Markovian context [484] [485] [486].

Given the runtime uncertainty, we consider the system as a finite Markov decision process (FMDP), where the Q-learning can identify an optimal action-selection policy (i.e. adaptation action), where the expected value of the total reward return is the maximum achievable at the current state. The technique works during runtime by successively improving its evaluations of the quality of particular adaptation actions at particular states (i.e. online learning) [484] [485] [486].

7.2.2.1 Learning Model

Formally, the learning process involves a set of states S and a set of adaptation actions A, where each state $s \in S$ present the status of stability attributes and each action $a \in A$ is one of the possible different configurations of the architecture. A state s is a tuple of the different stability attributes, as such $< rt, c, ... >$ for response time and cost are the stability attributes. An action a is a tuple of the architecture configuration settings subject to adaptation, such as the number of PMs and the number of VMs $< pm, vm, ... >$.

When performing an action $a \in A$, the system transitions from state s_t to state s_{t+1}. Executing an action in a specific state is evaluated by a reward r value, where r could be a positive or negative value according to the benefit of the executed action. The Q-Learning algorithm has a function that calculates the quality of a state-action combination:

$$Q : S \times A \rightarrow \mathbb{R} \tag{7.1}$$

where $\mathbb{R}$ is the reward Q-matrix. The matrix is in the following format:

$$\mathbb{R} = \begin{bmatrix} r_{s_1,a_1} & r_{s_1,a_2} & \dots & r_{s_1,a_n} \\ r_{s_2,a_1} & r_{s_2,a_2} & \dots & r_{s_2,a_n} \\ & \ddots & & \\ r_{s_m,a_1} & r_{s_m,a_2} & \dots & r_{s_m,a_n} \end{bmatrix}$$

where the rows of the matrix represent the different states s of the system, the columns represent the possible adaptation actions a, and the matrix values are the learnt reward values r.

The Q function returns the reward used to provide the reinforcement and stand for the quality of an action taken in a given state. At each time instance t, the algorithm selects an action a_t, observes a reward r_t, enters to a new state s_{t+1} that depends on the previous state s_t and the selected action a_t, and the Q-matrix is updated using the weighted average of the old value and the new information, as follows:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)\, Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) \right) \tag{7.2}$$

where r_t is the reward observed for the current state s_t, α is the constant learning rate ($0 < \alpha \leq 1$) that determines the extent to which the newly acquired information overrides old information, and γ is the discount factor ($0 < \alpha \leq 1$) that determines the importance of future rewards. If the learning factor α is set $= 1$, the algorithm uses only the most recent information, and if $\alpha = 0$, this forces the algorithm to learn nothing and use historical information only. If the discount factor $\gamma = 0$, the algorithm will consider

only current rewards, and if $\gamma = 1$, the algorithm will use long-term high reward, which is the case of stability.

7.2.2.2 Online Learning Algorithm

The goal of the learning algorithm is to maximise the total reward in the future, by learning which action is optimal for each state. The optimal action for each state is the one that has the highest long-term reward, in order to achieve stability in the long-run. The reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state.

The matrix is initialised with a possibly arbitrary fixed value. For simplicity, we assume a certain number of states with ranges of each stability attribute and a certain set of configurations. These ranges of stability attributes and possible configurations could be easily refined by adding more columns and rows in the Q-matrix.

The algorithm keeps running while the system is online. First, the current state is observed, and the algorithm selects an action with the highest reward value among the set of actions for the current state using equation 7.2. Then, the new state and new reward are observed, and the Q-matrix is updated with new reward value.

Algorithm 7.3 Stability Q-Learning

procedure QLEARNING(S, A, α, γ)
Input:
    S : set of states
    A : set of actions
    γ : discount factor
    α : learning rate
Output:
    a' : new action
Local variables:
    $Q[S, A]$: Q-matrix
    s : previous state
    a : previous action
    r : reward
    s' : new state
initialise:
    $S = \{s_1 < rt, c, ... >, s_2 < rt, c, ... >, ...\}$
    $A = \{a_1 < pm, vm, ... >, a_2 < pm, vm, ... >, ...\}$
    get current state s
    repeat ▷ while online
        select action a' from possible actions for s
        $Q(s, a) \leftarrow (1 - \alpha)\, Q(s, a) + \alpha\, (r + \gamma\, max_{a'} Q(s', a'))$
        observe reward r and new state s'
        $s \leftarrow s'$
    until termination
end procedure

7.2.3 Meta-Self-Awareness for Managing Trade-offs between Stability Attributes

We investigate the use of game theory to achieve an equilibrium point between different stability quality attributes, i.e. modelling and analysing the consequent trade-offs between stability attributes given the uncertainty of the running environment. The proposed methodology considers the value implications of choosing an architectural tactic for adaptation with respect to multiple quality attributes subject to stability and potentially uncertain future runtime conditions. In more details, we tend to evaluate architectural tactics for their pay-off values and based on such an evaluation, an architectural tactic is selected in a way that supports the management of trade-offs between different stability goals. The goal is to select the tactic for better adaptation leading to the long-term welfare of the architecture. Architectural tactics are intended to aid in creating architectures that meet quality requirements [68]. Such tactics are employed to achieve a desired quality attribute behaviour, which, in turn, imparts utility to the architecture. The utility should not be in terms of one quality attribute, yet an aggregate utility comprehending multiple quality attributes.

We consider the continuous runtime process of managing trade-offs under the uncertainty of the environment as a *stochastic game*, where the players are the runtime stability attributes and their strategies are the possible adaptation actions. A central idea is that architectural decision, such as the application of a tactic for adaptation, is analogous to a game strategy. Quality attributes and their expected utility under uncertainty act as underlying assets for the valuation of architectural decisions, similar to the valuation of game strategies. This approach provides a quantitative decision for selecting architectural tactic based on the utility objectives and uncertainty of runtime workloads, quality goals and environmental changes. Part of the objectives is to evaluate the overall adaptation process and its implication for the long-term welfare of the architecture and goals fulfilment.

7.2.3.1 Problem formulation

Achieving runtime architectural stability among different stability attributes should involve a careful understanding of the relationship, impact, correlation and sensitivity among attributes subject to stability, as well as handling potential conflicts. Given the runtime uncertainty arising from many sources, the runtime stability is seen to be a probable behaviour rather than deterministic.

The proposed approach builds upon the framework for modelling, analysing and automatic verification of turn-based stochastic multi-players games (SMGs) [487]. A natural fit for modelling systems that exhibit probabilistic behaviour is using stochastic games [487] [489]. Probabilistic model checking provides verification of quantitative properties (stability gaols) and provides a means to synthesise optimal strategies to achieve these goals [489] and leave the architecture stable on the long-run.

A natural fit for modelling systems that exhibit probabilistic behaviour is adopting a game-theoretic perspective [487] [489]. In particular, *stochastic games* can used to model the self-adaptive (stochastic) system and its (conflicting) stability goals. Probabilistic model checking provides a means to model and analyse these systems, by providing ver-

ification of quantitative properties in probabilistic temporal logic [489]. PRISM-games tool, built on the code-base of PRISM model checker, provides modelling of quantitative verification for SMGs, where the games are specified using the PRISM modelling language [489]. In this tool, SMGs are described as a model composed of *modules*, where their state is determined by a set of *variables* and their behaviour is specified by a set of *guarded commands*, containing an optional action label, a guard and a probabilistic update for the module variables [489]:

$$[\texttt{action}]\ \texttt{guard} -> \ \texttt{prob}_1 : \texttt{update}_1 \ + \ \ldots \ + \ \texttt{prob}_n : \texttt{update}_n$$

PRISM-games' properties specification are written using a probabilistic temporal logic with rewards called rPATL [489] [487]. rPATL is an extension of the logic PATL [490], which is itself an extension of ATL [491], a widely used logic for reasoning about multi-player games and multi-agent systems [489]. Properties, quantitatively specified, in rPATL can state that a coalition of players has a strategy which can ensure that either the probability of an event's occurrence or an expected reward measure meets some threshold [487]. rPATL is a CTL-style branching time temporal logic that incorporates the coalition operator $\langle\langle C\rangle\rangle$ of ATL [491], the probabilistic operator $\mathsf{P}_{\bowtie q}$ of PCTL [492], and the reward operator $\mathsf{R}^r_{\bowtie x}$ from [493] for reasoning goals related to reward/cost measures [489]. Beside the *precise* value operators, rPATL also supports the quantification of maximum and minimum accumulated reward until a ϕ-state is reached that can be guaranteed by players in coalition C, noted as $\langle\langle C\rangle\rangle \mathsf{R}^r_{max=?}[\mathsf{F}^*\phi]$ and $\langle\langle C\rangle\rangle \mathsf{R}^r_{min=?}[\mathsf{F}^*\phi]$ respectively.

By expressing properties that enable us to quantify the maximum and minimum rewards a player can achieve, we can reason about different adaptation strategies and synthesise strategies that optimise stability rewards. This allows to choose an optimal adaptation action that would achieve stability attributes, and hence leave the architecture in a stable state in the long-run. The approach consists of SMG model (discussed in section 7.2.3.2) and strategy synthesis (section 7.2.3.3). Then, we describe the model specification (in section 7.2.3.4).

7.2.3.2 Stochastic Multi-Player Game Model

We model the self-adaptive system and its environment as two players of an SMG, in which the system's objective is reaching stability state, that is a goal state that maximises a utility/reward (i.e. achieve stability attributes), and the environment as an opponent whose actions cannot be controlled. In each turn, only one player can choose between different strategies, and the outcome can be probabilistic. The system can choose between a set of adaptation actions, i.e. adaptation tactics, to achieve stability goals, while the environment is considered as an adversary to the system.

Definition. (SMG) A *turn-based stochastic multi-player game* (SMG) is a tuple $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i\in\pi}, \Delta, AP, \mathcal{X}, r\rangle$, where Π is a finite set of players, $S \neq \emptyset$ is a finite set of states, $A \neq \emptyset$ is a finite set of actions, $(S_i)_{i\in\pi}$ is a partition of S, $\Delta : S \times A \to \mathcal{D}(S)$ is a partial transition function ($\mathcal{D}(S)$ denotes the set of discrete probability distributions over finite set S), AP is a finite set of atomic propositions, $\mathcal{X} : S \to 2^{AP}$ is a labeling

function, and $r : S \to \mathbb{Q}_{\geq 0}$ is a reward structure mapping each state to a non-negative rational reward.

In each state $s \in S$, the set of available actions is denoted by $A(s) = \{a \in A | \Delta(s, a) \neq \perp\}$, assuming that $A(s) \neq \emptyset$ for all states. The choice of action in each state s is under control of one player $i \in \Pi$, for which $s \in S_i$.

The set of players $\Pi = \{sys, env\}$ is formed by the self-adaptive system and its environment. The set of states $S = S_{sys} \cup S_{env}$ is formed of the states of the system S_{sys} and the states of the environment S_{env} ($S_{sys} \cap S_{env} \neq \emptyset$). The set of actions $A = A_{sys} \cup A_{env}$ is formed of the set of actions available for the system and the environment denoted by A_{sys} and A_{env} respectively. AP is the subset of all predicates that can be built over the state variables and includes the *goal* that is satisfied when achieving stability goals.

Definition. (Path) A *path* of SMG $\mathcal{G}$ is a possibly infinite sequence $\lambda = s_0 a_0 s_1 a_1 \ldots$, such that $a_j \in A(s_j)$ and $\Delta(s_j, a_j)(s_{j+1}) > 0$ for all j. $\Omega^+_{\mathcal{G}}$ is used to denote the set of finite states in $\mathcal{G}$.

r denotes the reward for labelling goal states with their associated utility. The reward of a state s is defined as $r(s) = \Sigma_{i=1}^{q} u_i(v_i^s)$ if $s \models$ (satisfies) *goal*, where $u_i \in [0, 1]$ is the utility function for the stability goal $i \in \{1, \ldots, q\}$, and v_i^s is the value of the state variable associated with the architectural property representing stability attribute i in state s.

Definition. (Reward structure) A reward structure for $\mathcal{G}$ is a function $r : S \to \mathbb{R}_{\geq 0}$ or $r : S \to \mathbb{R}_{\leq 0}$.

The reward structure is used to maximise or minimise the goals. A reward structure assigns values to pairs of states and actions.

Players of the game can follow strategies for choosing actions that result in achieving their goals.

Definition. (Strategy) A *strategy* for player $i \in \Pi$ in $\mathcal{G}$ is a function $\sigma_i : (SA) * S_i \to \mathcal{D}(A)$ which, for each path $\lambda.s \in \Omega^+_{\mathcal{G}}$ where $s \in S_i$, selects a probability distribution $\sigma_i(\lambda.s)$ over $A(s)$.

A strategy σ_i is memoryless if $\sigma_i(\lambda.s) = \sigma_i(\lambda'.s)$ for all paths $\lambda.s, \lambda'.s \in \Omega^+_{\mathcal{G}}$, and deterministic if $\sigma_i(\lambda.s)$ is a Dirac distribution for all $\lambda.s \in \Omega^+_{\mathcal{G}}$.

7.2.3.3 Strategy Synthesis

Reasoning about *strategies* is an fundamental aspect of SMGs model checking. rPATL queries check for the existence of a strategy that is able to optimise an objective or satisfies a given probability/reward bound [489]. Model checking also supports optimal *strategy synthesis* [489] for a given property. In our case, we use *memoryless deterministic strategies*, that resolve the choices in each state selecting actions based on the current state [489]. Such strategies are guaranteed to achieve the optimal expected rewards [489].

We perform strategy synthesis using *multi-objectives queries* supported by PRISM-games 2.0, by computing Pareto set or optimal strategies for managing trade-offs between multi-objective properties [488]. Multi-objectives queries are expressed as a boolean combination of reward-based objectives with appropriate weights [488], which allows reasoning

about the long-run average reward. Generally, higher weights are given to the stability of quality of service attributes (e.g. response time), as these are the main objective of adaptation.

Properties are specified as follows: $\langle\langle sys \rangle\rangle \mathsf{R}^{r}_{max=?}[\mathsf{F}^{c}\phi]$, to synthesise a strategy that maximises the utility rewards from all stability attributes, where ϕ state represents the state where adaptation goals are achieved. The multi-objective query to reason about stability multi-objective property is specified as follows:

$$\langle\langle sys \rangle\rangle (\mathsf{R}\{response\_time\}_{\leq v_1}[\mathsf{C}] \wedge \mathsf{R}\{energy\}_{\leq v_2}[\mathsf{C}])$$

where the targets $v_1, v_2, \ldots$ for the stability objectives are defined from Service Level Agreements (SLAs).

7.2.3.4 Model Specification

Our formal model is implemented using PRISM-games 2.0 [489] [488]. The state space and behaviours of the game are generated from the stochastic processes under the control of the two players of the game, the system and the environment. In more details:

The self-adaptive system (player *sys*) controls the process that models the adaptation controller of the self-adaptive system, which is responsible about triggering and executing adaptation actions. The set of actions available to the system A_{sys} are the set of adaptation tactics defined in the adaptation controller, e.g. horizontal scaling, vertical scaling, increasing VM capacity. Each action $a \in A_{sys}$ command follows the pattern:

$$[\mathtt{a}]\ \mathtt{C_a} \wedge \neg\mathtt{goal} \wedge \mathtt{t} = sys\ -{>}\ \mathtt{prob}^{1}_{\mathtt{a}}\ : \mathtt{update}^{1}_{\mathtt{a}} \wedge \mathtt{t}' = env\ +\ \ldots\ +\ \mathtt{prob}^{n}_{\mathtt{a}}\ :\ \mathtt{update}^{n}_{\mathtt{a}} \wedge \mathtt{t}' = env \quad (7.3)$$

where $\mathtt{C_a}$ is the constraints for executing the tactic $\mathtt{a}$ (e.g. capacity of a physical machine (PM) to accommodate virtual machines (VMs)), a predicate $\neg\mathtt{goal}$ to prevent expanding the state space beyond the satisfaction of the adaptation goal, $\mathtt{t} = env$ constraints the execution of actions of the player in turn $\mathtt{t}$ to states $s \in S_{syss}$. The command includes the possible updates $\mathtt{update}^{i}_{\mathtt{a}}$, corresponding to one probabilistic outcome for the execution of $\mathtt{a}$, along with their associated probabilities $\mathtt{prob}^{i}_{\mathtt{a}}$. And the turn is given back to the *env* player by the control variable $\mathtt{t}'$.

The environment (player *env*) controls the process that models potential disturbances to the stability of the system that are out of the system's control, e.g. VM failure, server fault, network latency. The environment process is specified as a set of commands with asynchronous actions $a \in A_{env}$, and its local choices are specified non-deterministically to obtain a rich specification of the environment's behaviour. Each

command follows the pattern:

$$[\mathtt{a}]\ \mathtt{C}^{\mathtt{e}}_{\mathtt{a}} \wedge \neg \mathtt{end} \wedge \mathtt{t} = env\ -{>}\ \mathtt{prob}^{1}_{\mathtt{a}}\ : \mathtt{update}^{1}_{\mathtt{a}} \wedge \mathtt{t}' = sys\ +\ \ldots\ +\ \mathtt{prob}^{\mathtt{n}}_{\mathtt{a}}\ :\ \mathtt{update}^{\mathtt{n}}_{\mathtt{a}} \wedge \mathtt{t}' = sys \quad (7.4)$$

where $\mathtt{C}^{\mathtt{e}}_{\mathtt{a}}$ is the environment constraints for the execution of action $\mathtt{a}$, $\neg\mathtt{end}$ prevents the generation of further states, and $\mathtt{t} = env$ constraints the execution of actions of the player in turn to states $s \in S_{env}$. The command includes the possible updates $\mathtt{update}^{\mathtt{i}}_{\mathtt{a}}$, corresponding to one probabilistic outcome for the execution of $\mathtt{a}$, along with their associated probabilities $\mathtt{prob}^{\mathtt{i}}_{\mathtt{a}}$. And the turn is given back to the system player.

The SMG model consists of the following modules:

Players definition. Listing 7.1 shows the definition of the stochastic game players: player env which is control of the actions that the system environment can take, and player sys which controls the actions to be taken by the adaptation controller and the execution of adaptation tactics. The global variable t is used to control turns in the game, alternating between the system and the environment.

Listing 7.1: Players definition in PRISM-games 2.0

```
1 player env environment [] endplayer
2 player sys system [increase_pm_num], [decrease_pm_num], [increase_vm_num],
    [decrease_vm_num], [increase_vm_cap], [decrease_vm_cap] endplayer
3 const TURN_SYS, TURN_ENV;
4 global t:[TURN_SYS..TURN_ENV] init TURN_ENV;
```

Environment. The environment module (encoding shown in Listing 7.2) allows obtaining a representative specification of the system's environment, introducing disturbance to the stability of the system. This is done using variables that represent configurations that might affect stability, e.g. changing the number of VMs, changing the number of PMs. These behaviours are parametrised by the constants: MAX_TOTAL_VM_NUM and MAX_TOTAL_PM_NUM that constraints the maximum number of VMs and PMs respectively that the environment can use to introduce disturbance, MAX_TOTAL_VM_CAP and MAX_TOTAL_PM_CAP that constraints the maximum capacity of VMs and PMs respectively, MAX_VM_CHANGE is the maximum numbers of virtual machines (VMs) that the environment can change to interrupt the system execution and cause instability, MAX_PM_CHANGE is the maximum number of physical machines (PMs) that the environment can change to cause instability in QoS provision (e.g. response time). For simplicity, we consider all PMs and VMs are of the same capacity.

The current state of the environment is defined using the variables: current_vm_num, current_pm_num corresponding to the changes introduced by the environment at the current turn with respect to the number of VM and PM respectively, total_vm_cap and total_pm_cap that keep track of the total capacity of VM and PM respectively.

At each turn, the environment action is setting the disturbance variables (changing system configurations) using the command in Listing 7.2 line 12. First, the guard checks

that: (i) it is the turn of the environment (t=TURN_ENV), (ii) an absorbing state has not been reached yet (!end), and (iii) the total number of VMs and PMs as well as their total capacities will not exceed the maximum specified for all types of disturbance. If the guard conditions are satisfied, the command: (i) sets the current configuration variables (e.g. d_{vm}), (ii) updates the total capacity variables with the current disturbance variables, and (iii) gives the turn to the system (t'=TURN_SYS).

Listing 7.2: Environment module

```
1  const MAX_VM_CHANGE, MAX_PM_CHANGE;
2  const MAX_TOTAL_VM_NUM, MAX_TOTAL_PM_NUM,
3          MAX_TOTAL_VM_CAP, MAX_TOTAL_PM_CAP;
4
5  module environment
6  current_vm_num: [1..MAX_TOTAL_VM_NUM] init 1;
7  current_pm_num: [1..MAX_TOTAL_PM_NUM] init 1;
8  total_vm_cap: [1..MAX_TOTAL_VM_CAP] init 1;
9  total_pm_cap: [1..MAX_TOTAL_PM_CAP] init 1;
10 [] (t=TURN_ENV)&(!end)&
11     (d_vm<MAX_VM_CHANGE)&
12     (d_pm<MAX_PM_CHANGE)&
13     (d_vm+current_vm_num<MAX_TOTAL_VM_NUM)&
14     (d_pm+current_pm_num<MAX_TOTAL_PM_NUM) ->
15     (current_vm_num=current_vm_num+d_vm)&
16     (current_pm_num=current_pm_num+d_pm)&
17     (total_vm_cap=current_vm_num*cap)&
18     (total_pm_cap=current_pm_num*cap)&
19     (t'=TURN_SYS);
20 endmodule
```

System. The system module models the behaviour of the system, including the adaptation controller and the execution of adaptation tactics (Listing 7.3). This is parametrised by the constants: (i) MIN_PM_NUM and MAX_PM_NUM whcich specify the minimum and maximum number of PMs, (ii) MIN_VM_NUM and MAX_VM_NUM which are the minimum and maximum number of VMs that PMs can accommodate, (iii) MIN_PM_CAP and MAX_PM_CAP is the minimum and maximum computational capacity of a PM configuration, (iv) MIN_VM_CAP and MAX_VM_CAP is the minimum and maximum computational capacity of a VM configuration, (v) STEP_NUM and STEP_CAP which are used to increase or decrease configuration, and (vi) INIT_PM_NUM, INIT_VM_NUM, INIT_VM_CAP, INIT_PM_CAP for the initial configuration of the architecture with respect to PMs, VMs and VMs capacity.

The variables of the module represent the current configuration of the architecture (pm_num, vm_num, pm_cap, vm_cap), the current provisioned quality of service (respnse_tine, energy, cost), and quality of adaptation (settling_time, resources_overshoot, adaptation_frequency). To update the value of quality variables, we employ multiple $M/M/1$ queueing model (from our earlier work [48]) to compute them based on the current architecture configuration and the request arrivals.

Listing 7.3: System module

```
const MIN_PM_NUM, MAX_PM_NUM, MIN_VM_NUM, MAX_VM_NUM,
    MIN_VM_CAP, MAX_VM_CAP, MIN_PM_CAP, MAX_PM_CAP,
    STEP_NUM, STEP_CAP;
const INIT_PM_NUM, INIT_VM_NUM, INIT_PM_CAP, INIT_VM_CAP;

module system
pm_num: [1..MAX_PM_NUM] init INIT_PM_NUM;
vm_num: [1..MAX_VM_NUM] init INIT_VM_NUM;
pm_cap: [1..MAX_PM_CAP] init INIT_PM_CAP;
vm_cap: [1..MAX_VM_CAP] init INIT_VM_CAP;

respnse_tine, energy, cost;
settling_time, resources_overshoot, adaptation_frequency;

[] (t=TURN_SYS)&(goal)&(!end) -> (t'=TURN_ENV);
    [increase_pm_num](pm_num<MAX_PM_NUM) ->
        (pm_num=pm_num+STEP_NUM);
    [decrease_pm_num](pm_num>MIN_PM_NUM) ->
        (pm_num=pm_num-STEP_NUM);
    [increase_vm_num](vm_num<MAX_VM_NUM) ->
        (vm_num=vm_num+STEP_NUM);
    [decrease_vm_num](vm_num>MIN_VM_NUM) ->
        (vm_num=vm_num-STEP_NUM);
    [increase_vm_cap](vm_cap<MAX_VM_CAP) ->
        (vm_cap=vm_cap+STEP_CAP);
    [decrease_vm_cap](vm_cap>MIN_VM_CAP) ->
        (vm_cap=vm_cap-STEP_NUM);
endmodule
```

Properties and Rewards. To perform adaptations leading to stability and managing trade-offs between its attributes, we use rPATL for the specification stability properties. These properties are used as input to PRISM-games, which can synthesise optimal adaptation actions for the attributes subject to stability. We use *long-run* properties from PRISM-games 2.0 (an extension for PRISM-games) [488], which allow expressing properties of autonomous systems that run for long periods of time and specify measures, such as energy consumption per time unit [488].

The effect of adaptation strategies on stability goals is encoded using a reward structure that assigns real-values of stability goals [488]. We use long-run average reward for expressing cumulative rewards towards stability. Each stability goal has a *target* value $\mathtt{v}$ for a reward value as a maximum or minimum. Goals for the expected long-run average reward $\mathtt{r}$ is expressed as $\mathtt{R}\{\text{“}\mathtt{r}\text{”}\}_{\geq \mathtt{v}}[\mathtt{S}]$, where $\mathtt{S}$ denotes long-run rewards. Satisfaction objectives for long-run rewards are expressed as $\mathtt{P}_{\geq 1}[\mathtt{R}(\mathtt{path})\{\text{“}\mathtt{r}\text{”}\}_{\geq \mathtt{v}}[\mathtt{S}]]$.

7.3 Experimental Evaluation

The main objective of the experimental evaluation is to examine stability when using different self-awareness capabilities for reasoning about stability and to assess associated

overhead. We compare goal-, time- and meta-self-awareness with stimulus-awareness (as a foundational self-adaptive capability).

7.3.1 Experiments Setup.

We use the cloud architecture instantiated in section 6.5 and stability objectives defined in Table 6.2. The proposed self-awareness techniques are implemented in the corresponding components, i.e. goals management and online learning are implemented in the goal- and time-awareness component. Regarding the trade-offs management, we used PRISM-games 2.0.beta3 off-the-shelf, where we implemented our model and run it on the same machine with OS X10.13.4 and exported the outcome strategies to the simulator for performing adaptations.

The benchmarks and testbed configuration used are described in section 5.5.1.1 and 5.4.2.2. The initial deployment of the experiments is: 10 hosts running 15 VMs (5 x m4.large, 5 x m4.xlarge, 5 x m4.2xlarge). Initially, the VMs are allocated according to the resource requirements of the VM types. However, VMs utilise fewer resources according to the workload data during runtime, creating opportunities for dynamic consolidation.

7.3.2 Results of Stability Attributes

The average of response time, energy consumption and operational costs are depicted in Figure 7.2, 7.3 and 7.4 respectively. On average, the self-awareness capabilities outperformed the self-adaptive one in keeping response time (highest priority) within stability objective. As shown in Figure 7.2, time-awareness achieved the best response time for all service types. Meanwhile, meta-self-awareness was capable of achieving the best performance for service type 2 and 5 which require the higher computational resources.

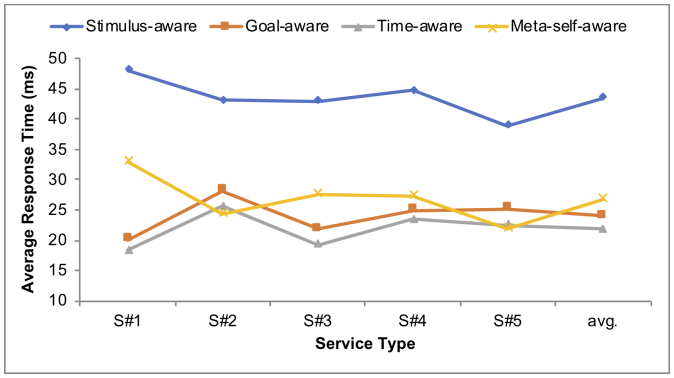

Figure 7.2: Average Results of Response Time

Regarding energy consumption, while all awareness algorithms succeeded in maintaining the energy consumption stability objective, time-awareness has consumed less energy reflecting the minimal number of PMs running (resources overshoot). This is due to performing adaptations that are capable of keeping stability goals for longer periods. Meanwhile, goal-awareness used the highest number of hosts, due to more frequent adaptation

(frequent shut-down and re-run of hosts) to keep stability goals. Meta-self-awareness was capable of maintaining the trade-offs between energy consumption and response time.

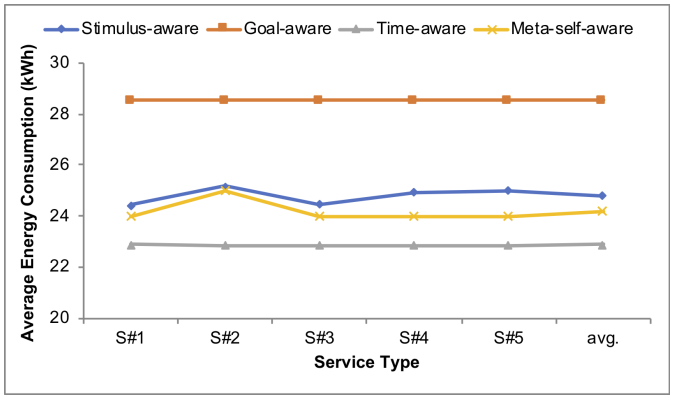

Figure 7.3: Average Results of Energy Consumption

Similar to energy consumption, operational costs (reflecting the number of VMs running) was better achieved by time-awareness, followed by meta-self-awareness. Goal-awareness has the highest cost, even though within stability objective.

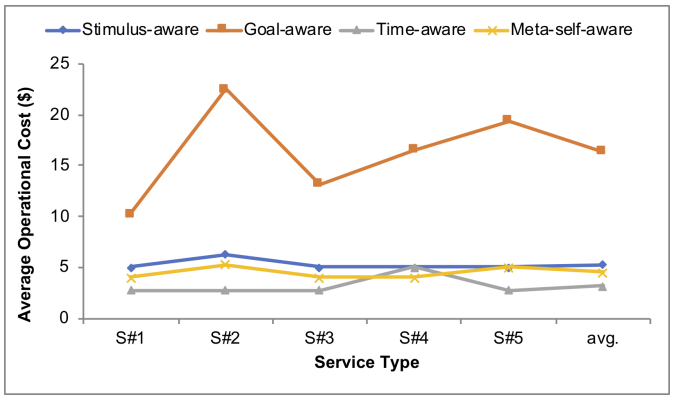

Figure 7.4: Average Results of Operational Cost

7.3.3 Results of Adaptation Properties and Overhead

Given the direct impact of the frequency of adaptation on architectural stability, we evaluate the number of adaptation cycles taken by each capability. As shown in Figure 7.5, time-awareness performed the least number of adaptation cycles, followed by meta-self-awareness. Meanwhile, goal-awareness is higher and close to self-adaptive, but achieved better response time than self-adaptive.

We also evaluated the adaptation overhead by calculating the total time spent by the architecture in the adaptation process. Figure 7.6 shows the overhead of each service type and their average. As goal-awareness performs pro-active adaptations for keeping stability gaols, its overhead is higher than self-adaptive (127.78 vs. 123.78 sec on average), which obviously resulted in better response time. Meta-self-awareness is on average of all

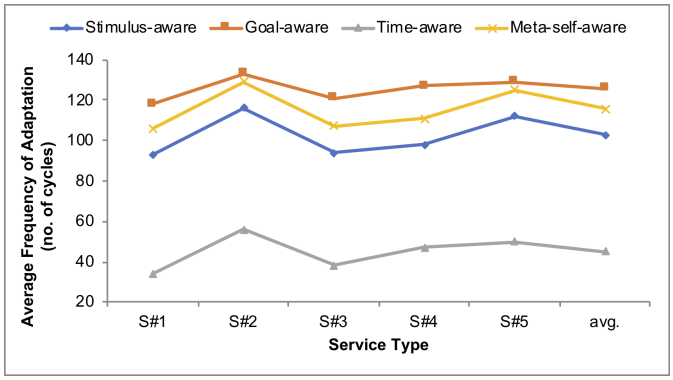

Figure 7.5: Average Results of the Frequency of Adaptation

capabilities (103.78 sec on average), while time-awareness has achieved the lower overhead (75.42 sec on average).

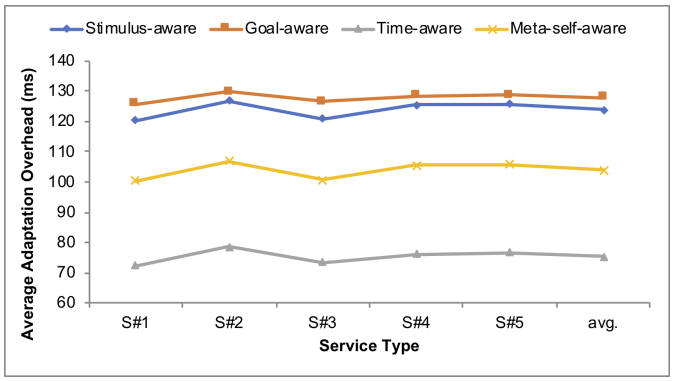

Figure 7.6: Average Results of Adaptation Overhead

7.3.4 Discussion

The proposed framework with different self-awareness capabilities has successfully achieved stability in terms of quality attributes and adaptation properties under runtime changing workload. Evaluating the features of the proposed framework is summarised as follows. First, different self-awareness principles were capable of successfully achieving stability attributes, combining quality attributes subject to stability and quality of adaptation. The generic framework allowed featuring different combinations of self-awareness capabilities for reasoning about long-term stability using machine learning and stochastic games techniques. Further, these reasoning techniques could be extended easily. Also, different stability goals could be easily configured, and other reasoning techniques could be employed. The proposed framework and architecture are generic for using one single self-aware capability, as well as switching between different capabilities during runtime.

Generally, the proposed approaches have proven feasibility in reasoning about stability during runtime, where the implemented components tend to make more intelligent adaptation actions. The quantitative evaluation has proven their ability to efficiently

reason about stability, avoid unnecessary frequent adaptations and minimise adaptation overhead and resources overshoot.

7.4 Related Work

In this section, we discuss related work to learning for self-adaptation (section 7.4.1) and trade-offs management in self-adaptive systems (section 7.4.2).

7.4.1 Learning for Self-Adaptation

Learning for self-adaptation has been studied by a number of researchers using different learning techniques for different purposes. For instance, a learning approach for engineering feature-oriented self-adaptive systems has been proposed in [494], learning revised models for planning self-adaptive systems [495], modelling self-adaptive systems with Learning Petri Nets [496], and handling uncertainty using self-learning fuzzy neural networks [497],

Focusing on the dynamic learning behaviour during runtime operation of adaptive systems, Yerramalla et al. [307] 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. Yet, quality of adaptation has not been considered in the stability behaviour. A reinforcement learning-based approach has also been proposed for planning architecture-based self-management [498]. Meanwhile, the behavioural stability aspect we are seeking has not been learnt online.

7.4.2 Trade-offs Management

Research has encountered many efforts for managing architectural trade-offs and the field has attracted a wide range of researchers and practitioners. Seminal works for trade-offs management include Architecture Tradeoff Analysis Method (ATAM) [320], Cost Benefit Analysis Method (CBAM) [323], PerOpteryx [403], the work of Kazman et al. [427] and the Quality-attribute-based Economic Valuation of Architectural Patterns [333]. Despite the maturity of research in evaluating and analysing architecture trade-offs, self-adaptive architectures call for special treatment, since self-adaptation has been primarily driven by the need to achieve and maintain quality attributes in the face of the continuously changing requirements and uncertain demand at runtime, as a result of operating in dynamic and uncertain contexts.

In our systematic review on trade-offs management for self-adaptive architectures (summary in Appendix C), we differentiated between approaches for design-time and runtime. By design-time, we mean trade-offs management is considered while evaluating the architectural design alternatives and making architectural decisions. The runtime is meant to be managing trade-offs while the system is operating, and the change requests

are implemented. Our findings show some attention given for explicit consideration of trade-offs management at runtime. Examples include [499] [500] [501] [502] [503]. But analysing the research landscape [43], our observation is that there is an adoption for the general "self-adaptivity" property without a discrete specialisation on self-* properties. The generality also applies to the quality attributes considered in trade-offs management. When considering certain qualities, they tend to be limited to two or three attributes, as explicit examples. As a general conclusion, the current work tends to be a solution for trade-offs management that act on trade-offs, not fundamental work that changes the architectural self-adaptivity. Although the studies found have provided much that is useful in contributing towards self-adaptive architectures, it has not yet resolved some of the general and fundamental issues in order to provide a comprehensive, systematic and integrated approach for runtime support for change and uncertainty while managing trade-offs.

With respect to considering multiple quality concerns, Cheng et al. [504] have presented a language for expressing adaptation strategies to calculate the best strategy for decision-making to be carried out by system administrators. Though this work has considered multiple QoS objectives and represented uncertainties in adaptation outcome, it is useful only for human operators use, not autonomous use during runtime. The work of Camara et al. [351] [505] has employed stochastic games for proactive adaptation, to balance between the cost and benefits of a proactive approach for adaptation. Meanwhile, self-adaptive architectures need to ensure the provision of multiple quality attributes. This also requires considering the quality of adaptation [25], as well as the cost and overhead of adaptation. Yet, our work considers architectural stability in terms of quality provision and quality of adaptation, using stochastic games with multi-objectives queries and long-run rewards.

7.5 Summary

In this chapter, we proposed self-awareness techniques for reasoning about architectural behavioural stability during runtime. We proposed symbiotic relation between goals model and goal-awareness for managing stability goals during runtime. We also implemented an online learning technique for reasoning about stability in the long-run while learning from historical information. Trade-offs between different stability attributes are managed using model verification of stochastic games. Using the case of cloud architecture, quantitative experiments have proven enhancements in achieving stability and quality of adaptation when using different self-awareness techniques.

Chapter 8

Systematic Approach for Evaluating Architectural Stability

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

— Sir Winston Churchill

8.1 Introduction

Given the architectures evaluation approaches found in the literature, stability evaluation was limited to the architecture's structure at design-time, and evaluation of behavioural stability was not explicitly tackled. Further, while architectures have been derived and evaluated for their fitness to quality attributes [427] [370], yet stability is also not explicitly considered. The shortcoming of current software engineering practices regarding stability is that the stable provision of certain critical quality attributes (e.g. response time for real-time systems) has not been explicitly considered in architecture evaluation, neither during design-time nor at runtime. This imposes new questions on how to evaluate architecture stability during design-time and while the system is in operation. In particular, practitioners and architects are challenged by how they can systematically evaluate stability and make architectural decisions that are stable and dependable in supporting likely changes in requirements and the environment.

To address this problem, we propose an evaluation framework for evaluating architectural behavioural stability. The proposed framework contributes to the gap of misconception when evaluating stability and helps in unifying concerns when evaluating stability. The proposed framework can assist architects and practitioners in explicitly addressing stability as a software property. Such evaluation would help in enhancing the efficiency of the architecture decisions and runtime operation, preventing the architecture from drifting and phasing-out as a consequence of the continuous unsuccessful provision of quality requirements.

Contributions. The main contributions of this chapter include:

- a systematic approach for evaluating architectural stability, following the "ISO/IEC 42030, Systems and Software Engineering —Architecture Evaluation" standards for general architectural evaluation [506]. Following such standards allows having a near-to-completeness standards-conforming systematic approach for stability evaluation. The framework explicitly addresses the process of planning, execution and documentation of behavioural stability evaluations. The main components of the framework are: context of evaluation, stability evaluation and stability attributes analysis.

- addressing stability evaluation in the different phases of the software lifecycle. We explicitly discuss the framework components to evaluate architectural stability during design-time and runtime for making architectural and operational decisions respectively.

- evaluation of applicability to self-adaptive cloud architectures case and experimental evaluation of runtime stability.

- a novel symbiotic simulation environment for self-adaptive and self-aware cloud architectures, to be employed for stability evaluation. As part of the evaluation, assessment approaches are required by the ISO standards [506] that have suggested discrete-event simulations as one of the feasible tools for architects to collect assessment results and to make decisions or draw conclusions about the architecture performance. We consider symbiotic simulation as a powerful tool for stability assessment, because of their ability to dynamically incorporate real-time data, providing the system with the effects of decisions on stability made by the simulation [450] [451] [452].

Organisation. The chapter is organised as follows. Section 8.2 describes the framework and its components. In section 8.3, we discuss stability evaluation in the software lifecycle. Section 8.4 evaluates our framework using the case of self-adaptive cloud architectures. We discuss related work in section 8.6. Section 8.7 concludes the chapter.

8.2 Architectural Stability Evaluation Framework

The framework aims to provide a systematic approach for evaluating architectural stability, in order to complement and usefully support architecture evaluations. Employing this systematic approach provides the basis on which to compare, select or create stability assessment and related attributes analysis. The framework addresses the planning, execution and documentation of architectural stability evaluations.

8.2.1 Conceptual Model

Various methods for evaluating software architectures have been defined to support holistic reasoning and decision making. We adhere to the holistic ethos in defining our framework for evaluating architectural stability. Our framework extends the "ISO/IEC 42030, Systems and Software Engineering —Architecture Evaluation" [507] [506]. More specifically, we follow the overall conceptual framework of the ISO/IEC standards for general architectural evaluation [507] [506] to generate a standards-conforming framework for stability evaluation. Following such standards allows having a near-to-completeness systematic approach for stability evaluation. We borrow the basic concepts from the ISO/IEC standards to be employed with the focus on architectural stability.

Figure 8.1 traces the progress of architectural stability evaluation through the conceptual framework. Following the ISO/IEC standards [507] [506], which considers stakeholders' concerns as a base of the evaluation, we adopt the concept of concerns to represent the assurances about the behaviour of the architecture that stakeholders expect to obtain. These concerns are framed into stability evaluation objectives, that drive the evaluation approach. Stability evaluation objectives are traced down to stability attributes analysis and assessment objectives, in order to express them in terms of attributes of interest. Stability attributes of interest cover the attributes that the architecture's behaviour is expected to exhibit while operating in its environment. The results of the attributes analysis and assessment are used to inform the stability evaluation approach, in order to determine whether stability concerns of the stakeholders are satisfied, for decision making and reporting purposes.

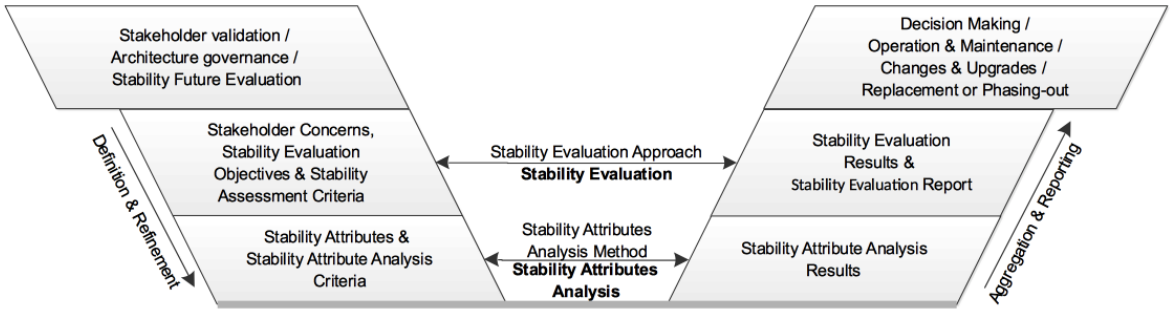

Figure 8.1: Progress of Stability Evaluation

The overall conceptual model is depicted in Figure 8.2[1]. Table 8.1 illustrates the mapping between the ISO/IEC standards for general architectural evaluation and the Stability Evaluation Framework[2]. The *Stability Evaluation Framework* aggregates three main components: (i) context of evaluation, (ii) stability evaluation approaches and their assessment criteria, and (iii) stability attributes analysis methods and their analysis criteria. Details of these components are presented in the next sections respectively.

[1]Figures use the conventions for class diagrams defined in ISO/IEC 19501 [508]. We use coloured fill shapes to indicate entities that are outside the boundaries of the model depicted in the figure.

[2]Concepts in the Architecture evaluation column are referenced from [507] [506]

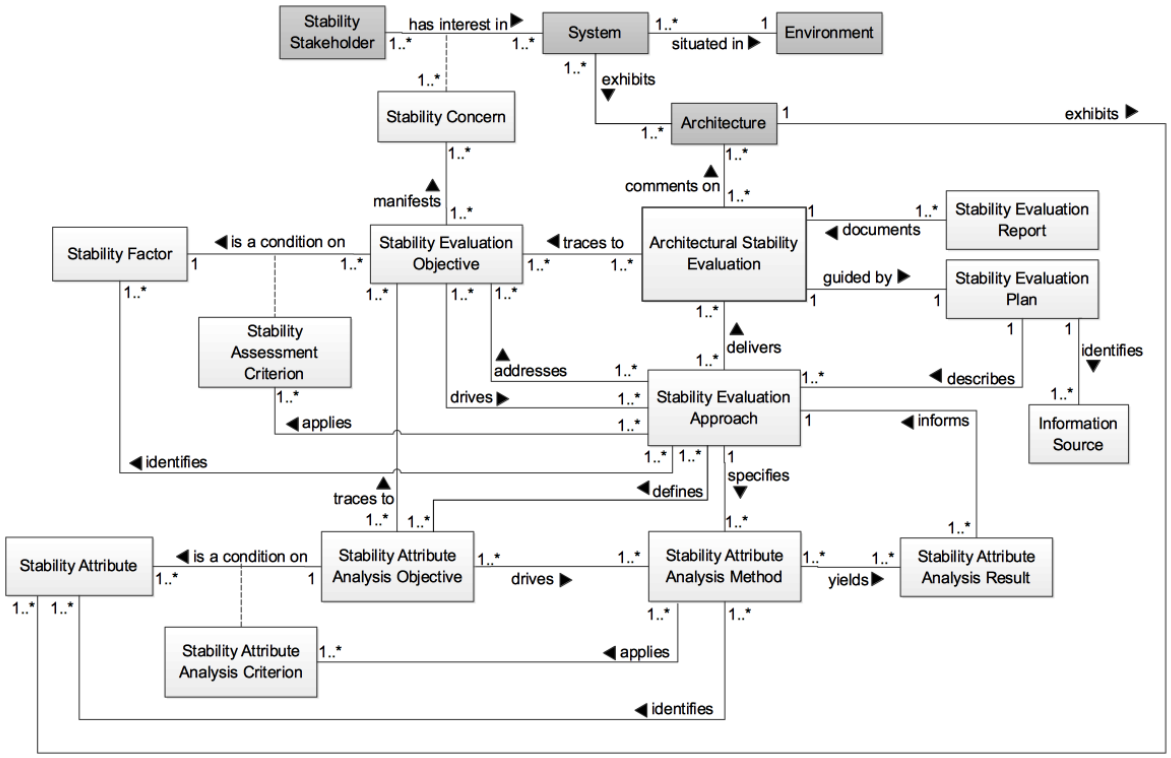

Figure 8.2: Conceptual Model of Stability Evaluation Framework

Table 8.1: Mapping of the ISO/IEC Standards for General Architectural Evaluation and the Stability Evaluation Framework

Concept	**Architecture Evaluation**	**Stability Evaluation**
Evaluation	the degree to which the architecture meets end-user needs, expectations or requirements	the degree to which the architecture is able to maintain the expected behaviour stable
Stakeholders	individual, team, organisation, or classes thereof, having an interest in a system	the entities that have interest in the architecture's behaviour and related concerns
Concerns	interest in a system relevant to one or more of its stakeholders	represent the assurances about the behaviour of the architecture that stakeholders expect to obtain
Evaluation Objective	manifests one or more stakeholder concerns	define how stability concerns will be addressed
Evaluation Approach	defines value assessment objectives deriving them from the evaluation objectives, and specify value assessment methods and related information sources to address evaluation objectives	describes how stability information will be gathered and processed
Factor	represents one or more stakeholder concerns, traceable to one or more architecture evaluation objectives, is used by a set of value assessment objectives to drive the value assessment	expresses a stability concern or stability advantage associated with the architecture
Evaluation Plan	includes the architecture evaluation objectives, and their relative importance, and the evaluation scope	guides the derivation of stability evaluation
Evaluation Report	documents the scope and objectives for the architecture evaluation, stakeholders' concerns addressed by the architecture evaluation, and the derivation of the final conclusions from the value assessment results	documents the scope and objectives for stability evaluation, stakeholders' stability concerns, and the overall conclusions of the stability evaluation
Information sources	useful for creating an understanding of the architecture as a basis for making judgements and drawing conclusions for an architecture evaluation	useful for creating an understanding of the architecture's behaviour
Assessment Method	describes how information will be gathered and processed, and how value assessment criteria will be applied to the processed information to yield value assessment results	describes how behavioural stability information will be gathered and processed, and how stability assessment criteria will be applied on the processed information to yield stability assessment results
Attribute	some characteristic or property of the architecture	a behavioural property of the architecture's intended behaviour that needs to be considered for stability

Table 8.1 (*cont.*)

Concept	**Architecture Evaluation**	**Stability Evaluation**
Attribute Analysis Objectives	express the value assessment objectives in terms of conditions on attributes of interest	are down-traces of Stability Evaluation Objectives to express them in terms of attributes of interest
Attribute Analysis Criteria	are conditions that must be met by or the tests that must be passed by the measured values of the attributes of interest	are stability conditions that must be met or the tests that must be passed by the system
Attribute Analysis Method	describes the method for analysing one or more attributes of interest to address the attribute analysis objectives	describes the method for analysing one or more stability attributes to address the Stability Attribute Analysis Objectives
Attribute Analysis Results	are used as evidence in a value assessment method to make decisions or draw conclusions as to how well the architecture and/or the system characterised by the architecture addresses concerns	are the outcomes of attribute analysis methods, used as evidence on how well the architecture's behaviour addresses the concerns for stability

8.2.2 Context of Stability Evaluation

Architectural stability evaluation is a judgement about how an architecture is stable in provisioning the intended behaviour, considered in the context of its respective environment. Figure 8.3 depicts the context of architectural stability evaluation in terms of key concepts and their relationships.

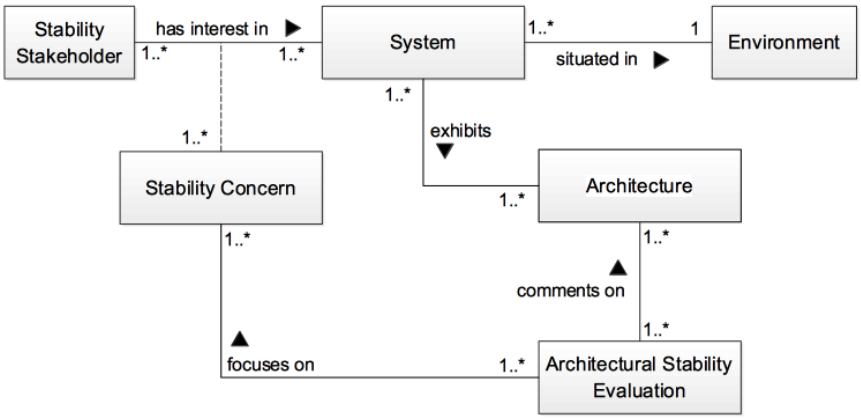

Figure 8.3: Context of Architectural Stability Evaluation

Generally, *stakeholders* of a *system* have interests (i.e. stakes) in that system and its associated *architecture* [507] [506]. We borrow from the ISO/IEC standards [507] [506] the term of "stakeholders" to express the entities that have interest in the architecture's behaviour and related concerns. *Stability Stakeholders* could include: end-users, architects, maintainers. They could also include evaluators and authorities engaged in certifying the system for a variety of purposes, such as the system's conformance to legal provisions, the regulatory compliance of the system and the system's compliance to environmental impact or other relevant policies. Architecture evaluation for stability is highly driven by stakeholders' concerns. Such concerns can cross-cut one or more dimension that could be technical, economic, business, strategic, market, etc. dimensions.

Stakeholders, having interests in a system, ascribe concerns regarding the behaviour of that system and its associated architecture. *Stability Concerns* represent the assurances about the behaviour of the architecture that stakeholders expect to obtain. Examples include stability of the quality of service provisioned, stability in complying with environmental regulations and stability of economic concerns. *Architectural Stability Evaluation* focuses on these concerns.

The architecture of the system of interest is situated and operating in its environment throughout its lifecycle [507] [506]. This environment can contain external entities that interact with or relate to the system and its architecture. The *environment* determines all external influences on the stability of the architecture.

8.2.3 Stability Evaluation

Stability evaluation is meant to examine the behaviour of one or more candidate architectures for the system of interest. Architectural stability evaluation aims to: (i) evaluate the

extent to which the candidate architecture can meet its expected behaviour (ii) validate that the architecture addresses stakeholders' concerns for stability, and (iii) support and inform architectural decision making for design, maintenance and evolution. Figure 8.4 depicts architectural stability evaluation in terms of key concepts and their relations. The *Architectural Stability Evaluation* central component is the mean for determining whether stability concerns of the stakeholders are satisfied.

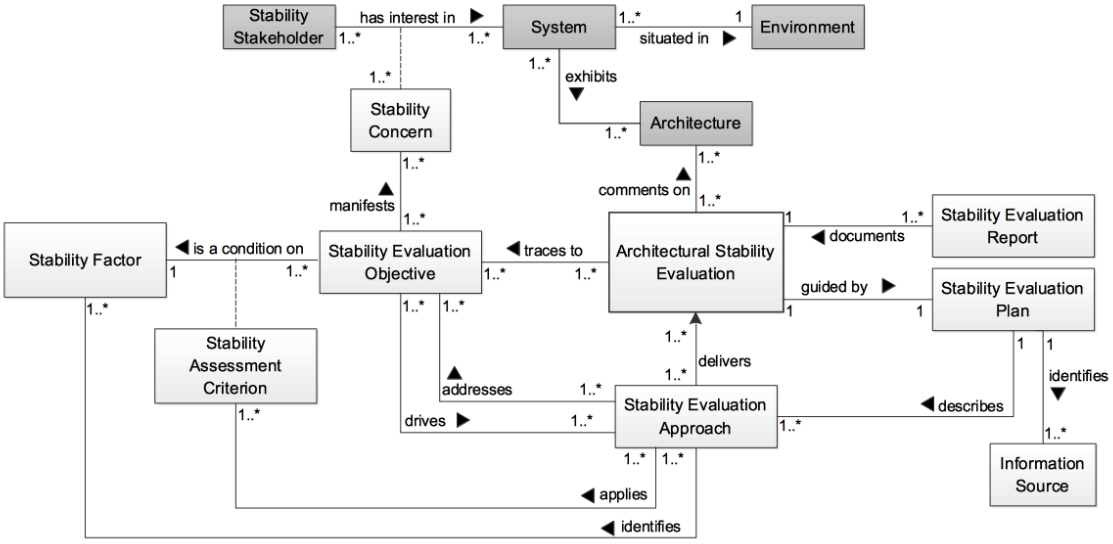

Figure 8.4: Conceptual Model of Architectural Stability Evaluation

From the context, Architectural Stability Evaluation focuses on Stability Concerns (the *focuses on* relation (appearing in Figure 8.3). This relation is further refined by introducing *Stability Evaluation Objectives* (in Figure 8.4). Stability Evaluation Objectives frame one or more of these concerns for evaluation. Stability Evaluation Objectives define how stability concerns will be addressed, i.e. what questions about stability factors will be answered by the *Stability Evaluation Approach*. Heuristics can help to determine the appropriate Stability Evaluation Objectives to express a stability concern or stability advantage associated with the architecture in terms of *Stability Factors*.

The *Stability Evaluation Plan* guides the derivation of evaluation, and the *Stability Evaluation Report* documents it. In more details, the Stability Evaluation Plan documents the context and scope of the evaluation, as well as the stability evaluation objectives that will be addressed to manifest the stakeholders' concerns. The plan also describes the Stability Evaluation Approaches that will be adopted to address these objectives. The specification of stability evaluation approaches for a particular assessment can be influenced by the nature of stability evaluation objectives or relevant architecture viewpoints. Stability objectives help to determine what to be included in or excluded by the evaluation approach. The plan also identifies information sources useful for creating an understanding of the architecture's behaviour as a basis for making judgements and drawing conclusions for the stability evaluation.

A *Stability Evaluation Approach* describes how information will be gathered and processed, as well as how *Stability Assessment Criteria* will be applied on the processed information to yield *Stability Assessment Results*. Stability evaluation approaches vary

between design-time and runtime assessment and may include manual or tool-based techniques, suitable enablers and resources for conducting the stability assessment. They come in various forms, such as quality workshops, expert panels, design walk-through and quality assurance, modelling and simulation, prototype demonstration, system experiment, technical analysis, multi-attribute utility analysis (MAUA), business case analysis, socio-economic analysis, subject matter experts. Additional resources, such as test environments, discrete event simulations or queuing theory models could also be used.

Following the Quality Function Deployment methodology, Stability Assessment Criteria could be based on "Quality Factors" that are derived from end-users desired attributes [509]. Examples of Stability Assessment Criteria could be "within budget constraints" for the affordability/ economic concern, or "consistent all or none behaviour of transactions" for reliability concern.

The Stability Evaluation Objectives and the Stability Evaluation Approaches form the elements of architectural stability evaluation. The latter results in two evaluation work products —Stability Evaluation Plan and Stability Evaluation Report. The overall conclusions of the stability evaluation are expressed in the Stability Evaluation Report along with any supporting findings. The stability evaluation report, if following the ISO standards, could be turned into "Architectural Knowledge" to be reused in the evaluation of other architectures of the same software paradigm or in other contexts [446] [510].

8.2.4 Stability Attributes Analysis

Figure 8.5 depicts the Stability Attribute Analysis in terms of key concepts and their relations. Stability Evaluation Objectives are traced down to *Stability Attribute Analysis Objectives* that are expressed in terms of attributes of interest. The attributes of interest cover attributes of the architecture's intended behaviour that should be kept stable while the architecture is operating. A *Stability Attribute* is a behavioural property that needs to be considered for stability and can be determined quantitatively or qualitatively. Examples include response time, throughput, latency, energy consumption. Each attribute analysis objective should enable the selection of usable measurement protocols for the attribute it describes.

The Stability Evaluation Approach selects *Stability Attribute Analysis Methods* depending on the information needed for assessing stability. Stability Attribute Analysis Methods describe the method for analysing one or more attributes of interest to address the Stability Attribute Analysis Objectives, i.e. they specify the particular ways in which attributes are examined, suitable analysis scale and analysis protocol. Stability Attribute Analysis Methods use manual or tool-based techniques and other suitable tools, such as test environments, to measure and analyse the attributes of interest. They come in various forms, such as performance analysis, behavioural analysis, risk and opportunity analysis, failure, modes, effects and criticality analysis (FMECA) [507] [506]. Stability Analysis methods can borrow the kinds of analysis protocols frequently used for architecture analysis, such as check-list based examination, scenario-based examination, experimentation, model-based examination and benchmarking [507] [506]. Our analysis method (proposed in Chapter 4) could also be used in this context. But instead of using these methods to simply check if the architecture is capable of achieving an attribute, they will be used to

check whether the architecture is constantly achieving it in a stable manner.

The *Stability Attribute Analysis Criteria* are stability conditions that must be met or the tests that must be passed by the system. These conditions could be informed by threshold, benchmarks, expert opinions or Service Level Agreements (SLAs). They are applied as part of the analysis, and their application will yield attribute analysis results as an output.

The *Stability Attribute Analysis Results* are the outcomes of attribute analysis methods. These are used as evidence in a Stability Evaluation Approach to make decisions or draw conclusions on how well the architecture's behaviour addresses the concerns for stability. The analysis can deliver information about attribute values in different scenarios and conditions in which the architecture is evaluated. The results might include determination of a "Measure of Performance" for the architecture attribute. ISO/IEC 15939 [509] can be used as the basis for such measurements. Here it is worth to note that quantitative measurements can sometimes be no more accurate than qualitative measurements due to uncertainty in the measurement devices, methods or tools, or from uncertainty in the contextual conditions where the measurements were taken [507] [506]. For this reason, both quantitative and qualitative measurements should be accompanied by an indicator or estimate of uncertainty in the measurement [507] [506].

The Stability Evaluation Approach specifies the mechanisms through which the Stability Assessment Criteria will be applied on results produced by the attribute analysis methods. The mechanism for using the information produced by attribute analysis can include a specific function for combining the various results, such as a linear weighted sum, harmonic average or other integrative mechanisms [507] [506].

The distinction between Stability Assessment Criterion and Stability Attribute Analysis Criterion is their scope [507] [506]. The Attribute Analysis Criterion depends on architecture attribute(s) and determines a measurement and analysis of those attributes pertinent to Attribute Analysis Objectives in the scope of the Attribute Analysis Method. The Stability Assessment Criterion is used to determine whether the aggregation of the results of the attribute analysis is aligned with the Architecture Evaluation Objectives. The Attribute Analysis Criterion usually can involve far more quantification, whereas the process of aggregating results from the attribute analysis can involve a degree of expert judgement.

8.3 Stability Evaluation in the Software Lifecycle

Architectural stability evaluation can be conducted at one or more stages in the lifecycle of a system and stakeholders are motivated for various reasons. Generally, stakeholders can use stability evaluation to support decisions related to maintenance, evolution, refactoring, replacement and phasing out, further investment etc., throughout the lifecycle of a system. Architects are motivated for stability evaluation to take architectural decisions in the design phase, while maintainers are motivated by taking decisions for maintaining and updating the system while in operation. The characteristics of a particular lifecycle stage can influence the evaluation, e.g. evaluation objectives, method, stakeholders involved. Conversely, the evaluation can inform the iterative and further refinements needed to

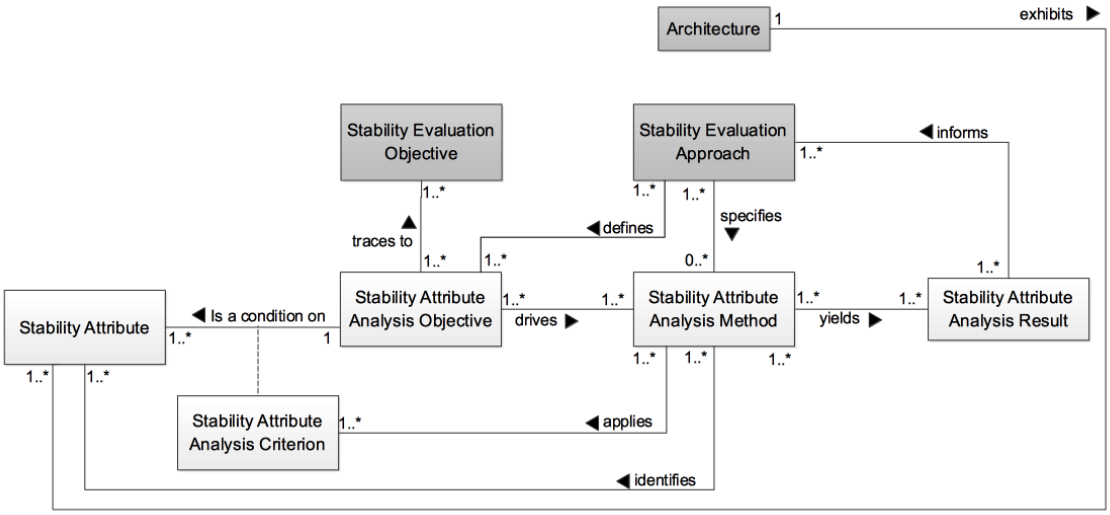

Figure 8.5: Conceptual Model of Stability Attributes Analysis

reach a good enough architecture.

As stakeholders have varying levels of involvement with the system of interest [507] [506], their concerns for the stability of the architecture's behaviour and the lifecycle stages where they need stability evaluations are different. The Stability Evaluation Framework tends to be used at different stages during the software lifecycle for: (i) addressing specific stability objectives, (ii) evaluating certain stakeholders' concerns for the architecture's behaviour, and (iii) addressing specific stability attributes. We focus on the framework components with respect to behavioural stability evaluation during design-time and runtime (in the next sections 8.3.1 and 8.3.2 respectively).

General uses of stability evaluation. The uses of stability evaluation outcomes include: (i) supporting architectural decision making for the system of interest, (ii) determining if the architecture's behaviour addresses stakeholders' concerns for stability, (iii) determining the degree to which the architecture's behaviour meets the end-users' requirements, and (iv) making inferences and decisions about changes or evolution of the architecture. The outcome of the evaluation framework could be turned to constitute the "Architectural Body of Knowledge" to be reused in other contexts for guiding the evaluation of other architectures or benchmarking [446] [510].

8.3.1 Design-time Evaluation

In this context, the evaluation is meant to be conducted at the design stage of the software. This could include evaluating one or more architectures to make architectural decisions. The evaluation can be conducted to justify the choice of the architecture among other candidate alternatives. Here it is worth to note the difference between architecture evaluation and stability evaluation. The former is the "judgment of the value, worth, significance, importance, or quality of architectures" [507] [506] that aims to determine the degree to

which the architecture meets end-user needs, expectations or requirements [507] [506]. The latter focuses on the degree to which the architecture is able to maintain the stability of the expected behaviour. As an example, architecture evaluation can determine that a certain architecture complies to the environmental regulations, while stability evaluation determines if the architecture's behaviour will remain "stable" with respect to energy consumption under varying operational workloads. The evaluation is also concerned with understanding the impact of stability of different attributes on each other. Stability evaluation can help in determining the possible set of adaptation strategies that allow the architecture to keep its stability.

The design-time evaluation aims to capture the stability evaluation objectives that reflect stability concerns, identify evaluation approaches and relevant attributes that characterise the expected architecture's behaviour. A stable architecture is expected to provide constant satisfaction of certain quality and performance levels, as well as sustain other expected behavioural aspects. An architecture that will fail to sustain its expected behaviour will be phased-out.

Context. At the design stage, stakeholders that could have stability concerns include end-users, finance managers and authorities certifying the system's compliance to regulations. Other stakeholders could be identified depending on the context and domain of the system. Examples of their stability concerns include quality of service, economic and environmental concerns.

Stability Evaluation. Stability Evaluation Approaches in design-time could be quality workshops, subject matter expert panels, end-users' workshops, prototype demonstration, technical analysis, multi-attribute utility analysis (MAUA), business case analysis, socio-economic analysis, architects or expertise judgements. Modelling and simulations could also be used, if possible.

Stability Attributes Analysis. Stability Attributes Analysis Methods that could be used in the design-time evaluation are check-list based examination, scenario-based examination, model-based examination and benchmarking. These methods shall specify suitable analysis scale and analysis protocol for analysing stability attributes. For instance, some attributes might be noted as true/false, some use a formula or algorithm to determine the needed information about the attribute. Other attributes might be measured on a fabricated scale from very low to very high with each level in the scale defined to have specific characteristics or using the measurement protocol that has been specified by the attributes analysis method.

8.3.2 Runtime Evaluation

The runtime evaluation is performed while the system is in operation; the objective is to assess the stability of runtime adaptations, maintenance or evolution purposes. In this

context, we focus on keeping the architecture's intended behaviour stable during runtime, considering the attributes critical to be kept stable without violations. Behavioural stability also encompasses the architecture's runtime decisions that would leave the architecture stable in the long-term. Runtime decisions could be either automatic or interactive (e.g. set by the architect).

The runtime evaluation could be conducted either on the system itself or on symbiotic simulations. Symbiotic simulations run close to a physical system, benefiting from real-time measurements from the physical system, and provide feedback to the system [450] [451] [452]. The results of stability evaluation could be used for taking adaptation decisions autonomously during runtime by the adaptation controller (managing system) or to be taken for further offline analysis and decisions.

Context. Stakeholders with concerns for runtime stability could include the system administrators responsible about the operation, maintenance or evolution of the system, added to the stakeholders of the design-time evaluation. Examples of system administrators concerns could be the stability of runtime autonomous decisions and changes required to converge the architecture to a stable state.

Stability Evaluation. Realising runtime behavioural stability requires both continuous provision of quality requirements and stable runtime decisions. Quality requirements include the critical attributes that are required to be kept stable throughout the operation of the architecture without violations. Evaluation approaches in the case of runtime could be automated while the architecture is operating or based on architects and system maintainers judgements. Simulation-based decision support could be also considered [451]. Runtime stability assessment methods could be modelling and simulation, system experiment, technical analysis, multi-attribute utility analysis (MAUA), business case analysis, socio-economic analysis, subject matter experts. Additional resources such as test environments, discrete event simulations, or queuing theory models could also be used. Symbiotic simulation is also a powerful tool to be considered for stability assessment, because of their ability to dynamically incorporate real-time data, providing the system with the effects of decisions on stability made by the simulation [450] [451] [452].

Stability Attributes Analysis. Stability Attributes Analysis Methods that could be used at runtime evaluation are scenario-based examination, experimentation, performance analysis, behavioural analysis, model-based examination and benchmarking. The what-if-analysis, which is concerned with the evaluation of a number of what-if scenarios by means of simulation, would complement symbiotic simulations.

8.4 An Evaluation of Applicability

The proposed framework is applied to evaluate the behavioural stability of self-adaptive cloud architectures case (described in section 4.4.1). The result of the applied conceptual model is depicted in Figure 8.6, and the framework components are presented in the next sections.

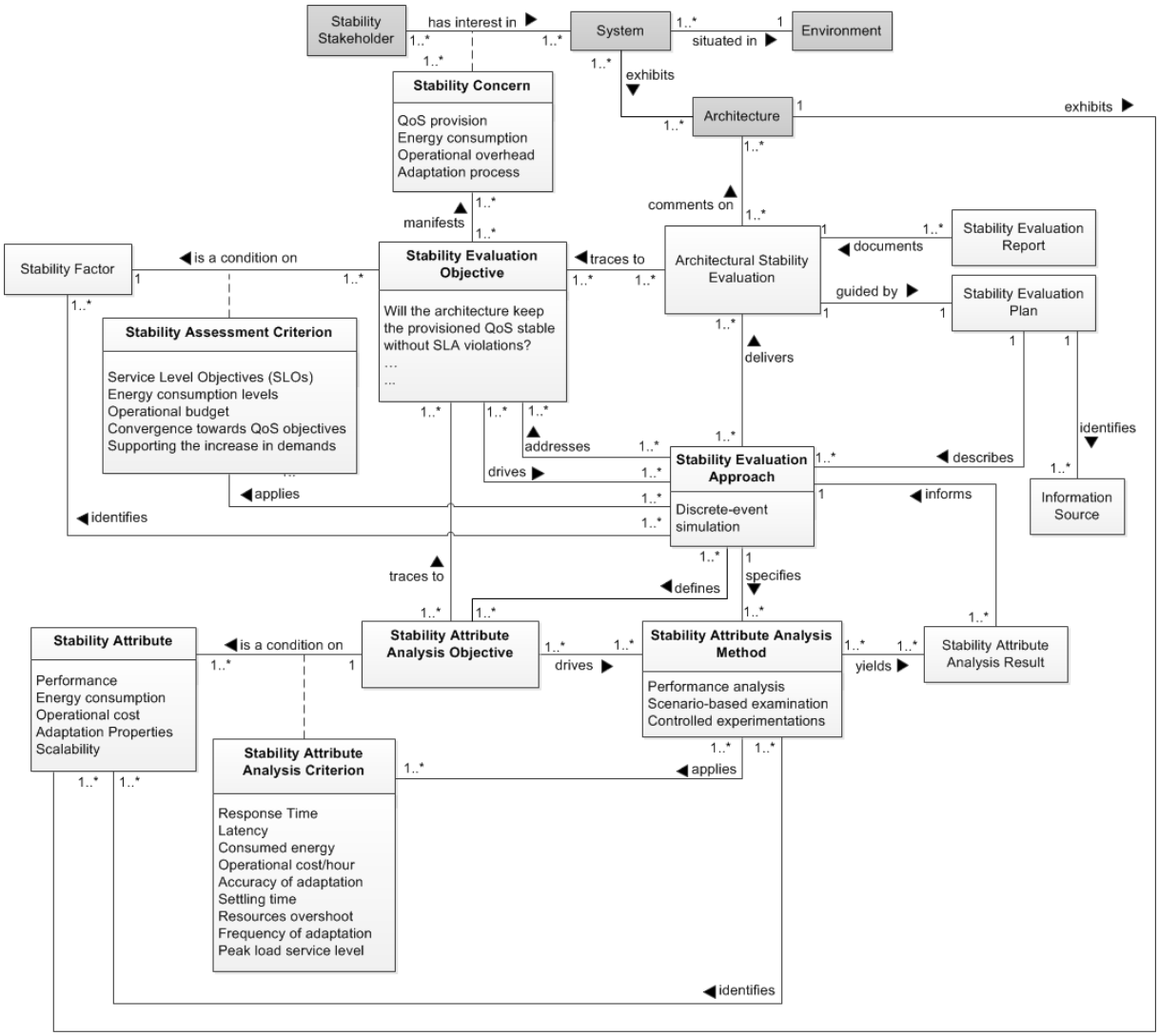

Figure 8.6: Evaluation Case: Application of Stability Evaluation Framework

8.4.1 Context of Stability Evaluation

Self-adaptivity has been motivated as a solution to achieve the level of dynamicity and scalability necessary for these systems, as well as to comply with the changes in components, fluctuations in workloads and environmental conditions during runtime [21] [22] [23]. Self-adaptive architectures are expected to manage themselves following the principles of autonomic computing, to respond to changes in end-users requirements and the

environment coping with uncertainty in runtime operation [24], for continued satisfaction of quality requirements under changing context conditions [25]. In such case, the architecture is required to sustain a stable level of the expected service quality throughout the operation. Meanwhile, the quality of adaptations taking place has impact on the overall behaviour. An adaptation indefinitely repeating the action will risk not reaching the adaptation goals, or even degrading the system's behaviour to unacceptable levels, or probably degrading other stability attributes [25] [26]. Continuous and frequent runtime adaptations might also cause architectural instability leading to performance degradation [25] [26].

In this case, stability is considered as an architectural property concerned with the behaviour of the architecture with respect to QoS provision and the behaviour of the adaptation controller [25]. QoS provision is concerned with ensuring the continuous satisfaction of *architecturally significant quality requirements* [64]. The behaviour of the adaptation controller focuses on observable properties or qualities that are particular to the adaptation process [25] [26].

The context of stability evaluation for the case of self-adaptive cloud architectures is shown in Figure 8.7. Stakeholders having stability concerns include end-users, environment authorities (responsible about the compliance to environmental impact regulations), operational managers and system administrators. End-users concerns for stability are mainly the provision of quality attributes as defined in their SLAs. Here, we focus on attributes critical for end-users which their provision need to be kept stable without violations. Other properties could be stabilised within a given tolerance. These properties vary from one system to another. For instance, responsiveness in a real-time system is a critical attribute, while throughput in a data analytics system is the critical one. The environmental impact concerns are mainly related to stabilising energy consumption and CO_2 emission during operation [157]. Operational managers are concerned about the operational overhead, i.e. ensuring that operational costs are within the identified budget (to ensure a minimum profit) without encountering penalties due to SLAs violations. Concerns of system administrators are related to the stability of the adaptation process, that is the degree in which the adaptation process will converge toward the adaptation goals [25] citeVillegas2017.

8.4.2 Stability Evaluation

Behavioural stability evaluation is about assessing the architecture stability state of fulfilling the runtime quality requirements during operation and assessing the stability of the adaptation process [25]. Such evaluation can help in identifying adaptation actions when necessary to fulfil the changing workload, ensuring the adaptation actions will leave the architecture stable in the long term and avoiding unnecessary adaptations. Table 8.2 summarises how the stakeholders' concerns for stability are framed into evaluation objectives. These objectives indicate qualities representing the architecture's expected behaviour. The qualities are tangibly expressed with the help of one or more attributes subject to stability. By analysing the stability of these attributes and applying relevant analysis criteria, it is possible to address the stability objectives.

In more details, to manifest the previously identified stability concerns, evaluation

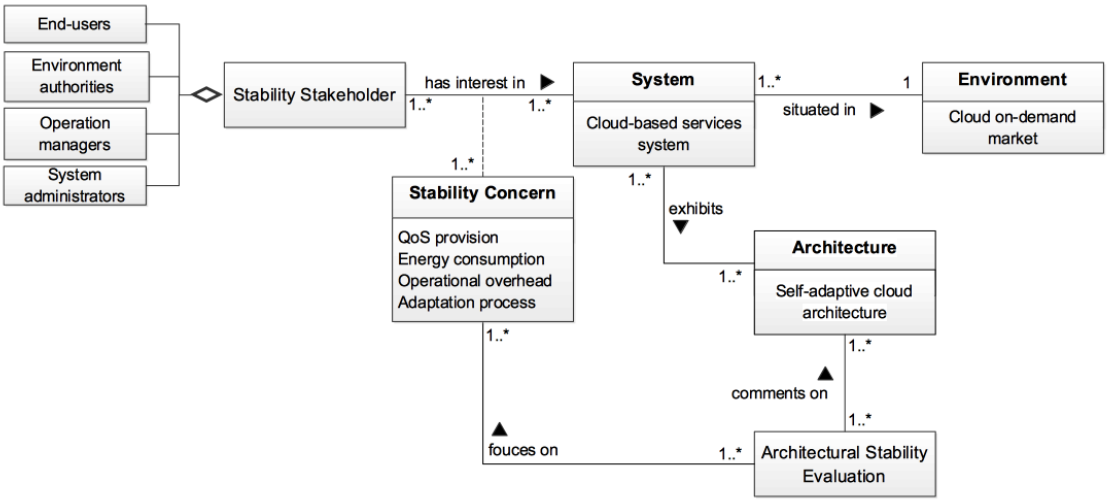

Figure 8.7: Evaluation Case: Context of Stability Evaluation

objectives focus on:

(i) the capability of the architecture to keep the QoS provision stable without SLA violations,

(ii) the capability of the architecture to retain the environmental impact stable according to the regulations,

(iii) the capability of the architecture to keep operational overhead stable with the varying workload and consequent adaptations, and

(iv) the capability of the architecture to perform adaptations that converge toward adaptation goals without indefinitely repeating adaptations or performing unnecessary adaptations.

By these evaluation objectives, the first three objectives could be assessed with the following criteria: (i) Service Level Objectives (SLOs) defined in SLAs for the attributes subject to stability, (ii) energy consumption levels defined in environmental regulations, and (iii) the operational budget. Stability of the adaptation process can be assessed by the degree to which the adaptation process converges towards quality of service (adaptation goals) [25] without unnecessary adaptations that might cause architectural instability. Stability can also be assessed by the degree of supporting the increase in demands [25].

In the case of runtime evaluation, stability evaluation methods could be either offline or online. In the case of offline evaluations, we can rely on the architect's expertise. In the case of online evaluations, evaluations could be automated in the architecture managing component, i.e. adaptation controller. In both cases, tool-based methods are needed for stability assessment. Discrete-event simulations could be a feasible tool for architects to collect assessment results and to make decisions or draw conclusions as to how well the architecture is stable. In case of automated evaluations, the core of the simulation could be integrated as part of the architecture managing component.

Table 8.2: Evaluation Case: Breakdown of Stability Concerns into Measurable Stability Attributes

Stability Concerns	**Stability Evaluation Objectives**	**Stability Assessment Criteria**	**Stability Attributes**	**Stability Attribute Analysis Criteria**
Stability of QoS provision	Will the architecture keep the provisioned QoS stable without SLA violations?	Service Levels Objectives (SLOs) defined in SLAs	Performance	Response Time < 15 ms, Throughput $< x$ req./sec, Latency < 1 sec
Stability of environmental impact	Will the architecture retain the environmental impact stable according to the regulations?	energy consumption levels defined in regulations	Energy consumption	Consumed energy $< x$ kWh
Stability of operational overhead	Will the architecture retain the operational overhead stable with the varying workload and consequent adaptations?	within operational budget	Operational cost (cost of CPU, memory, storage, bandwidth)	Operational cost/hour within a certain limit
Stability of adaptation process	Will the architecture adaptations converge toward adaptation goals without indefinitely repeating controlling actions?	Degree to which the adaptation process converges towards adaptation goals with finite discrete controlling actions	Quality of Adaptation	Accuracy of adaptation, Settling time, Resources overshoot, Frequency of adaptation
	Will the architecture adaptation be able to support increasing demands with sustained performance?	Support of increasing demands within expected peak loads	Scalability	All requests serviced with a sustained performance at peak load, 90% requests serviced with a sustained performance at unexpected peak load

8.4.3 Stability Attributes Analysis

The above stability evaluation objectives and criteria are traced down and analysed to the following stability attributes of interest: (i) performance (considering performance attributes critical to the end-user to be subject to stability), (ii) energy consumption (measured by kWh), (iii) operational cost (calculated by the monetary cost of CPU, memory, storage and bandwidth), and (iv) quality of adaptation that affect stability.

Adaptations properties include: accuracy of adaptation, settling time, resources overshoot and frequency of adaptations [25]. The accuracy of adaptation is measured in terms of how close adaptation goals are met within given tolerances [25], or stability objectives in our case. Higher accurate adaptations would leave the architecture in a stable state and eliminates frequent adaptations that might cause architectural instability. Settling time is the time required by the adaptation system to achieve the adaptation goal, reflecting how fast the architecture adapts and reaches adaptation goals [25]. Long settling time can leave the architecture in unstable states, as a result of slow adaptations or reaching goals. Resources overshoot expresses the number of resources used in excess by the adaptation process to achieve the adaptation goals in a required settling time [25]. Managing the utilisation of computational resources is important to avoid reaching instability states. The frequency of adaptations reflects that the architecture produces finite discrete controlling actions for adaptations, i.e. not indefinitely repeating adaptations. As adaptations are motivated by continuous provision of quality requirements (adaptation goals) [25], the continuous runtime adaptations to meet these requirements might lead to architectural instability, due to the high *frequency of adaptations* or unnecessary continuous adaptations [22] [21] [23] [25]. Given the latency of adaptations —"the time it takes for an adaptation to cause its intended effect" — [29], more adaptations could be performed unnecessarily, leading to an unstable state. A stable architecture would perform less frequent adaptations and eliminates unnecessary adaptations that might cause instability. Scalability of adaptations reflects the capability of the architecture to adapt for servicing end-users during peak loads.

Methods to be adopted for Stability Attributes Analysis could be performance analysis and scenario-based examination. We identify *change scenarios*, that are the most relevant (sequences of) system or environmental changes that might affect stability. A scenario is "a postulated sequence of events that captures the state of the system, its environment, and its goals during a given time frame, as well as changes affecting all the aforementioned elements" [29]. It is defined in terms of state (system and environment), changes applied to that state, and system goals. The change scenarios are employed by stability analysis method (simulation in our case) for analysing stability attributes.

8.5 Experimental Evaluation of Runtime Stability

The main objective of the experimental evaluation is to examine the behaviour of the architecture when considering stability evaluation during runtime while making adaptation decisions, in comparison with a foundational self-adaptive architecture (described in section 5.4.2.1).

8.5.1 Developed Evaluation Tools

To conduct stability runtime evaluation, we developed (i) a symbiotic simulator for the self-adaptive cloud architectures as an example of stability runtime evaluation tools, and (ii) a dynamic model for evaluating stability during runtime.

The symbiotic simulation environment was built using the widely adopted *CloudSim* simulation platform for self-adaptive and self-aware cloud architectures [5]. The proposed environment extends *CloudSim* with novel extensions useful for modelling and testing self-adaptivity and self-awareness. The toolkit allows running dynamic runtime workload and can be used as a symbiotic simulator during runtime. Details of the symbiotic simulator and its validation appear in Appendix D. The concept of using symbiotic simulations during runtime is illustrated in Figure 8.8.

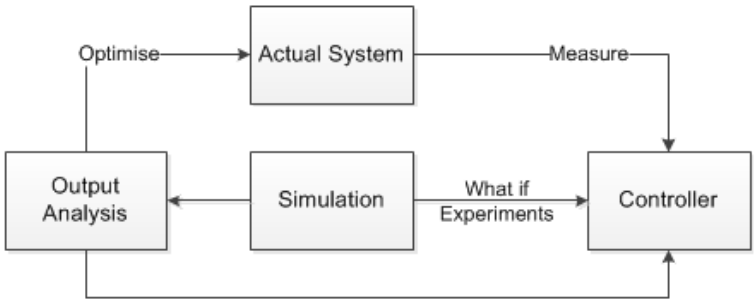

Figure 8.8: Using Symbiotic Simulation for Runtime Evaluation

We also developed a queuing theory-based model for evaluating runtime stability. The dynamic modelling of tactics impact on stability is based on Markov analytical model and queueing theory. The premise is that the model can enable the analysis and evaluation of the extent to which candidate tactics can meet stability goals and keep the architecture in stable behaviour. The model is detailed in Appendix E.

8.5.2 Experiments Setup

We used the developed simulation environment in conducting our experimentations. The benchmarks and testbed configuration used are as described in section 5.5.1.1 and 5.4.2.2 respectively. The initial deployment of the experiments is: 20 hosts running 20 m4.2xlarge VMs. In order to closely observe the adaptation properties, we set one stability attribute, which is response time (25 ms). In these experiments, the architecture is configured once with the foundational self-adaptation controller, and once with the stability-evaluation controller. The latter implements the proposed stability evaluation framework.

8.5.3 Results of Stability Attributes

Considering the experiments total results, we report the average response time of 30 runs in Figure 8.9. Generally, the average response time of all requests for each service type

is better achieved by the stability-evaluated architecture due to proactive adaptations, compared to self-adaptive architecture.

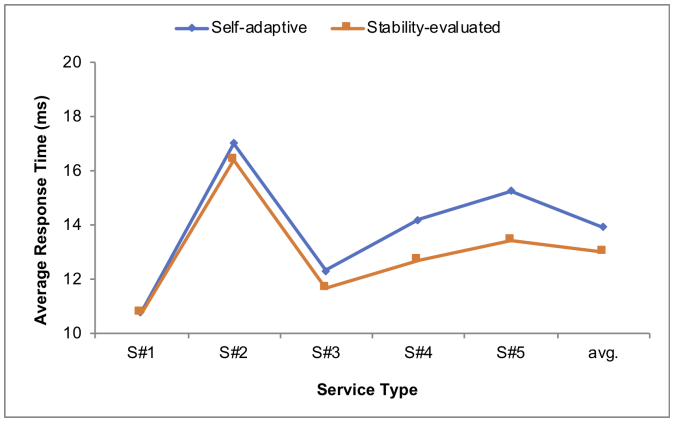

Figure 8.9: Average Results of Response Time

8.5.4 Results of Adaptation Properties

The accuracy of adaptation is shown, here, by the percentage of requests completed without violation in response time, and settling time is shown by the total time periods where the response time was violated until the adaptation actions became effective. As shown in Figure 8.10, the accuracy of adaptation is highly achieved in the stability-evaluated case for service type 1 and is generally better for all other service types. Regarding the settling time shown in Figure 8.11, the stability-evaluated case was capable of keeping it much less than the self-adaptive architecture. This reflects the quality of adaptation decision in achieving its objectives and avoiding possible violations.

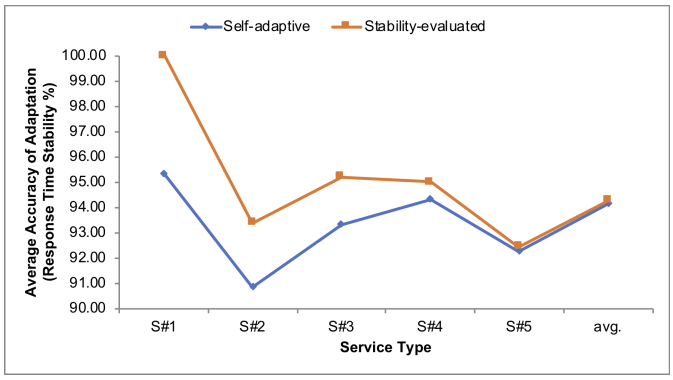

Figure 8.10: Average Results of the Accuracy of Adaptation

Resources overshoot is expressed by the total number of hosts and VMs utilised by the adaptation process. As shown in Figure 8.12, the stability-evaluated controller was capable of considering the resources consumed beside achieving the stability attribute. This is due to the pro-active adaptations that are capable of keeping a longer effect and avoid unnecessary adaptations. Resources overshoot implicitly reflects the less energy

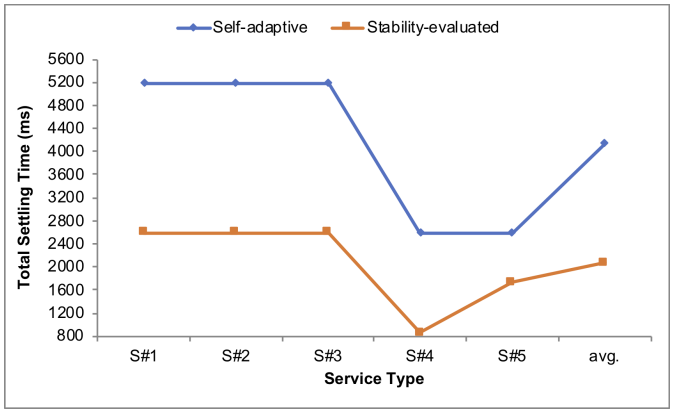

Figure 8.11: Average Results of the Adaptation Settling Time

consumption and operational costs, as they directly related to the number of hosts and VMs respectively.

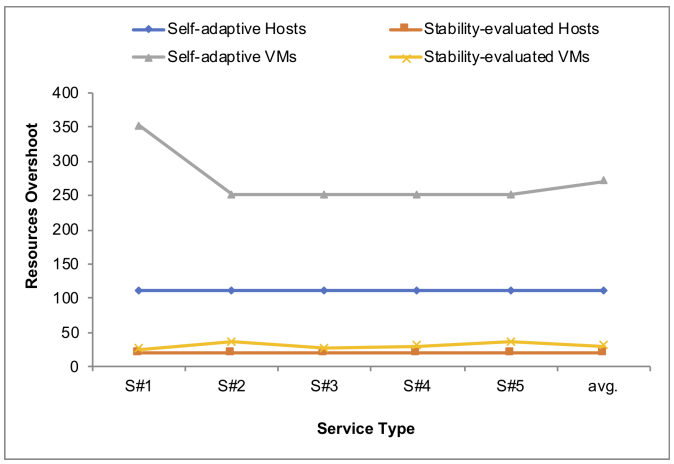

Figure 8.12: Average Results of Resources Overshoot

Regarding the frequency of adaptation, it is shown here in terms of the number of adaptation cycles. As shown in Figure 8.13, the self-adaptive architecture was performing adaptation cycles constantly for all service, as per the workload fluctuation. Meanwhile, the frequency of adaptation is found much less in the stability-evaluated adaptations, and sensible to the size of the requests. This is due to the evaluation that is taking into consideration the long-term effect of adaptation on stability. Evidently, adaptation overhead is directly proportional to the frequency of adaptation (evaluated by the total time spent by the architecture in the adaptation process). This is shown in Figure 8.14, where the overhead in case of stability evaluation, nearly following the same pattern of the adaptation frequency, is less than the overhead of self-adaptive.

Generally, evaluating stability when taking adaptation decisions has successfully realised stability attributes (performance) without violations, and has taken into consideration quality of adaptation properties, resulting in better settling time and fewer adaptations frequency and overhead.

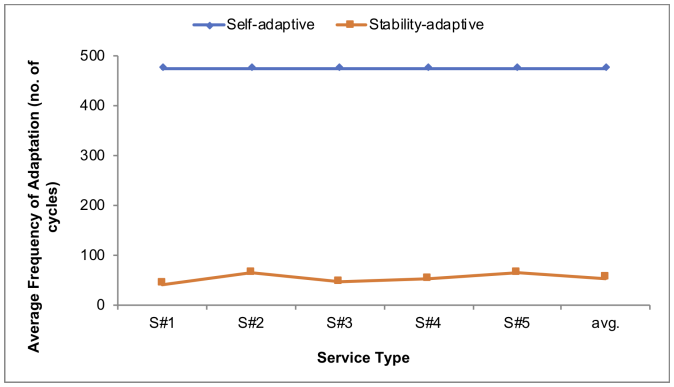

Figure 8.13: Average Results of the Frequency of Adaptation

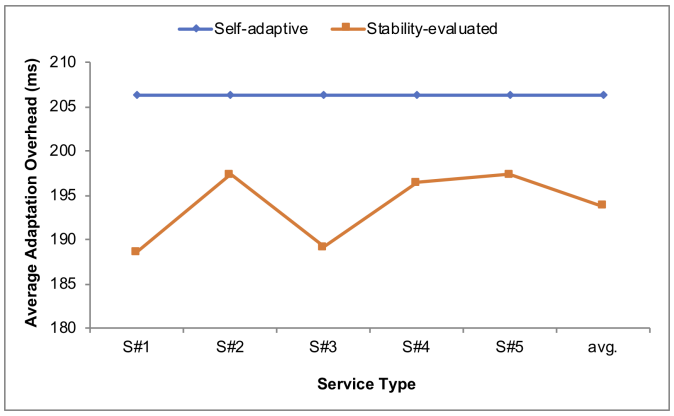

Figure 8.14: Average Results of Adaptation Overhead

8.5.5 Discussion

The evaluation case has illustrated the importance of evaluating the stability of critical qualities that are needed to be kept stable without violations, where our framework is found useful for understanding, addressing and analysing such qualities, avoiding dangerous pitfalls like instability states and indefinitely repeated adaptations. The framework helped in explicitly capturing stability concerns that might be ignored in classical architecture evaluation or might be considered without considering their stability. In our approach, various stakeholders are taken into consideration, explicitly the ones related to the operation of the system, e.g. system administrators. Using the stability evaluation in the self-adaptive cloud architecture case has revealed stability of the adaptation process as an explicit concern that should be taken into consideration. This concern has been widely ignored in the self-adaptivity literature.

The case study has also illustrated how our stability evaluation approach has successfully linked the stakeholders' concerns for stability to software qualities and attributes that the architecture exhibits. Analysing and assessing stability attributes allows making operational decisions that would leave the architecture in a stable state in the long-term. The framework enables software practitioners to consider trade-offs across qualities that are critical to being kept stable. Generally, our systematic evaluation approach has added-value to the architecture evaluation. That is by making more-informed architectural and operational decisions for the well-being of the architecture in the long-term. While the in-

stantiated case has served the illustrative purpose, the framework could be further refined after application to practical cases.

On the other side, the symbiotic simulator of self-adaptive cloud architectures illustrates how the stability evaluation framework can be applied during runtime operation of large, complex systems that operate in an uncertain continuously changing environment. Integrating our stability evaluation framework into the operation of cloud architectures provides valuable support to managers and system engineers trying to maintain technical, economic, and environmental requirements on the long-run. These decisions could be either taken autonomously by the adaptation controller or the system administrators. Such evaluation also gives insights for possible maintenance, changes and upgrades.

8.6 Related Work

Evaluating stability at the design phase aims at measuring to which extent a particular architecture design is capable of accommodating future changes while remaining intact [39], i.e. structural aspect for evolution purpose. 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 [124]. Approaches of stability evaluation could be categorised as: (i) retrospective, and (ii) predictive [117] [121] [511]. Retrospective approaches aim at analysing how easily the evolution occurred [39]. Predictive approaches aim at predicting how the evolution will take place, by examining how the architecture will endure the likely changes [39].

An early survey of design-time evaluation approaches [121] 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) [330], Attribute-Based Architectural Styles (ABAS) [331], Scenario-Based Architecture Reengineering [332], Quality-Attribute-Based Economic Valuation [333], and CHARMY for verifying architectural specifications [334]. These methods focused on evaluating architectural decisions in relevance to traditional quality attributes [39]. 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]. The "ArchOptions" approach, based on Real Options Theory and taking the economic perspective in evaluation, explicitly studied evaluating architectures' stability for evolutionary purpose [120] [121].

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. Some representative work that partially tackled aspects related to architectural stability include the work of Ghosh et al. [339] that 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 [143], the impact of environmental changes on resilience was quantitatively evaluated

[1]Further details about the critical relation between stability and these evaluation methods could be found in [121] [39].

using Exploratory Data Analysis (EDA). The works of [340] [341] 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" [168], the resilience benchmarking presented in [142] has addressed the robustness and resilience issues. The survey presented in [148] identified the challenges and opportunities for provisioning dependable and resilient cloud-based software services.

Tough 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. Yet, the behavioural aspect of stability was not addressed in the design and runtime phases.

8.7 Summary

In this chapter, we presented a framework for evaluating architecture behavioural stability, based on the ISO/IEC Architecture Evaluation standards. The main components of the framework are: context of evaluation, stability evaluation and stability attributes analysis. The framework explicitly addresses the process of planning, execution and documentation of behavioural stability evaluations. The framework has been applied to the self-adaptive cloud architectures case. One feature of the framework is making explicit consideration of stakeholders' concerns for stability and environment factors. The systematic evaluation approach has added-value to architectures evaluation practices. That is by making better informed architectural and operational decisions for the well-being of the architecture in the long-term. The design-time evaluation has shown that our approach has revealed stability and quality of adaptation as explicit concerns. When considering stability in runtime adaptations, the experimental evaluation has proven enhancements in achieving stability and quality of adaptation.

Architectural stability evaluation is particularly important for long-living software systems. Evaluating stability is beneficial for many reasons, including: (i) determining if the system is architected in a way that it keeps its behaviour stable, (ii) evaluating the architecture's effectiveness and suitability in keeping its intended behaviour stable and maintaining stability over time while the system is in operation and as it evolves, and (iii) identifying risks which can threaten the structural and/or behavioural stability of the system, and (iv) identifying preventive measures for maintaining stability. Such evaluations allow to keep the architecture's intended behaviour and maintain long-living architecture without phasing-out. Architectural stability evaluation can also be an instrument of architecture governance [507] [506] to conduct recurring evaluations in a systematic manner at different stages of the software lifecycle, e.g. architectural design decision, runtime operation, continuous maintenance and evolution.

Chapter 9

Conclusions and Future Directions

I may not have gone where I intended to go, but
I think I have ended up where I needed to be.
— Douglas Adams

9.1 Summary and Discussion

This thesis tackles the problem of characterising and engineering the notion of stability in software engineering. One of the main motivations of our work is the strategic importance of stability for software longevity.

To characterise the notion of stability in software engineering (*RQ1*), we reviewed the state-of-the-art related to stability as a software property, with a special focus on software architectures. Having found that stability has been interpreted in various ways, we proposed a taxonomy for characterising the concept. 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. This survey serves as a primary investigation for deeply characterising architectural stability, to take it further towards handling the wider concept and the related challenges.

As the review indicates the need to shift from a narrow concept of stability (architecture intactness), we discussed a multi-dimensional perspective for characterising stability as a software property (*RQ1*). The multi-dimensional perspective contributes to advancing the state-of-the-art and improving the state-of-the-practice of stability. This perspective contributes also to determine the primitives and requirements for realising and engineering stability as a software property (*RQ2*). Focusing on the behavioural aspect of stability, we have drawn conceptual design principles inspired by Control Theory to capture the intended behaviour (*RQ2*).

For analysing and modelling stability (*RQ3*), the thesis performed stability analysis inspired by the ISO/IEC/IEEE 42010 Architecture description standards. The stability analysis introduced a qualitative model for representing the knowledge of attributes subject to stability synthesised from multiple stakeholders concerns and architectural view-

points. In modelling stability, we employed probabilistic relational models that capture the correlations between stability attributes of different viewpoints. Bayesian networks are, then, used for quantitatively calculating probability distributions of the impact of stabilising specific attributes on interdependent attributes, as well as reasoning about stability under runtime uncertainty. The approach can effectively conduct runtime inference to reason about stability attributes given the continuous runtime uncertain changes. Such reasoning improves the achievement of the intended behaviour and supports seamless operation.

Regarding the engineering practices to support runtime behavioural stability for self-adaptive architectures (*RQ4*), the thesis presented a reference architecture and goals modelling for stability, as well as online reasoning mechanisms. In this context, self-awareness computing has been found as a potential mechanism for dealing with stability attributes on the long-run and their associated trade-offs.

Therefore, the reference architecture leverages on self-awareness and self-expression primitives. The architecture also embeds the *SAwGoals@run.time* component for modelling stability runtime goals, in order to enable efficient use of self-awareness and self-expression in achieving stability goals. The proposed design-support artefacts would assist architects and practitioners in planning for stability, as well as designing stable and long-living systems. Such design principles would increase the efficiency of the architecture runtime operation, delaying the architecture drifting and phasing-out as a consequence of the continuous unsuccessful provision of quality requirements.

Further, we extended the architecture to dynamically reason about stability, where we implemented different computational intelligence techniques in self-awareness components. The goal-awareness was capable of managing stability goals, by realising the symbiotic relation between the runtime goals model and self-awareness. The online learning technique in time-awareness has efficiently provided insights for stability on the long-run. Stochastic games have managed trade-offs between stability attributes, allowing the architecture to take adaptation decisions for better tuning, responding and achieving stability goals.

Given the particular importance of architectures evaluation and the lack of behavioural stability evaluation practices, the thesis proposed a framework for evaluating architecture behavioural stability, based on the ISO/IEC Architecture Evaluation standards (*RQ4*). The framework explicitly addresses the process of planning, execution and documentation of behavioural stability evaluations. The use of this framework shall enable more effective architecture evaluations and enhance the body of knowledge on architecture evaluation. Such evaluation approach could be considered as a guide for architectural stability evaluation.

In terms of evaluation, we used the self-adaptive cloud architectures case throughout the thesis for its runtime dynamics. We have demonstrated the applicability of the proposed analysis methodology, reference architecture and stability evaluation approach. The qualitative evaluations have shown the added values of our approaches, where architectural and operational decisions are better informed for the long-term well-being of the architecture. Our approaches have revealed stability and quality of adaptation as explicit concerns, where the latter properties have been widely ignored in the literature. We have also shown the feasibility of the probabilistic modelling and computational intelligence techniques for reasoning about stability. The quantitative experimentations have proven

enhancements in achieving stability and quality of adaptation on the long-run.

9.2 Threats to Validity

In this section, we discuss potential threats to validity common to this research. The Threats are identified according to the classification of [512] and [513]. We describe these threats below.

1. **Threats to Validity Related to the Literature Review.** Though we followed the guidelines of secondary studies [92] [93] when conducting our literature review on stability (in Chapter 2), trade-offs management (in Chapter C) and self-awareness (in Appendix B), common threats to validity are:

   - **Construct validity.** This relates to sources investigated and data collection, including:
     - *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 [93] [514] and most relevant electronic databases to computer science and software engineering [515]. 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 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. Tough it was necessary to fulfil the targeted schedule, data extracted by the main researcher was validated by the thesis supervisor, which lead us to believe that the effect of this error is minimal.

- **Internal validity.** This is concerned with the methods used in conducting the above-mentioned surveys and related conclusions. The following could threaten internal validity:
  - *Scope of the review.* The questions defined might not have covered the whole research area. More specifically, the third question focused on the engineering practices for the architecture. This implies that relevant information may not be found in the review if one is concerned with other artefacts, such as software design. For this issue, we had several discussions to refine the 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 a set of 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 2.8), we presented seminal and/or representative work 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 do not claim completeness in this part. But the review is mainly based on the concepts and questions defined earlier. Tough we argue that the search strategy (defined in Appendix A) 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) [98] [21] [99] [100]. 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.** This is concerned with the generalisation and applicability of the study findings, including:
  - *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.** This is concerned 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.

2. **Threats to Validity Related to Proposed Approaches.** Though we proposed systematic approaches for engineering stability (in Chapter 3, 4, 5, 6 and 8), potential threats to validity could be:

   - The dependency on the human capabilities in the analysis step of the proposed method (in Chapter 4) would form a threat to validity on the end results when using the proposed approach. This might be due to the lack of information or expert knowledge. Yet, our approach could be complemented with formal methods of causality discovery [516] [517], structuring causal trees [443] and learning structure from data [444]. Similarly, depending on the human in selecting the architecture pattern (in Chapter 6) would form a threat to validity on the end results. This might be due to the lack of information or expert knowledge. Yet, our approach could be complemented with symbiotic simulations for testing the architecture design [451] [452].
   - With respect to the generalisability of the proposed work, we believe the method provides systematic guidance to architects and practitioners. Yet, customisation might be needed if the adaptation controller of the self-adaptive system has different components. As the application of the method tends to be subject of the system under consideration, applicability and generalisability of the method to different software domains can uncover new modalities, customisation, simplification or extension to the method.

3. **Threats to Validity Related to Evaluation.** The potential threats to validity to our evaluation case and experiments (in Chapter 4, 5, 6, 7 and 8) are:

   - Subjectivity might be considered a threat to validity in setting the stability attributes. This was conducted based on the author's background and knowledge, i.e. we have chosen the stability goals thresholds purely based on our observations. Our mitigation strategy for this issue is to base the evaluation case on others previous work [3], this makes us believe that the case study is practical and reflects the nature of cloud-based software systems. Also, these goals have proved to be challenging when running the experiments.

- Another threat to the validity of our evaluation lies in the fact that the approach was evaluated using one case. Yet, the dynamics presented in cloud architectures is an appropriate case study representing the dynamics of modern software systems, and we plan to conduct other cases in industrial contexts and different business segments.
- Experiments were conducted in a controlled environment and have not considered the real-life scenario of switching between different service patterns and changing stability goals during runtime for different end-users. Given the use of a real-world workload trend and the *RUBiS* benchmark, we consider that our experiments have given good enough indication and approximation of likely scenarios in a practical setting.
- The fact that the proposed work is evaluated by its author presents a threat to objectivity. Yet, the evaluation case has served as an illustration for the potential value of this research. This could be further evaluated in industrial contexts by independent practitioners without bias

9.3 Future Directions

There are many possible directions in which the work of this thesis could be further developed. These fall into a number of distinct areas, as follows.

- *Stability Modelling.* As seen in Chapter 5, stability modelling was based on Bayesian networks. It would be beneficial if online learning techniques are employed to update prior knowledge and obtain posterior stability probabilities. We are also interested in modelling temporal (dynamic) relationships among stability attributes, i.e. representing how the value of an attribute may be related to its value and the values of other attributes at previous points in time. In the same vein, learning the structure of parameters when building stability Bayesian network would advance our methodological analysis.
- *Reasoning about Stability.* Our reasoning about stability could be augmented with other computational intelligence techniques to support self-awareness capabilities and engineer stability-aware adaptations. A number of directions could be investigated:
  - Different online learning techniques could be investigated to compare their efficiency in given scenarios.
  - Our development of trade-offs management could be further extended by defining heterogeneous properties to consider real-world scenarios of PMs and VMs with different capacities.
  - Another possible improvement for reasoning about stability using self-awareness is switching between different self-awareness techniques during runtime. This would achieve better results by using the technique with the highest benefit in each given scenario.

  - It would also be interesting to extend the proposed work for interaction-awareness in relevant environments, such as cloud federations and geo-distributed clouds. Interaction-awareness techniques would support the stability of multiple cloud nodes when interacting with each other.

- *Stability evaluation.* It would be beneficial to extend the approach for evaluating architectural stability (proposed in Chapter 8) to reconcile different views in order to get a comprehensive evaluation of architectures. As the proposed approach could be considered as a guide for architectural stability evaluation, it could be further extended by the 4+1 View Model [325] [518].

- *Domain-specific extensions.* The proposed work could be further extended in a domain-specific way. Possible extensions include:

  - This work could be further tailored to be aligned for certain types of architectures, such as enterprise architectures and System-of-Systems architectures. The type and complexity of the architecture would reveal new results when analysing architectural stability.

  - This research, which is concerned with software architectures, could be taken further towards alignment of software architectures with the deployment domain (such as grid, volunteer, mobile computing), where other factors could affect stability.

- *Practical setting.* The practicality of this work could be investigated in an industrial setting. Potential future work is:

  - The state-of-the-art survey on stability in software engineering (presented in Chapter 2) would be complemented with a review in an industrial setting to cover related state-of-practice. An industrial survey would be beneficial to provide an overview of the industry needs for researching about stability and developing techniques that will better match the demands from industry.

  - The practical verification of our work is another potential. For characterising the notion of stability, it would be interesting to investigate the maturity of our taxonomy, characterisation and working definitions of stability (proposed in Chapter 3). From our point of view, a working workshop for researchers, practitioners and educators would help in advancing this matter and creating a widely-adopted set of concepts in the software engineering community. Our stability analysis methodology (Chapter 4), proposed reference architecture (Chapter 6) and stability evaluation approach (Chapter 8) could also benefit from verification by other researchers and academics, as well as practitioners in industrial settings.

  - We also plan to practically validate the proposed work by implementing its elements for cloud infrastructure-as-a-service (IaaS) management software systems, such as OpenStack [519].

- *Evaluation and Application.* On the evaluation side, different case studies and applications could give extra insights for considering stability. For instance:

– Conducting experimental evaluation using other benchmarks would be useful in finding possible strengths and weaknesses of the proposed work. Example include the Wikipedia workload [520], workload of e-commerce sites [521], static Web sites [522], content delivery websites [523] and multimedia delivery systems [524] [525]. While being another realistic workload, such benchmarks would stress the architecture by different functionalities, requests types and fluctuations.
– We would also like to exploit other evaluation case studies in different research domains, such as scientific computing [526]. In this domain, Scientific Workflows have been widely used by the scientific community to model large-scale scientific problems in areas, such as bioinformatics, astronomy, and physics, where they are used to analyse and process large amounts of data efficiently [526] [527]. As workflows are commonly interconnected via data or computing dependencies, they require a distributed platform in order to be executed in a reasonable amount of time, and they are often data- and resource-intensive applications [527]. Their processing is mainly related to the orchestration of the tasks on the distributed computing resources and is guided by a collection of QoS requirements defined by the application users (such as total execution time) or meeting a specified budget or deadline. Evidently, this case has different stability dimensions and attributes.

9.4 Closing Remarks

In conclusion, this thesis has contributed to characterising and engineering the notion of architectural behavioural stability. The thesis has introduced a new multi-dimensional way for engineering stability as a long-term software property. Focusing on the architectural level, we proposed novel techniques and extensions for analysing, modelling, engineering and evaluating runtime stability. The proposed work would assist architects and practitioners in explicitly addressing stability as a software property. The conducted experiments have shown evidence on the effectiveness of the proposed work in dealing with runtime dynamics and uncertainty on the long-run. The findings of this thesis can provide a better understanding of stability property and the requirements for engineering long-living software systems.

To this end, the thesis provides systematic support for characterising and reasoning about the stability of software architectures during runtime under uncertainty. The benefits of this research are: (i) continuously meeting behavioural requirements, (ii) assisting in software adaptation under uncertainty, (iii) engineering for long-lived architectures, and (iv) minimising ramifications of architectural erosion. Particularly, the runtime yield is enhancing the adaptation process, by informing it during runtime for the choice of adaptation strategies that keep the architecture stable in the long-term, while the long-term yield is having long-lived software, which keeps the intended behaviour.

Appendix A

Survey on Stability in Software Engineering: Review Protocol and Analysis Results

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 [92] [93]. 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 questions: (i) how stability could be defined and characterised as a software property? what is the current state of research on software stability? and (iii) which engineering practices have been developed by the research community for realising and evaluating architectural stability?

In the first question, we identify the definitions of stability proposed in the literature, with the goal of getting a sound definition and characterisation of this quality property. The second question 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 the third question, 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.

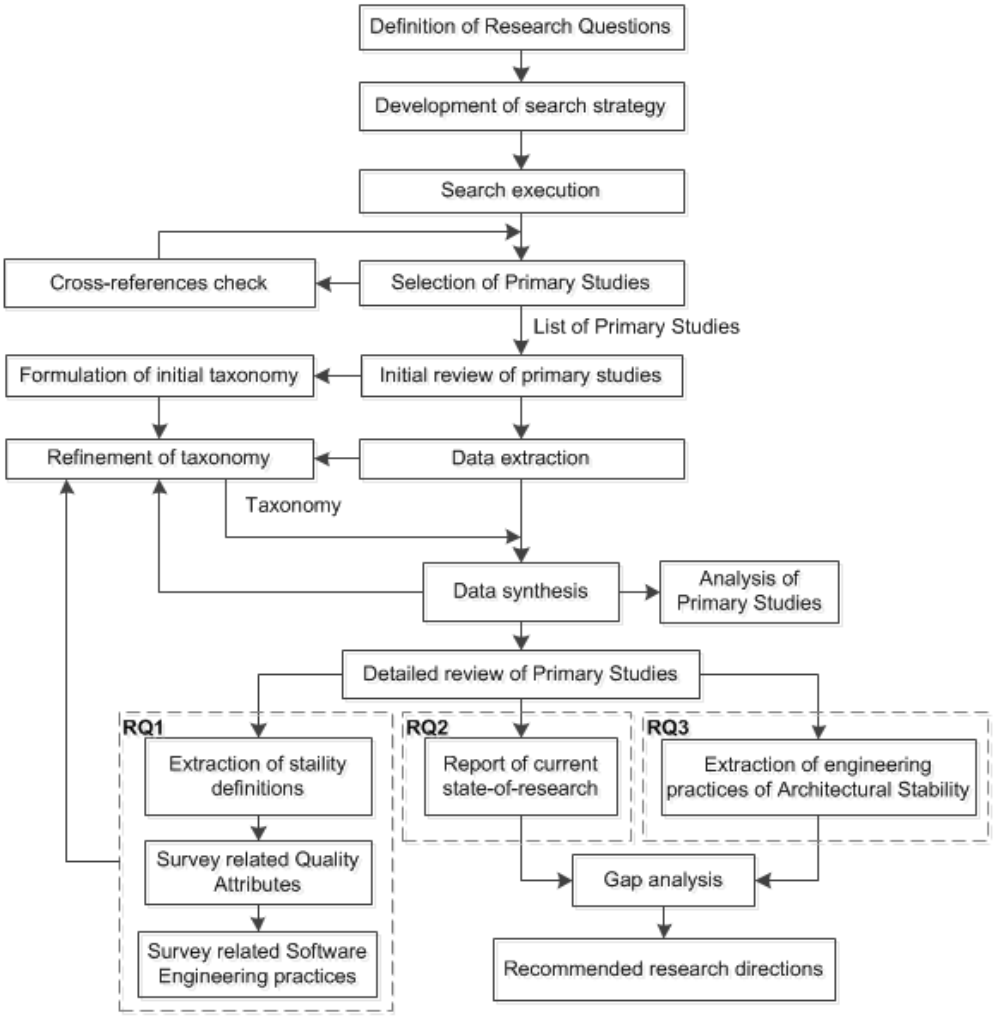

Figure A.1: Review Protocol

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 [93] [514] and most relevant electronic databases to computer science and software engineering [515]. 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 [94] —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 sections 2.6.2 and 2.6.3).

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.

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

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

As a result of this step, we obtained a total of 2418 papers (details in Table A.3). Search results were extracted as bibliography in BibTeX format, having a final collection of bibliographies for each data source. We, then, used JabRef [528], 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.

Table A.3: Search Results

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

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 to 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.

Table A.4: Selection Criteria of Primary Studies

	Inclusion Criteria
I1.	Papers published in conferences and journals, as full research paper, short and position paper presenting new and emerging ideas, as well as doctoral symposiums
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.	Papers with abstract not available
E3.	Papers not written in English language
E4.	Papers focusing on stability in other computer science areas (e.g. operating systems, robotics, networks, hardware, algorithms, logic programming, computational logic)
E5.	Papers focusing on stability in other disciplines (e.g. control theory or dynamic systems)
E6.	Papers focusing on stability of software product lines, project management or development process

A.5 Data Extraction

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. Stability data extracted from the primary studies according to the taxonomy dimensions is shown in Table A.5, A.6, A.7, A.8 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 [116] and the IEEE Recommended Practice on Software Reliability [147], which are not listed in any table.

Within the tables in this appendix, we used the following abbreviations: **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.5: 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
[111]	x				x			x		x	x		x		x		
[112]		x			x			x		x				x	x		
[113]		x			x			x		x				x	x		
[183]		x			x			x		x			x		x		
[200]				x				x			x		x		x		
[184]		x			x			x		x			x			x	
[186]		x			x			x		x			x		x		
[128]	x	x						x				x	x			x	
[210]		x			x	x			x					x	x		
[206]		x	x			x					x			x		x	
[201]				x				x			x		x		x		
[132]	x	x		x						x					x		
[207]		x	x			x					x			x		x	
[185]		x			x			x		x			x			x	
[190]		x		x				x		x					x		
[189]	x	x		x						x							
[135]	x	x		x						x					x		
[136]	x			x				x			x		x		x		
[211]					x	x			x					x			x
[16]	x		x	x								x					
[137]	x		x	x								x					
[192]		x		x				x		x					x		
[204]		x		x				x			x		x		x		
[191]		x		x				x		x					x		
[208]			x			x					x			x		x	
[187]	x	x		x						x							
[209]					x		x		x					x		x	
[194]		x		x				x		x					x		
[195]		x		x				x		x					x		
[138]					x		x		x					x			x
[193]		x		x				x		x					x		
[205]					x			x			x		x		x		

Table A.5 (*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
[197]		x		x				x		x					x		
[212]					x		x		x					x			x
[199]		x		x				x		x				x			
[188]	x	x		x						x							
[149]		x		x				x			x		x		x		
[202]	x	x						x			x		x			x	
[203]	x			x				x			x		x			x	
[198]				x				x		x				x		x	
[299]	x							x			x	x	x		x		
[196]		x		x				x		x					x		

Table A.6: 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
[217]		x															
[218]				x				x					x		x		
[115]																	
[222]		x			x			x			x		x		x		
[223]		x			x			x			x		x		x		
[226]		x				x					x			x	x		
[227]		x				x					x			x	x		
[224]		x				x					x			x	x		
[219]		x				x								x			
[163]			x					x		x					x		
[228]					x		x		x				x				x
[225]		x				x					x			x	x		

Table A.7: 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
[114]	x					x				x				x	x		
[282]				x											x		
[232]	x		x					x			x		x		x		
[234]	x		x					x			x		x		x		
[233]	x		x					x			x		x		x		
[200]				x				x			x		x		x		
[235]	x		x					x			x		x		x		
[247]	x	x				x						x		x	x		
[248]	x	x				x						x		x	x		
[249]	x	x				x						x		x	x		
[253]	x	x				x						x		x	x		
[118]				x				x						x	x		
[259]	x	x				x						x		x	x		
[258]	x	x				x						x		x	x		
[9]	x	x				x						x		x	x		
[252]	x	x				x						x		x	x		
[287]	x							x		x	x		x			x	
[126]		x						x		x			x		x		
[262]	x	x				x						x		x	x		
[119]				x				x						x	x		
[269]	x	x				x						x		x	x		
[268]	x	x				x						x		x	x		
[127]	x	x									x			x		x	
[263]	x	x				x						x		x	x		
[271]	x	x				x						x		x	x		
[270]	x	x				x						x		x	x		
[284]				x				x		x						x	
[237]				x				x			x		x		x		
[129]	x							x		x			x		x		
[264]	x	x				x						x		x	x		
[266]	x	x				x						x		x	x		
[272]	x	x				x					x			x	x		

Table A.7 (*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
[273]	x	x				x					x			x	x		
[236]			x	x				x			x			x	x		
[244]	x									x					x		
[130]		x						x			x		x		x		
[239]				x				x			x		x		x		
[70]	x					x						x		x	x		
[251]	x	x						x		x		x		x	x		
[260]	x	x				x						x		x	x		
[261]	x	x				x						x		x	x		
[131]	x										x		x		x		
[289]	x							x		x	x		x			x	
[285]				x				x			x			x	x		
[254]	x	x				x						x		x	x		
[243]	x					x								x	x		
[242]	x														x		
[255]	x	x				x								x	x		
[275]	x	x				x								x	x		
[230]	x					x								x		x	
[276]	x					x						x		x	x		
[278]	x					x						x		x		x	
[274]	x	x				x								x	x		
[265]	x	x				x						x		x	x		
[238]				x				x			x		x		x		
[241]	x	x				x					x				x		
[240]				x				x			x			x	x		
[288]	x							x		x	x		x		x		
[286]	x							x			x	x	x		x		
[229]	x		x			x					x			x			
[277]	x					x						x		x	x		
[245]	x	x		x						x					x		
[256]	x	x				x								x	x		
[139]		x												x		x	
[231]	x	x	x	x							x				x		

Table A.7 (*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
[181]		x				x				x	x			x	x		
[14]	x							x		x	x		x		x		
[291]	x							x		x	x		x		x		
[140]	x			x											x		
[283]	x			x						x		x			x		
[290]	x							x		x	x				x		
[292]	x		x		x	x								x	x		
[246]				x				x		x							
[281]					x	x						x		x		x	
[257]	x	x				x						x		x	x		
[267]	x	x				x						x		x	x		
[299]	x							x			x	x	x		x		

Table A.8: 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
[66]		x			x				x								
[117]	x							x			x		x		x		
[120]	x					x					x			x	x		
[121]	x					x					x			x	x		
[122]	x					x					x			x	x		
[123]	x					x					x			x	x		
[19]	x					x					x			x	x		
[307]					x		x		x					x			x
[124]	x					x					x				x		
[125]	x					x					x			x	x		
[293]	x					x					x			x	x		
[133]	x							x		x					x		
[39]	x					x					x			x	x		
[296]	x					x					x			x			x
[306]					x		x		x					x			x
[312]					x		x		x								x
[297]	x					x					x			x	x		
[134]	x							x		x					x		
[303]		x					x		x								x
[11]		x					x		x					x		x	
[241]	x	x				x					x				x		
[308]		x			x		x		x					x			x
[301]					x		x		x					x			x
[25]		x			x				x								
[305]					x		x		x				x				x
[159]					x				x								
[294]	x							x			x	x	x		x		
[152]		x			x				x								
[310]		x					x		x					x			x
[311]		x					x		x					x			x
[302]					x												
[309]		x					x		x					x			x

Table A.8 (*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
[298]	x							x			x		x			x	
[179]	x							x			x	x	x		x		
[295]	x							x			x	x	x		x		
[300]	x				x			x		x			x		x		
[299]	x							x			x	x	x		x		

A.6 Data Synthesis and Analysis

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 approach for synthesising findings is based on the synthesis method "thematic analysis/synthesis" [529], 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 A.7).

A.7 Analysis Results of Primary Studies

In this section, we present analysis results of the primary studies.

A.7.1 Demographic Analysis

Figure A.2 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.

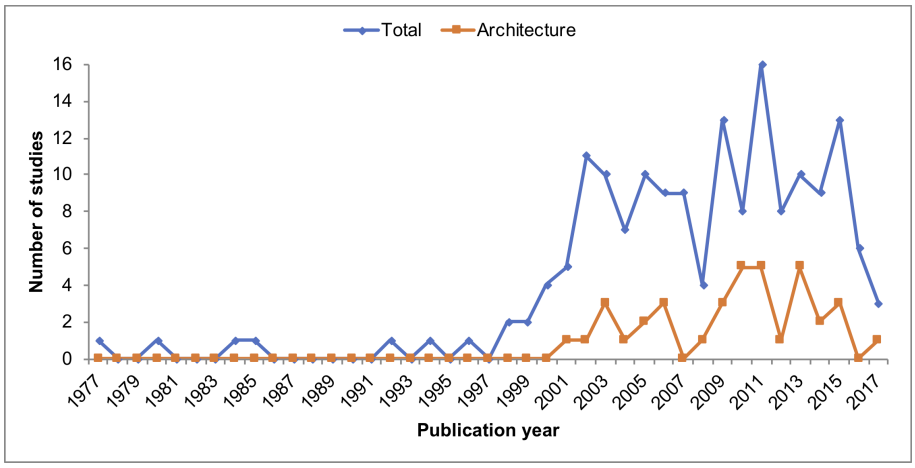

Figure A.2: Number of Studies per Publication Year

The distribution of the type of publications (Figure A.3) 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.

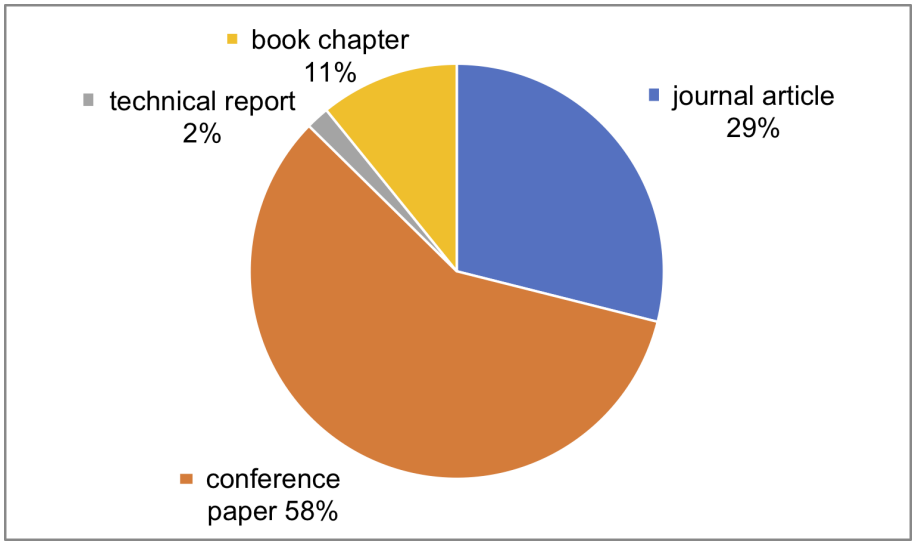

Figure A.3: Number of Studies per Publication Type

A.7.2 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.

Level (Where). Figure A.4 shows the distribution of studies considering stability at the 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).

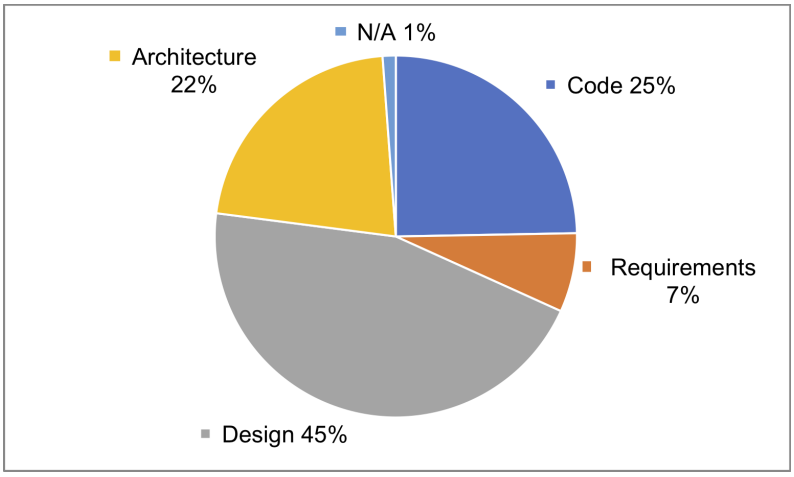

Figure A.4: Distribution of Primary Studies per Level

Aspect (What). Analysing the different aspects of stability found in the studies, Figure A.5 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).

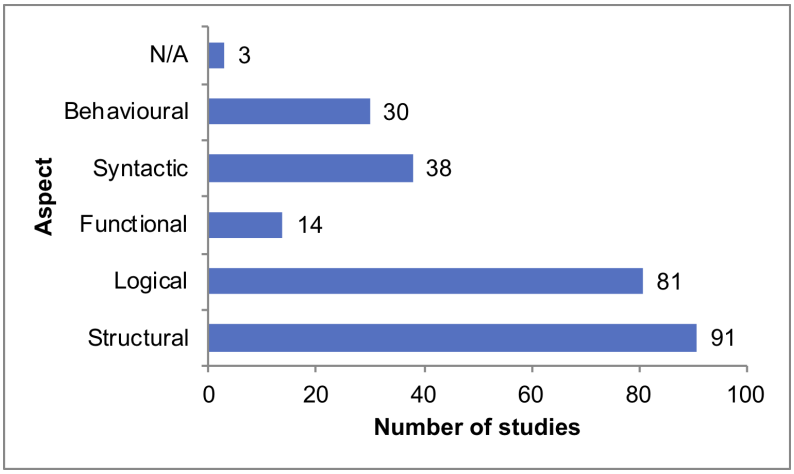

Figure A.5: Distribution of Primary Studies per Stability Aspect

Purpose (Why). The distribution of studies considering stability for different purposes is shown in Figure A.6. 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 as the other purposes (21 studies).

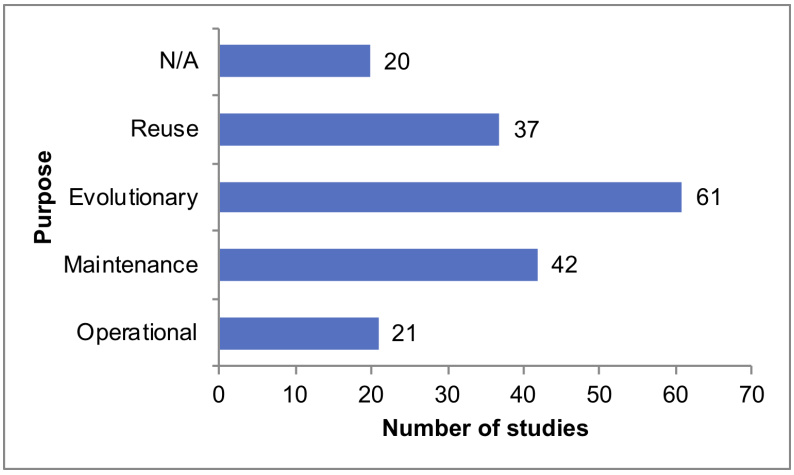

Figure A.6: Distribution of Primary Studies per Purpose

Time of consideration (When). Analysing the time dimension, Figure A.7 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 concerned about stability during the

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

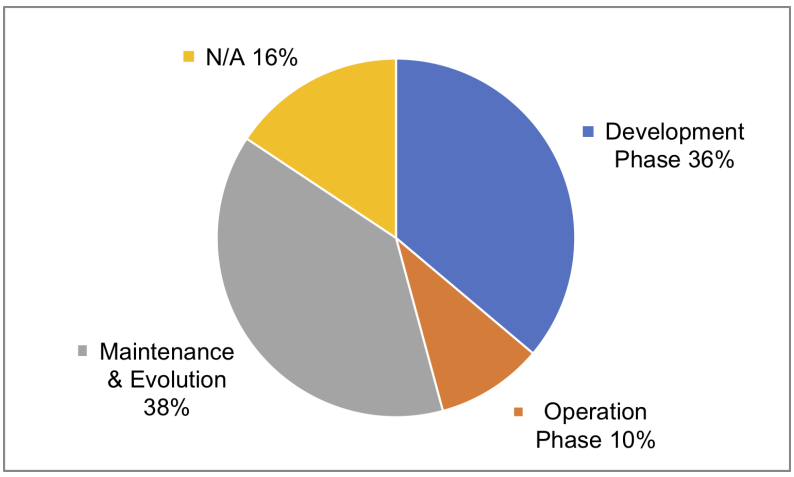

Figure A.7: Distribution of Primary Studies per Time of Consideration

Technique (How). The distribution of techniques by their temporal characteristics found in the studies is shown in Figure A.8. 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.

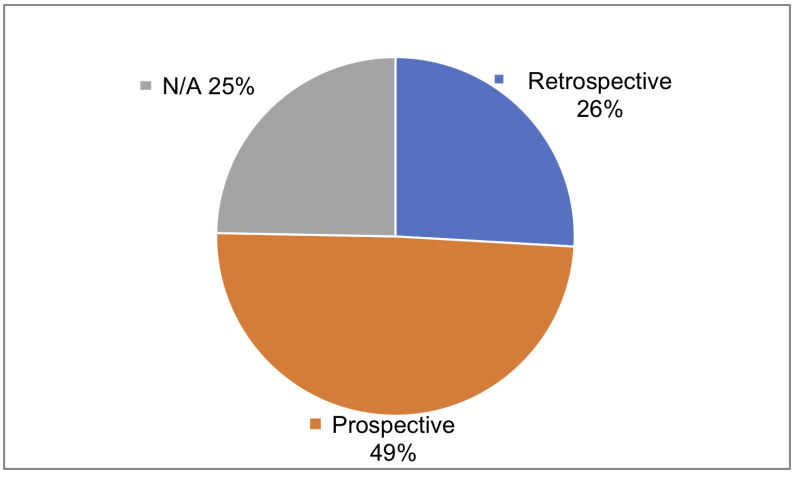

Figure A.8: Distribution of Primary Studies per Technique

Responsibility (Who). Figure A.9 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%).

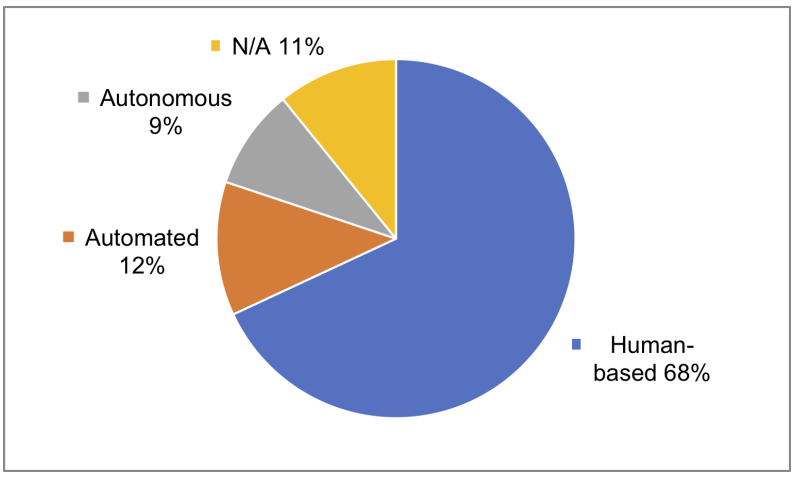

Figure A.9: Distribution of Primary Studies per Responsibility

Appendix B

Systematic Literature Review on Self-Awareness in Software Engineering: Summary of Findings

In this appendix, we present the summary of findings —related to stability engineering—of the systematic literature review on self-awareness in software engineering. The aim of this systematic review is to investigate how current research has adopted computational self-awareness to enrich the self-adaptation capabilities of autonomous software systems.

B.1 Summary of the Study

The contribution of this work is a Systematic Literature Review that compiles the studies related to the adoption of self-awareness in software engineering. The aim is to investigate the adoption of computational self-awareness concepts in autonomic software systems and explore how self-awareness is engineered and incorporated in software systems.

To this end, we conducted a systemic literature review following the guidelines for conducting systematic literature reviews [92]. From 591 studies found in search results, 70 studies have been selected as primary studies. We have analysed the studies from multiple perspectives, including: (i) motivations for employing self-awareness in software engineering, (ii) sources of inspiration in engineering self-awareness, (iii) approaches for engineering self-awareness, (iv) evaluation of self-awareness, and (v) the software paradigms that employed self-awareness. The findings of the review are summarised in Figure B.1.

B.2 Motivation for Employing Self-Awareness

The general motivation that has directed researchers towards self-awareness is the complexity, heterogeneity, large size of modern software systems, evolving functionality and quality requirements during run-time, emergent behaviours, and unpredictable changes of the highly dynamic operating environment [383] [530] [2] [531] [532].

More specifically, the motivation of employing self-awareness in software systems varied between a general one related to realising better autonomy for software systems, and

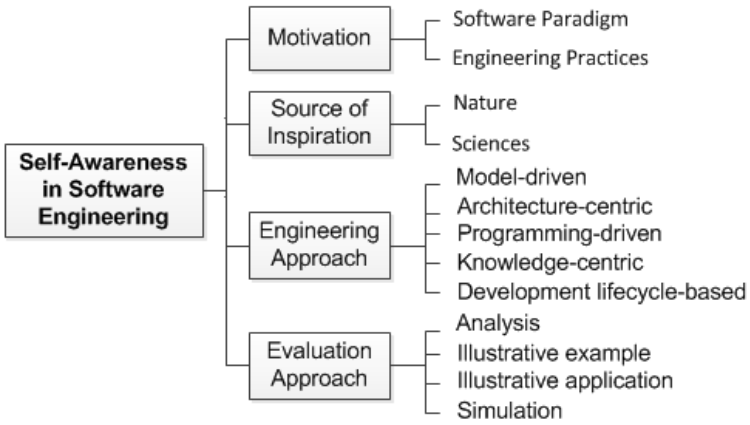

Figure B.1: Summary of Findings

others that are more specific. With respect to the former, researchers considered self-awareness for: (i) reasoning and engineering better adaptations with guaranteed functionalities and quality of service during runtime [533] [534] [464] [535] [383] [536] [537] [538] [539] [530] [532] [540] [541], (ii) managing complex systems without human intervention [542] [543] [414], (iii) dealing with real-world situations, operational contexts and dynamic environments of modern software systems to respond to such fluctuating environment and associated uncertainty [544] [383] [545] [546] [531] [532] [547] [548] [549] [550], (iv) managing complex trade-offs arising from adaptation due to conflicting goals [551] and the heterogeneity of the system [2] [552], and (v) realising intelligent software systems with sophisticated abilities [553] [464] [554] [555] [556].

Specific motivations (summarised in Table B.1) varied between domain-specific according to the software paradigm (e.g. ubiquitous applications, pervasive services, cloud-based services, mobile computing) and others driven by software engineering practices (e.g. formal specification, performance management, data access, security).

B.3 Sources of Inspiration

Few studies have clearly identified their source of inspiration in engineering self-awareness. Generally, nature and sciences inspired by nature are the main sources of inspiration in all studies. Examples of nature's inspiration include: biological systems [533] [553] [464], natural ecosystems [571] [559] [560] and human beings [533] [566] [554]. Sciences inspiring self-awareness are control theory [573], biology [464], psychology [460] [383] [2] [3] [548] [550], and cognitive science [464]. Table B.2 summarises inspirations cited in primary studies.

Within the studies mentioning their source of inspiration, we have found that the majority of studies named only their source of inspiration. More details, albeit in an abstract form, on about how self-awareness approaches are inspired by nature or sciences are found in a few number of studies; such as [533] [553] [554] [460] [383] [560]. The exception that could be found is [464], where the authors have explicitly mentioned how

Table B.1: Specific Motivations of using Self-Awareness

Study	Motivation
	Driven by Software Paradigm
[557]	Autonomous adaptations of hardware/software functionalities in ubiquitous computing applications to meet the dynamic requirements of various environmental situations and provide better QoS
[558]	Creating cloud markets platforms with self-* properties harmoniously working together in order to be capable to adapt effectively to dynamic changes in user requirements, services, and variability in resources.
[559]	Modelling integrated pervasive services and their execution environments, in a way that diverse issues of context-awareness, dependability, openness, flexible and robust evolution, can be addressed
[560]	The need for runtime self-adaptive interactions between pervasive computing services
[561]	Achieving parallelism within a reasonable cost and time range for data streaming applications operating in distributed environments
[562]	Acceleration and efficiency of biocollections' information extraction, while keeping the quality of the results similar to what capable humans can provide
[552]	The limitations of the security measures on mobile devices, and the lack of cooperation between different security solutions running on the same device
	Driven by Engineering Practices
[563]	The motivation of including the notion of self within object-oriented formal specification languages is to facilitate reasoning about object interaction.
[564]	The detection anomalies in the functioning of internet-based services and fault localisation (i.e., locating the responsible sub-services) are easier if service elements are aware of their own health status, determined by whether the current observed behaviour is consistent with expectations.
[565]	The need to access distributed and dynamic high-dimensional data about resources heterogeneity in a timely fashion in large, decentralised, resource-sharing environments
[566]	The invention of new abstractions as conceptualisation necessary to determine the behaviour of a software needed by users and the implementation details.
[567]	Enabling change at run-time for evolution purposes
[568], [569], [570]	The need to predict the performance of running services at run-time and related resources management
[571]	Balancing resources usage in order to improve performance, utilisation, reliability and programmability
[531]	Solving problems caused by QoS interference in shared resources environment to achieve auto-scaling for cloud-based services
[548], [550]	Dynamic context management
[572]	The complexity of managing end-to-end application performance
[549]	performance prediction that is necessary for efficient resource management
[556]	The need to re-arrange own knowledge structures for compactness and efficiency to survive for long periods in a demanding environment
[552]	The limitations of the security measures on mobile devices, and the lack of cooperation between different security solutions running on the same device

self-awareness have been inspired by biology and cognitive science. The mapping between the source of inspiration and the research work conducted in the study is expected to be clearly communicated. Further, studies investigating how self-awareness could be inspired by nature and other sciences can help to advance self-aware software systems.

Table B.2: Source of Inspiration in Engineering Self-Awareness

Study	Inspiration
	By Nature
[533]	Biological cell and the system of a human organisation (e.g. a company or government department)
[553]	Biological systems: the immune system and ant colonies
[566]	Human beings
[571]	Biological organic nature
[554]	Human wisdom
[559], [560]	Natural ecosystems
	By Sciences
[573]	Control Theory
[464]	Biology and cognitive science
[460], [2], [3]	Psychology
[383]	Psychology, philosophy and medicine
[548], [550]	Psychology and philosophy

B.4 Approaches for Engineering Self-Awareness

Engineering self-awareness aims for encoding self-aware properties within the software systems in an attempt to provide systematic treatment for managing the software system state, knowledge and execution environment. This research question looks for the approaches that have been used to engineer self-aware software systems and categorise these approaches.

In the literature, different approaches to engineering self-awareness in software engineering are found. On one hand, we have observed that 19 out of the 70 primary studies did not provide any engineering approaches for self-awareness in software engineering. These works have presented visions, outlined challenges, and raised questions. On the other hand, the remaining 51 studies claimed to provide engineering approaches for self-awareness. We have categorised these approaches into: model-driven, architecture-centric, programming-driven, knowledge-centric, and development lifecycle-based approaches. Table B.3 lists the engineering approaches categories and their related studies.

Table B.3: Engineering Approaches and Related Studies

Engineering Approach	Studies
Model-driven	[568], [558], [569], [535], [570], [532], [540], [548], [574], [572], [562], [575], [414], [576], [550], [556], [549]
Architecture-centric	[564], [577], [544], [557], [571], [578], [551], [545], [559], [383], [536], [543], [546], [3], [538], [579], [2], [539], [560], [531], [547], [580], [581], [582], [552]
Programming-driven	[563], [561]
Knowledge-centric	[565], [554], [555], [583], [530]
Development lifecycle-based	[537]

Figure B.2 shows the distribution of studies with respect to the classification of engineering approaches. Architecture-centric and model-driven approaches are found the most dominant approaches in the current literature. Other categories of approaches have taken less attention in the research community.

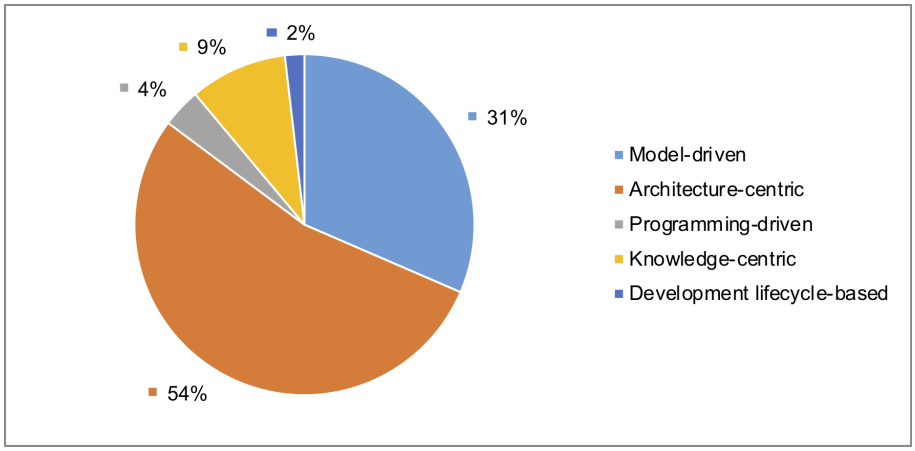

Figure B.2: Distribution of Studies by Self-Awareness Engineering Approaches

B.5 Evaluation of Self-Awareness

We observed that 28 papers out of the 39 (that proposed engineering approaches) have provided some kind of evaluation for their approaches. We categorise these approaches into the following categories: analysis-, illustrative example-, illustrative application-, and simulation-based evaluation. Figure B.3 illustrates the distribution of studies by evaluation approach categories. The majority of studies have evaluated their work using either illustrative example or illustrative application. Simulation-based evaluation, featuring scalability, is significantly less used. Table B.4 lists the evaluation approaches categories and their related studies.

Table B.4: Evaluation Approaches and Related Studies

Evaluation Approach	**Studies**
Analysis	[563]
Illustrative example	[577], [568], [571], [569], [545], [535], [536], [530], [570], [2], [540], [532], [584]
Illustrative application	[564], [557], [573], [551], [383], [543], [546], [3], [539], [538], [560], [561], [572], [576], [549], [550], [562], [575], [414], [585]
Simulation	[558], [579], [547]

We also investigated the evaluation criteria that have been used in the mentioned studies, and then we present how each of the approaches addressed them. Table B.5 lists the evaluation criteria and the corresponding studies.

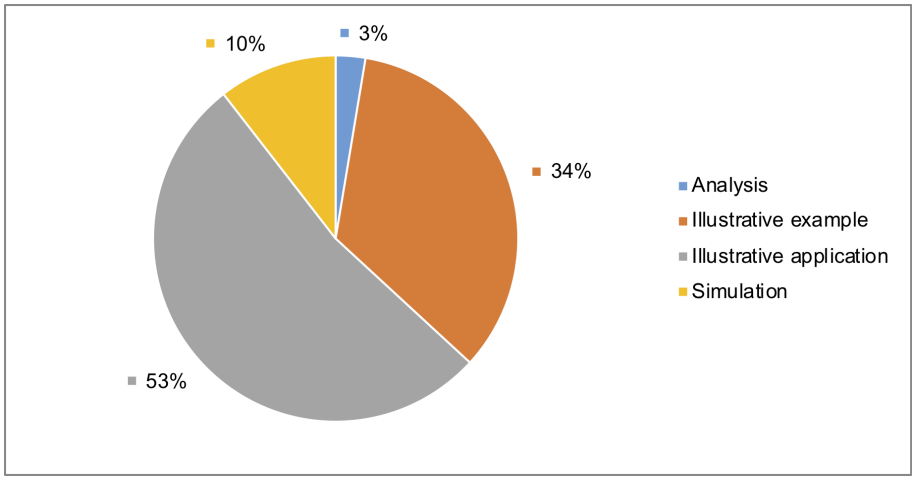

Figure B.3: Distribution of Studies by Self-Awareness Evaluation Approaches

Table B.5: Evaluation Criteria and Related Studies

Study	**Evaluation Criteria**
[564]	Accuracy
[573], [551], [543]	Accuracy, Efficiency
[557]	Processing time
[558]	Number of bids, tasks, allocations, average price, market revenue
[383]	Reduction in communication
[579]	Number of violations
[539]	Power efficiency, execution time
[546]	Power consumption
[538]	Lookahead, latency, number of achieved goals
[3]	Accuracy, adaptation quality, overhead, reliability
[560]	Local resources consumption, time performance
[561]	Performance per Watt
[572]	Number of violations
[549]	Prediction accuracy of response time
[550]	Lines of code, complexity, technical debt
[562]	Number of required humans, cost

With respect to the overhead resulting from adopting self-awareness in software systems, only 8 of the studies have reported the overhead of adopting self-awareness. All of them considered overhead in terms of computation time. In more details:

- Authors in [571] reported that the proposed approach is low-overhead without presenting experimentation results to demonstrate this claim.
- In [551] and [543], the authors reported that the overhead of the proposed approach is very low and that the system can take adaptation decisions in 20.09 nanoseconds. However, other overheads related to adopting self-awareness, e.g. the overhead of monitoring, registering events and taking an action, have not been considered.
- [539] reported on the overhead related to the monitoring component of the approach.

The reported runtime overhead is within 1-2%, which the authors consider it to be negligible compared to the normal system's execution time.

- [560] reported the overhead of propagating the monitoring information across a network and stated that the overhead is "acceptable" and limited. These approaches consider only the overhead of the monitoring activity.

- [550] reported a slight increase in the execution time due to the time needed to build contextual models and considered that increase as negligible.

- The study of [572] has provided a more profound analysis of the overhead. The authors reported on the overhead of analysing the captured information and forecasting, as well as the overhead of the adaptation process. They reported that both overheads depend on the data, configuration settings, the techniques used for performance forecasting and the application specifications.

B.6 Software Paradigms Employing Self-Awareness

Table B.6 lists the software paradigms found in the primary studies and related studies. Figure B.4 shows the distribution of studies by software paradigms (note that some studies appear multiple times under different categories, which interprets the total number of studies appearing in the figure is greater than the number of primary studies). The majority of studies considered self-awareness for autonomous computing (49%), i.e. engineering self-adaptive software systems as a general software paradigm, not explicitly designed for a particular paradigm or application type. Service-oriented systems and cloud-based services also received attention in a good number of studies (15% each), and less attention to ubiquitous and pervasive computing (9%) and distributed systems (7%). Within distributed systems, some studies considered a certain type of applications operating in decentralised environments, such as artificial intelligence systems [553], distributed smart cameras [383] [3]. Single works focused on software-intensive systems [555], stream programming [561], mobile computing [552] and Internet of Things [585].

The observation that the majority of the proposed work tends to be generic and not explicitly designed for a particular paradigm or application type implies that generality can come with advantages and disadvantages. Generality can imply the application and evaluation of the proposed work under different contexts and applications, reflection on their strengths and weaknesses in dealing with the said paradigm. This can consequently provide inputs for further improvements and extensions. On the other hand, employing self-awareness can take simplistic assumptions, or tend to be limited when addressing the requirements of some paradigms, where speciality and customisation are desirable for more effective adaptations. Self-awareness that considers characteristics of particular software paradigms will result in advancing these paradigms. Yet, the validity of these observations can be subject to further empirical studies.

Table B.6: Software Paradigms employing Self-Awareness

Software Paradigms	Studies
Self-adaptive Software Systems	[577], [533], [534], [566], [567], [573], [571], [551], [554], [535], [383], [536], [543], [538], [539], [530], [532], [583], [574], [572], [580], [575], [586], [587], [581], [582], [588], [549], [584], [589], [590], [556], [591], [592]
Service-oriented Systems	[564], [542], [568], [545], [546], [579], [570], [540], [414], [550]
Cloud-based Services	[569], [558], [578], [383], [2], [3], [531], [547], [585]
Distributed Systems	[565], [553], [383], [537], [3]
Ubiquitous and Pervasive Computing	[544], [557], [464], [559], [560], [576]
Software-Intensive Systems	[555]
Stream Programming	[561]
Mobile Computing	[552]
Internet of Things	[585]

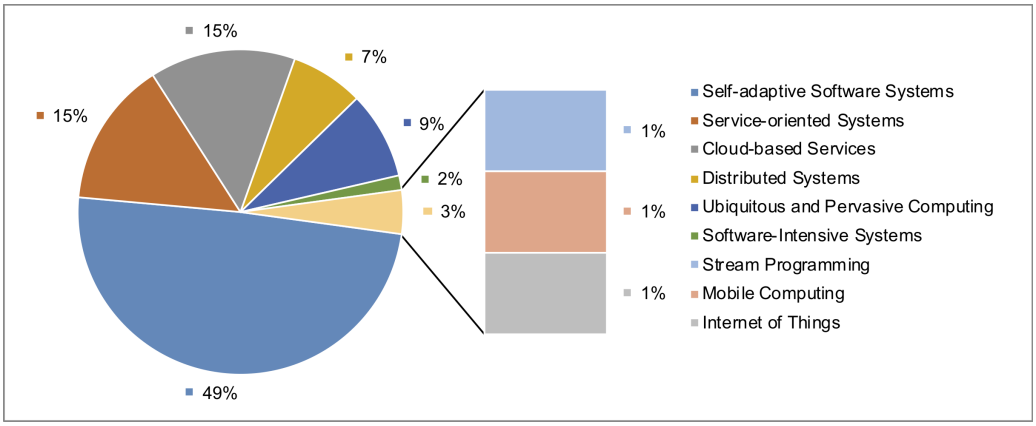

Figure B.4: Distribution of Studies by Software Paradigms

B.7 Summary

The main findings of this systematic review are summarised as follows. There is a growing attention to adopt self-awareness in modern software systems. Self-awareness has been used to enable self-adaptation in systems that exhibit uncertain and dynamic behaviour during their operation. Motivations for employing self-awareness varied between the general purpose of realising better autonomy for software systems and domain-specific purposes. Self-awareness was considered for self-adaptive software systems as a general software paradigm, with few studies focusing on a particular software paradigm or application type. The approaches for engineering self-aware software systems can be categorised as model-driven, architecture-centric, programming-driven, knowledge-centric and development lifecycle-based approaches. Yet, most of the approaches for engineering self-awareness tend to be architectural in nature. Evaluating self-awareness engineering approaches and exclusive mapping with their sources of inspiration still need to be addressed. Self-awareness was considered for self-adaptive software systems as a gen-

eral software paradigm, with few studies focusing on a particular software paradigm or application type. The review reveals that self-awareness for software systems is still a formative field and that there is a growing attention to incorporate self-awareness for better reasoning about the adaptation decision in autonomic systems.

Appendix C

Systematic Mapping Study on Managing Trade-offs in Self-Adaptive Architectures: Summary of Findings

In this appendix, we present the findings of the systematic mapping study on managing trade-offs in self-adaptive architectures. The study aims at analysing the research landscape that has explicitly addressed trade-offs management for self-adaptive software architectures, to obtain a comprehensive overview of the current state of research on this specialised area.

C.1 Summary of the Study

The contribution of this work is a Systematic Mapping Study analysing the research landscape related to managing trade-offs of self-adaptive software architectures. The aim was to draw a picture of the current state of the research in this specialised topic, to help researchers and developers identify what has been established so far, to understand which techniques have seen particular emphasis, as well as what is still under research and warrant greater attention.

To this end, the study was conducted methodologically, following the standard guidelines for conducting secondary studies [515] [593] [594], in order to ensure the quality of the analysis. The search was conducted in five main publications databases resulting in 462 studies that have been reviewed, and 20 relevant studies have been selected as primary studies for this study. The contributions of the studies that explicitly considered trade-offs management for self-adaptive software architectures are summarised below:

- [595] employed analysis-oriented models to support analysing and reasoning about non-functional system properties; precisely performance and reliability.
- [596] work resulted in optimised trade-offs between system design complexity, system performance and power impact, by proposing on-line self-test features in a multi-/many-core architecture.
- [597] attempted engineering resource-adaptive software systems targeted at small mobile devices, by empowering users to control trade-offs among service-specific

aspects of quality of service and coordinating resource usage among several applications.

- [598] further developed their earlier research [597]by presenting a framework for engineering resource-adaptive systems that empowers users to control trade-offs among a set of quality aspects, and coordinates resource usage among several applications.

- [499] presented a paradigm of parallel computing for giving embedded systems the ability to explore and claim resources in a certain neighbourhood, where the trade-off between flexibility and cost is considered.

- [500] introduced a dynamic adaptation for service-based systems that minimise the adaptation costs and guarantees the required quality of service, based on an optimisation model.

- [471] presented a research agenda for self-adaptive systems towards being able to dynamically adapt to new environmental uncertain contexts. This work called for research into how self-adaptive systems envisage runtime trade-offs of requirements that are present as the environment changes; i.e. how self-adaptive systems can have runtime flexibility to temporarily ignore some requirements in favour of others.

- [599] studied trade-offs between safety and capability in the autonomic agent infrastructure of self-organising real-time systems. This work used off-line simulations to tune the trade-off at deployment time – based on what is known or expected of the environment – as well as to monitor and change those assumptions when necessary.

- [600] presented a model-driven framework targeted at dynamic settings for self-architecting service-oriented systems in which requirements might change, taking into consideration trade-offs that reflect stakeholders' priorities.

- [601] proposed an adaptation process for service-based self-adaptive systems, which guarantees a trade-off between energy consumption and quality of service offered while maintaining suitable revenues for the service provider.

- [502] proposed a control-theoretic method for self-tuning software systems, combining goal models with feedback controllers, to dynamically tune the preferences of different quality requirements and make dynamic trade-off among conflicting soft goals. That was achieved through preference-based goal reasoning procedure, in order to find Pareto optimal configurations for the dynamic quality trade-off.

- [501] extended their previous work [601] by developing an adaptation framework for service-based applications that can be used to reduce power consumption according to the observed workload. This work aimed at guaranteeing a trade-off between energy consumption and performance, using stochastic Petri nets for the modelling where their analyses give results about the trade-offs.

- [503] proposed a quality-driven self-adaptation approach for designing architectures of self-adaptive software systems, which incorporates design decisions as the bridge between requirements- and architecture-level adaptations. This was based on making value-based quality trade-off decisions with the aim of maximising system-level

value propositions and using a preference-driven goal reasoner to reconfigure the runtime goal models based on the results of dynamic quality trade-off.

- [602] defined a model-based approach for design spaces representation and exploration which entails a search-based mechanism that points out decision trade-offs between feedback controls and performance overhead to find out a set of Pareto-optimal candidate architectures for self-adaptive software systems.

- [603] proposed an approach for service selection in a pervasive environment, framed as a quality of service optimisation problem. The approach evaluates at runtime the services optimal binding as well as the trade-off between the remote execution of software fragments and their dynamic deployment on local nodes of the computational environment.

- [604] reported the results of a controlled experiment that evaluates the design of self-adaptive systems using a search-based approach, in contrast to the use of a style-based non-automated approach, for finding out subtle effective designs and providing well-informed means to reveal quality attributes trade-offs.

- [458] addressed trade-offs between the global benefit of the cloud and local optimisation of virtual machines from one side, and between the global benefit of the cloud and overhead in the design for selecting an elastic strategy from another side, in order to dynamically and efficiently determine an architectural elastic strategy that produces globally-optimal benefit.

- [605] proposed an approach for analysing and evaluating trade-offs between the system adaptability and other system quality attributes, like availability or cost. The approach was based on a set of metrics that allows evaluating the system adaptability at the architecture level to guide architecture decisions on system adaptation for fulfilling system quality requirements.

- [606] proposed a reference architecture for context-aware adaptive systems, where the heuristics and metrics of design architecture strategies are used to refine conceptual architectures in trade-off analysis to deal with non-functional requirements.

- [607] reported the results of another controlled experiment, following their earlier work [604]. This experiment evaluated the design of self-adaptive systems using a search-based approach for explicitly eliciting design trade-offs, in contrast to a non-automated approach based on architectural styles catalogues, with the goal of investigating to which extent the adoption of search-based design approaches impacts on the effectiveness and complexity of resulting architectures.

A comprehensive view about the research landscape is shown by the correlation matrix in Table C.1, summarising the research conducted in the primary studies with respect to the software paradigms, the quality attributes and the mechanisms for trade-offs management.

Table C.1: Correlation of Software Paradigms, Quality Attributes and Mechanisms

Mechanism	**Software Paradigm**							
	self-adaptive	embedded	pervasive	large-scale	real-time distributed	mobile	cloud-based	service-based
Utility theory						[597] quality attributes; [598] flexibility, cost		[600] quality attributes
Stochastic Petri	[605] quality attributes, adaptation cost							[601] safety, adaptation cost; [501] performance, energy consumption
Multi-objective optimisation	[602] feedback control loop, cost; [604] quality attributes, cost		[603] flexibility, cost	[596] performance, reliability			[602] feedback control loop, cost	[500] quality attributes
Pareto-optimal solutions	[502] quality soft goals; [607] quality attributes							
Value-based reasoning	[502] quality soft goals; [503] quality attributes							
Analysis-oriented method	[595] performance, energy consumption, design complexity							
Invasive algorithms		[499] quality attributes						
Requirements reflection	[471] quality attributes, adaptation cost							
Simulations					[599] performance, energy consumption			
Objective functions	[458] quality attributes							
Heuristics						[606] quality attributes		

C.2 Quality Attributes investigated in Trade-offs Management

Analysing the details of research found in the literature, we have listed the quality attributes investigated in trade-offs management. Figure C.1 illustrates the statistics of these attributes among the primary studies. The major case of trade-offs management considered quality attributes on a general level. Special attention was given to performance and cost. Other attributes, such as adaptation cost, safety, reliability, were considered in single research efforts.

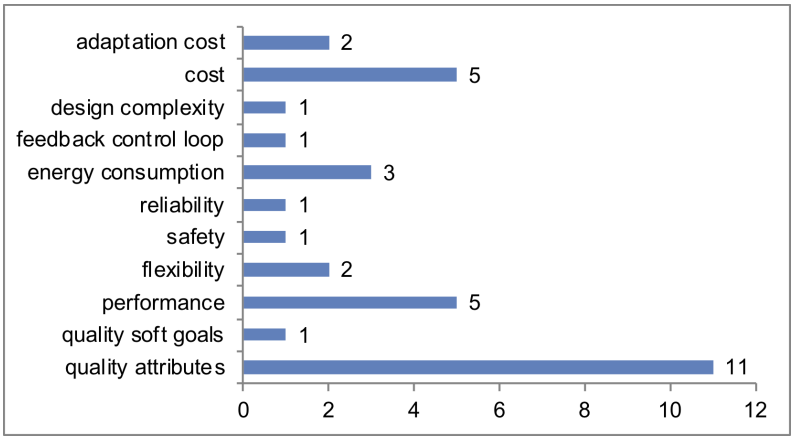

Figure C.1: Distribution of Quality Attributes investigated in Trade-offs Management

In the case of considering specific attributes, Table C.2 summarises the related studies and the attributes considered. Beside quality attributes, we have also considered feedback loops in trade-offs management, for feedback loops become a crucial element of the overall architecture in engineering self-adaptive software systems. Feedback loops can carry information about emerging or implied behaviour of the system and imply new trade-off that needs to be managed. Considering multiple feedback loops, or multiple decisions from a feedback loop, or internal and external feedback loops, more trade-offs will arise and need to be managed.

Another observation is that the majority of the studies, considering specific attributes, were concerned only with two attributes for trade-offs as examples in illustrating their approaches. Examples include [595] considered performance and reliability only, [601] and [501] considered performance and energy only. However, the formalism behind their trade-offs management process might not be limited to treating only two quality attributes.

Considering multiple quality attributes in trade-offs management will result in selecting better adaptation action that is able to fulfil multiple qualities. With many efforts and claims indicating the validity of this statement, we still need formal and rigorous investigations. Empirical studies provide the means for this, but no controlled experiments have been performed to provide that evidence. Afterwards, we call for more comprehensive trade-offs management approaches that consider multiple specific quality attributes.

Table C.2: Studies Considering Specific Attributes in Trade-offs Management

Study	Attributes
[595]	performance, reliability
[596]	design complexity, performance, power impact
[499]	flexibility, cost
[500]	adaptation cost, quality attributes
[599]	safety, resources
[600]	stakeholders' priorities
[601]	performance, energy
[501]	energy consumption, performance
[602]	feedback control loop, performance overhead
[603]	flexibility, cost
[458]	global QoS, cost

C.3 Mechanisms used in Trade-offs Management

The trade-offs mechanisms and their related studies are listed in Table C.3. Analysing the extracted mechanisms, we identified that utility theory and multi-objective optimisation appeared to be the most used techniques. Some efforts approached the use of stochastic Petri nets, value-based reasoning and Pareto-optimality. We have identified these mechanisms, listed as follows:

- Utility theory was used to model and quantify the quality of service trade-offs [597] [598] [600] for engineering resource-adaptive software systems targeted at small mobile devices in order to coordinate resource usage among several applications.
- Stochastic Petri nets were proposed for modelling trade-offs for service-based applications [601] [501] [605].
- Multi-objective optimisation was employed for optimising trade-offs between system design complexity, system performance and power impact [596], for minimising the adaptation costs while guaranteeing the quality of service [500], for pointing out decision trade-offs between feedback controls and performance overhead [602] [604], as well as for optimising service selection [603].
- Pareto-optimal solutions were also used to point out trade-offs decision using a search-based mechanism [607], and to find optimal configurations for the dynamic quality trade-off for self-tuning [502]
- Value-based reasoning was used to make design decisions that bridge between requirements- and architecture-level adaptations [503] and to dynamically make trade-off among quality requirements [502].

Table C.3: Trade-offs mechanisms and related studies

Mechanism	Related Studies
Utility theory	[597], [598], [600]
Stochastic Petri	[601], [501], [605]
Multi-objective optimisation	[596], [500], [602], [603], [604]
Pareto-optimal solutions	[502], [607]
Value-based reasoning	[502], [503]
Analysis-oriented method	[595]
Invasive algorithms	[499]
Requirements reflection	[471]
Simulations	[599]
Objective functions	[458]
Heuristics	[606]

C.4 Time Dimension of Trade-offs Management Approaches

Analysing the time dimension of these mechanisms (see Figure C.2), i.e. when these mechanisms tend to operate; we found 50% of the mechanisms were design-time mechanisms, approximately equals to 45% runtime ones and 5% off-line. Design-time and runtime mechanisms studied and analysed above are meant to be either design decisions or runtime adaptation decisions respectively. For both types of decisions, linkage of architectures with requirements is expected to enrich and better-inform the trade-offs decisions. More precisely, design-time trade-offs management requires linkage with requirements elucidated while designing the system, and runtime trade-offs management requires runtime monitoring of requirements changes. Such linkage should, then, employ requirements reflection, as proposed in [471].

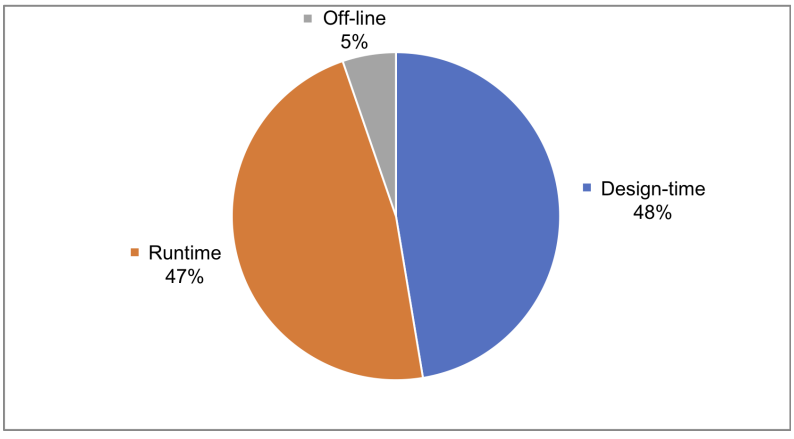

Figure C.2: Distribution of Time Dimension of Trade-offs Management Mechanisms

C.5 Summary

The study contributes to the understanding of the state of the research in this area and paves the way for solutions from both academia and industry. The results show a constant interest in finding solutions for trade-offs management at design-time and runtime, as well as the success of research initiatives even when new research challenges are found. Yet, the findings call for a foundational framework to analyse and manage trade-offs for self-adaptive software architectures, both while designing self-adaptive systems and at runtime during their operation, that can explicitly consider specific multiple quality attributes, the runtime dynamics, the uncertainty of the environment and the complex challenges of modern and ultra-large-scale systems.

Appendix D

Symbiotic Simulation Environment for Self-Adaptive and Self-Aware Architectures

In this appendix, we present the modelling and simulation environments for self-adaptive and self-aware cloud architectures, namely *SAd-CloudSim* and *SAw-CloudSim*. The proposed toolkits build on the widely adopted cloud simulation environment *CloudSim* [36] [5], due to its modular architecture that allows further extensions.

Organisation. The rest of this appendix is organised as follows. In section D.1, we describe relevant background about *CloudSim* the cloud simulation toolkit on which we build our simulation environment. Section D.2 presents the architecture of the proposed framework. Section D.3 presents technical details about the design and implementation. In section D.4, we experimentally validate and evaluate the performance and overhead of the tool. In section D.5, we discuss work related to simulators of self-adaptive, self-aware and cloud systems.

D.1 Background

The *CloudSim* simulation toolkit [5] [36] is currently one of the mostly-used general purpose cloud simulation environments [608], and the most sophisticated discrete event simulator for clouds [609]. Due to its modular architecture, it has been widely adopted and used in many further extensions modelling and simulating cloud-related problems.

Figure D.1 shows a cloud environment represented by the architecture of CloudSim. CloudSim defines the core entities of a cloud environment, such as datacenters, hosts physical machines (PMs), virtual machines (VMs), applications or user requests (called cloudlets) [36] [5]. Datacenter is the resources provider, simulating the infrastructure of the cloud, and hosts which run virtual machines responsible for processing user requests. Computational capacities of PMs and VMs (CPU unit) are defined by *Pe* (Processing Element) in terms of million instructions per second (MIPS) [36] [5]. Processing elements in a PM are shared among VMs, and among requests in a VM. The Datacenter Broker

is responsible about the allocation of requests to VMs. Once the simulation period is started, the requests are scheduled for execution, and the cloud behaviour is simulated.

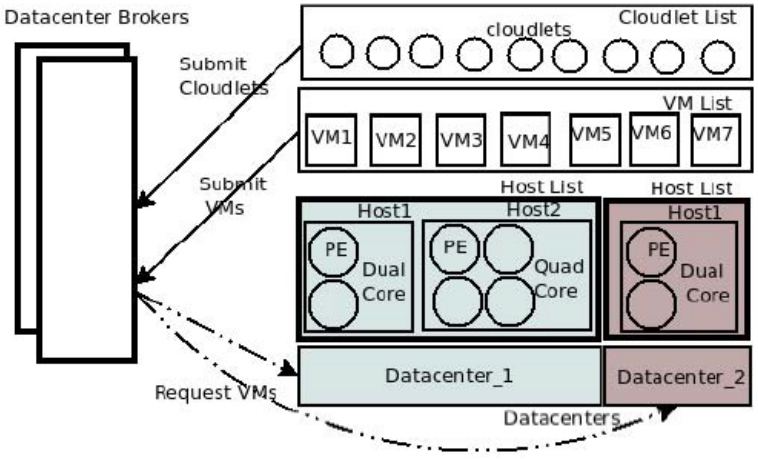

Figure D.1: CloudSim Architecture [5]

D.2 SAd-/SAw-CloudSim Architecture

In this section, we outline the architecture of SAd/SAw-CloudSim, the extensions made to CloudSim core framework and the rationale behind them. Figure D.2 and D.3 show the multi-layered design of CloudSim with the architectural components of *SAd-CloudSim* and *SAw-CloudSim* respectively (new components are shown in dark boxes).

Generally, the proposed environments are built on top of the CloudSim core simulation engine and CloudSim core. Extensions for some core classes of CloudSim were necessary for adaptation and awareness capabilities (more details in section D.3). The Self-Adaptation layer is added on top of the cloud core architecture, to model the adaptation controller of a self-adaptive software system. Researchers and practitioners, willing to design an adaptation technique or study the efficiency of an existing one, would need to implement their techniques in this layer. The Self-Awareness layer combines the self-awareness and self-expression capabilities, as well as necessary monitoring components. The top-most layer is the Simulation Application, inherited from CloudSim, that models the specification of the simulation to be conducted using the tool. Such specifications allow configuring the simulation of dynamic workloads, different service types and user requirements.

D.2.1 Modelling Self-Adaptation

A foundational self-adaptation controller consists of: (i) monitor for correlating quality data, (ii) detector for analysing the data provided by the monitor and detecting violations in order to trigger adaptation when necessary, (iii) adaptation engine to determine what

needs to be changed and select the optimal adaptation strategy, and (iv) adaptation executor responsible for applying the adaptation action on the underlying infrastructure. Our initial implementation of *SAd-CloudSim* includes this foundational version of adaptation controller. Such components could be further extended to study more complex adaptation mechanisms, such as pro-active adaptations or MAPE-K adaptation process [21].

The *Monitor* component is responsible for monitoring the achievement of quality requirements. The *Detector* checks any violations occurring during runtime against quality goals. Whenever a violation is detected, adaptation is triggered. The *Adaptation Engine* is responsible for analysing the current situation and selecting the optimal adaptation strategy that would achieve the quality targets, e.g. increasing VMs capacity, increase the number of PMs. The selected adaptation tactic is executed dynamically during runtime on the cloud infrastructure by the *Adaptation Executor*.

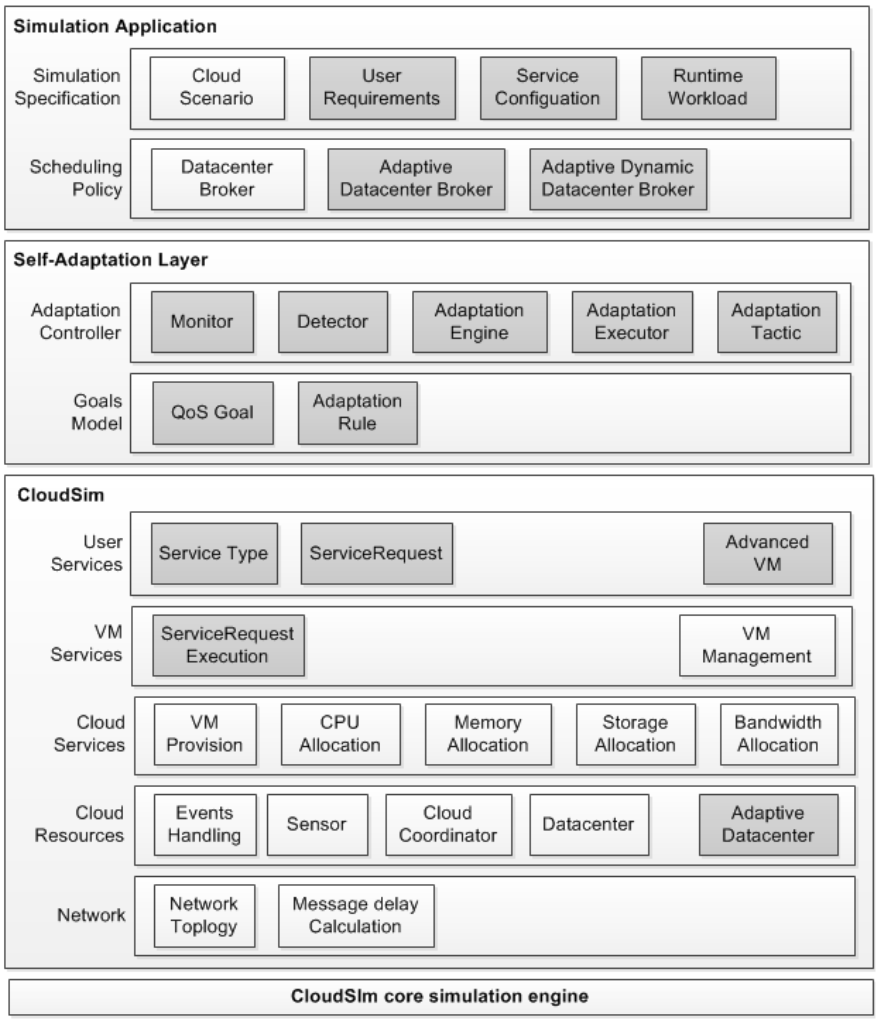

Figure D.2: *SAd-CloudSim* Architecture

D.2.2 Modelling Self-Awareness

Modelling self-awareness capabilities in a cloud architecture requires the following components: (i) QoS monitoring, (ii) different self-awareness capabilities as the system requires, and (iii) self-expression capability to execute adaptations. The monitoring component is composed of sensors responsible for measuring actual quality data, and the QoS monitor responsible for correlating data from sensors and monitoring changes in workload and quality attributes during runtime. The self-awareness component contains different awareness capabilities enabled according to the system requirements. The stimulus-awareness is the basic awareness capability responsible for triggering adaptations when a violation is detected and selecting an adaptation tactic from the tactics catalogue. Other self-awareness capabilities help in selecting the optimal tactic using their owned information. For instance, time-awareness can provide historical information about the performance of a tactic under similar conditions. The goal-awareness is capable to detect possible violations within a threshold. The meta-self-awareness decides on which awareness level the architecture would operate. The selected tactic is executed by the *Adaptation Executor* of the self-expression component. The *Architecture Evaluator* evaluates the new state after executing the tactic, where such information is passed to the time-awareness component.

D.2.3 Modelling QoS Goals and Adaptation Tactics

Goals are the main objective or trigger for self-adaptation. *QoS Goals* represent the quality of service targets required to be fulfilled. Whenever violated, an adaptation should take place to achieve the quality goals. For each QoS Goal, a set of possible adaptation tactics is implemented in the tactics catalogue. Also, adaptation rules are defined as *if-condition-then-action* rules, where the conditions are quality requirements and the actions are response tactics.

In *SAd-CloudSim*, the *Goals Model* combines these quality targets. For self-adaptive architectures, goals are specified as static values for quality attributes required to be fulfilled. These values are checked during runtime against the actual quality measured data, and adaptations are triggered whenever a violation is detected.

Employing self-awareness capabilities requires a more sophisticated goals model, where *Runtime Goals* can be dynamically settled at runtime or specified for different users. The *Runtime Goals Model* keeps historical information about the satisfaction of goals and the performance of adaptation tactics to be used for better informed decision when choosing the optimal tactic and for future learning using the time-awareness capability.

D.3 Design and Implementation

In this section, we provide details related to the classes and implementation of *SAd-CloudSim* and *SAw-CloudSim*.

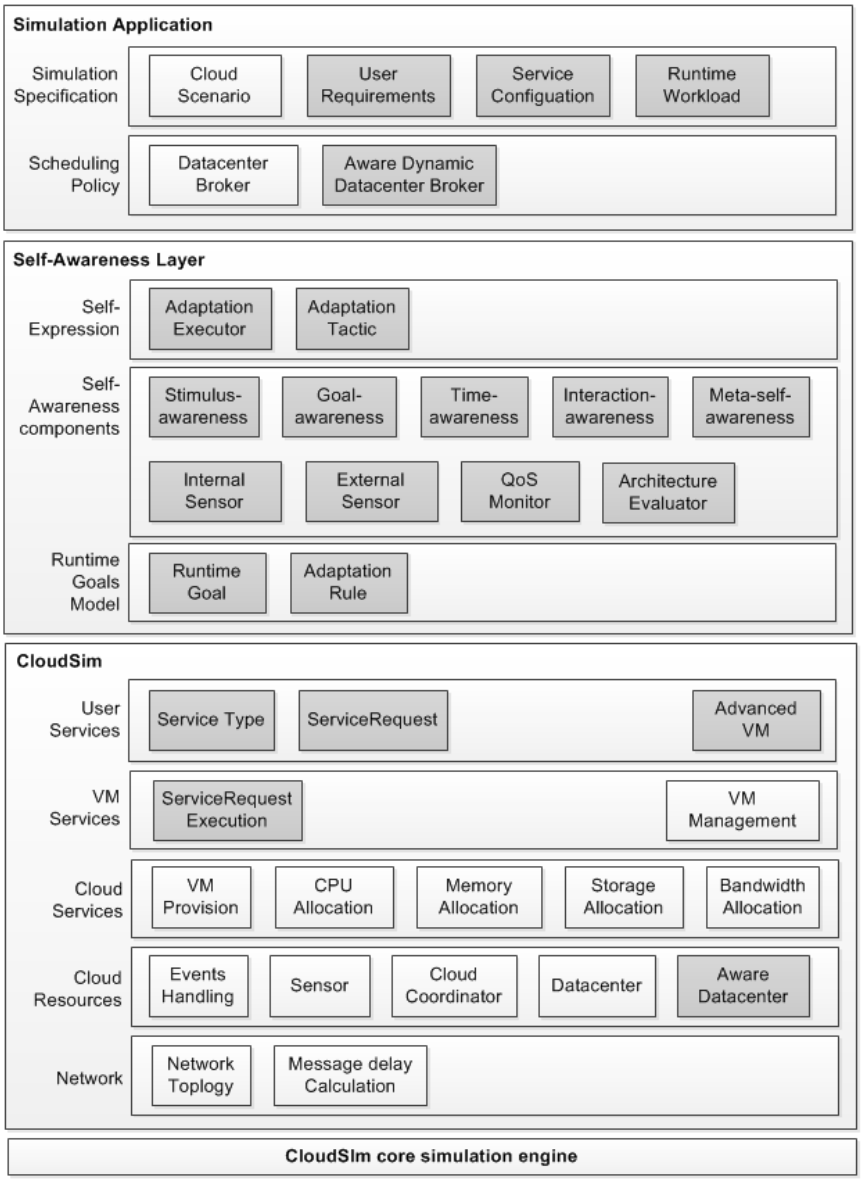

Figure D.3: *SAw-CloudSim* Architecture

D.3.1 Extensions to CloudSim Core

We have extended some core classes of CloudSim by adding necessary quality and power (energy) metrics, namely AdaptiveDatacenter, AwareDatacenter, AdvancedHost and AdvancedVM. The DatacenterBroker —responsible for workload distribution and resources provisioning —is also extended by queueing models necessary for adaptation and awareness capabilities. A *RuntimeWorkload* is added to allow conducting experiments for consecutive time intervals, and user requirements are added to configure QoS requirements.

We use the *Service Type* class to model an SaaS service offered by the cloud provider. A service type is configured by the computational resources it requires (MIPS). A *Service Request* is used to model a request made by an end-user for a specific service type. This allows modelling dynamic workloads by multiple end-users for a variety of services.

D.3.2 Self-Adaptation Simulation

The *Self-Adaptation* package encapsulates the components necessary for modelling and simulating a self-adaptive architecture. Our initial implementation includes the basic functionalities of these components. Figure D.4 depicts the flow of the simulation process in case of self-adaptation. These components could be further extended with more sophisticated implementations, such as MAPE-K. This package is composed of the following classes:

- *Self-Adaptive Architecture* class is the main class responsible for instantiating and managing the adaptation components, i.e. monitor, detector, adaptation engine, adaptation executor. Once instantiated, it loads the goals model from the user configuration `xml` file. It is also responsible for keeping track of the adaptation history and overhead for performance evaluation. This class is designed using the singleton pattern.

- *Goals Model* class is the list of goals objects loaded from a configuration file. Each *Goal* object contains the list of attributes, that are: goal id, name, constraint value, metric (e.g. ms), objective (if the objective is to minimise or maximise the attribute), weight, a boolean indicator whether it is violated. The constraint value is the requirement to be achieved.

- *Monitor* class runs as a thread in the background. It contains methods sensing, measuring and collecting actual data of the QoS parameters of the executed requests, e.g response time, throughput, energy consumption. The monitor is configured with the monitoring frequency to run and collect data. After cleaning the queue of the previous monitoring cycle, the collected data is put in the queue to be sent to the detector.

- *Detector* class contains a method triggered to run after receiving data from the monitor. It checks the runtime values of the quality metrics against the Goals Model. If a violation is detected, adaptation is triggered.

- *Adaptation Engine* class is responsible for selecting the optimal adaptation action after receiving the adaptation trigger. The adaptation action is selected from the Adaptation Tactics Catalogue according to the adaptation rules. Adaptations rules list object is set in this class using `xml` configuration file that contains the quality attributes, their associated tactics and their order of execution. The selection is based on a simple rule-based algorithm and could be further extended with knowledge-based models.

- *Adaptation Tactics Catalogue* class contains a list of adaptation tactics, loaded from `xml` configuration file. Examples of tactics could be increasing VMs capacity, the number of VMs or PMs for better response time and consolidating VMs for less energy consumption. Each *Adaptation Tactic* object contains the attributes of a tactic, that are: id, description, affected object (e.g., host, VM), change (increase or decrease) and the minimum and maximum limits (e.g. minimum one running host and maximum capacity of the datacenter).

- *Adaptation Rule* class links quality attributes with their adaptation tactics. It contains the details of an adaptation rule, that are: id, description, quality attribute, adaptation tactic and its priority in execution.
- *Adaptation Executor* class performs the actual execution of the selected adaptation action on the relevant object, i.e. VM instances, list of VMs, list of PMs.

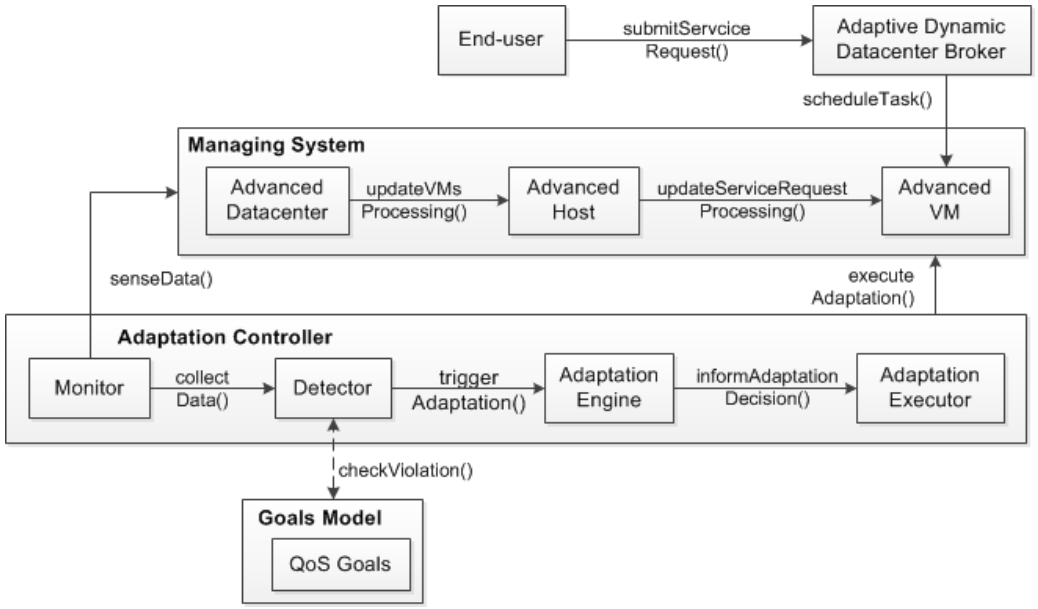

Figure D.4: Self-Adaptation Simulation Process

D.3.3 Self-Awareness Simulation

Figure D.5 depicts the flow of the simulation process in case of employing self-awareness and self-expression capabilities. The *Self-Awareness* package encapsulates the components necessary for modelling and simulating a self-aware and self-expressive architecture, as follows.

- *Self-aware Architecture* class is the main class responsible for instantiating and managing the main components, i.e. QoS Monitoring, Self-Awareness and Self-Expression components. Once instantiated, it loads the runtime goals model from the user configuration `xml` file. It is also responsible for keeping track of the adaptation history and overhead for performance evaluation. This class is designed using the singleton pattern.
- *Runtime Goals Model* class contains the list of runtime goals objects loaded from the configuration file. Each *Runtime Goal* object is inherited from the Goals Model class and contains a new set of attributes: user id (to mark the runtime goals of different users) and violation threshold (to reflect the threshold to take pro-active adaptations). The Runtime Goal Model contains history records to keep track of the

goals fulfilment (i.e. time instance, average violation value, tactic executed, average value after adaptation).

- *QoS Monitoring* component is composed of sensors for different quality requirements, QoS Monitor and Architecture Evaluator, as described below:
  - *Internal Sensor* and *External Sensor* classes contain methods running in the background for continuously sensing data about QoS parameters. The internal sensors are for sensing the actual quality parameters in the self-aware node. The external sensors are required for interaction-awareness for sensing data from the other nodes with which the node is interacting.
  - *QoS Monitor* class contains another background method is for correlating data received from the sensors. Such data is sent to the self-awareness component to take necessary actions. The basic version of the QoS Monitor constantly sends data to the self-awareness component. More sensitive monitors can vary the interval of data correlation according to sensed data.
  - *Architecture Evaluator* class continuously evaluates the response after executing the adaptation action and feeds the different levels of awareness for further actions if needed.
- *Self-Awareness* component encompasses the different levels of self-awareness. These levels that could be enabled as per the relevance to the system requirements using a configuration file. Each self-awareness component is designed using the *Self-Awareness* abstract class to implement the `act` method. Self-awareness components are:
  - *Stimulus-awareness* class embeds rules for selecting and composing optimal adaptation actions or tactics, by defining "if-condition-then-action" rules where the conditions are quality parameters subject of violation and actions are response tactics. The adaptation is triggered when violations are detected.
  - *Goal-awareness* class contains the `act` method operating as a "goal-oriented adaptation engine" that uses knowledge about runtime goals to make decisions about the tactic selection in line with the system's goals. This version of the adaptation engine is more sensitive towards violations and can take pro-active actions before violations.
  - *Time-awareness* class contains the adaptation trainer method that uses historical data about tactics responses under different runtime conditions to improve the quality of adaptation. Implementing machine learning techniques is useful for realising time-awareness.
  - *Interaction-awareness* class contains the interaction-oriented adaptation engine that should contribute to the selection of the tactic according to the runtime environmental conditions of other nodes. This, currently implemented as an abstract, could be implemented in cases of distributed clouds or cloud federations. [1]

[1] currently beyond the scope of this work

– *Meta-self-awareness* class contains the adaptation manager method to reason about the benefits and costs of maintaining a certain level of awareness (and degree of complexity with which it exercises this level), as well as the benefits and costs of selecting a tactic based on a certain level of awareness. This can also dynamically select a particular adaptation out of a set of possibilities for realising one or more levels, in order to manage trade-offs between different QoS attributes. Trade-offs management algorithms could be implemented here. A more sophisticated `act` method can adapt the way in which the level(s) of self-awareness are realised, e.g. by changing algorithms realising the level(s), thus changing the degree of complexity of realisation of the level(s).

- *Self-expression* component is responsible for the execution of the adaptation decision made by the self-awareness component. It is composed of the *Adaptation Executor* responsible for managing the process of adaptation execution during runtime. In more details, it makes necessary instructions about the composition and instantiation of the components required for the adaptation decision. As an example, in the case of VMs consolidation, it decides which VMs should be consolidated, where these VMs should be placed, which PMs should be switched off. Then, it performs the actual instantiation of the tactic components during runtime, such as creating new VMs or switching off PMs.

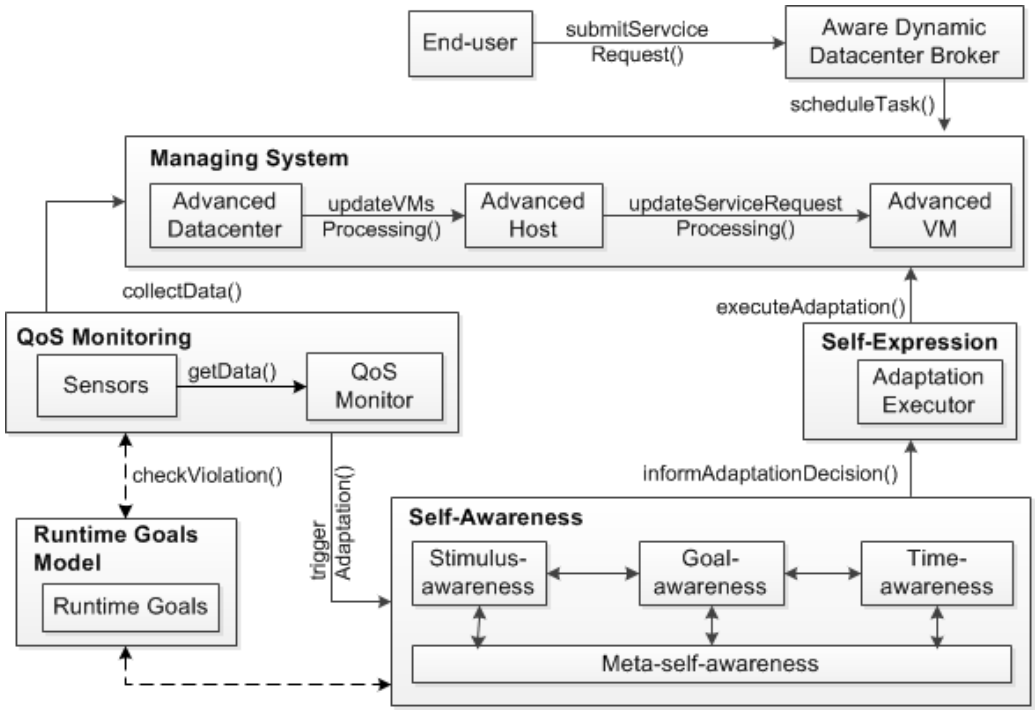

Figure D.5: Self-Awareness Simulation Process

D.4 Experimental Validation and Evaluation

This section aims to examine the capability of the proposed framework to instantiate different architectures of cloud nodes, validate the self-adaptation and self-awareness components, and assess associated overhead. In the course of the validation process, we do not contribute with new scheduling policies. We use current scheduling policies to test the new simulation toolkits.

D.4.1 Experiments Setup

We instantiated the architectures of a self-adaptive and self-aware cloud nodes using the proposed simulation environments. The architectures are configured to dynamically perform architecture-based adaptation to achieve the following QoS challenging objectives: (i) quality requirements, (ii) environmental restrictions, and (iii) economic constraints. Table D.1 lists details of the QoS attributes. With respect to the quality requirements, we consider performance (measured by response time from the time the user submits the request until the cloud submits the response back to the user in milliseconds). For the environmental aspect, we use the greenability property [157] [433] measured by energy consumption in kWh. For the economic constraints, we define the operational cost by the cost of computational resources (CPUs, memory, storage and bandwidth). The weights are hypothetical for testing purpose.

Table D.1: Settings of QoS Attributes

Attribute	**Weight**	**Metric**	**Objective**
Response time	0.50	ms	25
Energy Consumption	0.20	kWh	25
Operational cost	0.20	$	50

We defined the catalogue of architectural tactics (described in section 5.4.2.3) to fulfil the quality attributes subject to consideration. Adaptation rules (listed in Table 5.3) are, then, embedded in the adaptation engine and the stimulus-awareness component. We embedded the tactics catalogue in the self-adaptive and self-aware architectures and the relationships are made implicit within the interaction between different components.

The testbed configurations and benchmarks used are described in section 5.4.2.2 and 5.5.1.1 respectively. The initial deployment of the experiments is shown in Table D.2. When running self-adaptive, stimulus-aware and goal-aware architectures, the initial deployment is 10 hosts running 15 VMs. Initially, the VMs are allocated according to the resource requirements of the VM types. However, VMs utilise fewer resources according to the workload data during runtime, creating opportunities for dynamic consolidation. For the non-adaptive architecture, the deployment is 70 hosts running 210 VMs (the maximum number used by the self-adaptive architecture) to allow processing the maximum number of requests during peak load.

Table D.2: Initial Deployments of the Experiments

Configuration	
No. of hosts	non-adaptive: 70 adaptive: 10
No. of VMs	non-adaptive: 210 x m4.xlarge adaptive: 5 x m4.large, 5 x m4.xlarge, 5 x m4.2xlarge

The experiments were run on a 2.9 GHz Intel Core i5 16 GB RAM computer. To examine the accuracy of simulation results, we examined quality attributes at each time interval of 864 seconds in the cases of self-adaptive, stimulus-aware, goal-aware and non-adaptive architectures, i.e. we run the entire workload for each service type and measured the quality attributes.

D.4.2 Validation Results

To validate the simulation environment, we compare the average response time, energy consumption and operational cost of all architectures during the experiment time intervals. Figure D.6, D.7 and D.8 show the results of service type 2 (the service type with the most processing requirements) of the quality attributes respectively. As the non-adaptive architecture was running on a static configuration (the same number of hosts and VMs required to handle the highest load), the results of response time are the same for all time intervals. The adaptive and aware architectures have similar values like the non-adaptive architecture during off-peak intervals, where they were able to handle the workload with fewer resources. During peak intervals, response time started to fluctuate, where adaptations took place to meet the goal. It is noticed that the self-adaptive architecture was not able to achieve good response time once the workload started to increase compared to the stimulus- and goal-aware, as the adaptations of the former are reactive. As expected, the operational cost and energy consumption of the latter architectures are lower than the non-adaptive architecture, with a maximum equal to the values of the non-adaptive architecture. These are the expected behaviours for all architectures considering the testbed configurations. Hence, the results reflected that architectures components are correctly implemented. Obviously, the results showed the benefits of adaptivity and awareness with respect to achieve the required performance, while saving operational cost and energy consumption.

D.4.3 Experiments Results

Considering the experiments total results, we report the average results of the whole experiment (for 30 runs) for each service type in case of each architecture in Table D.3. The non-adaptive architecture has a fixed value for all attributes, due to the static configuration. The average response time of all requests for each service type is much better achieved

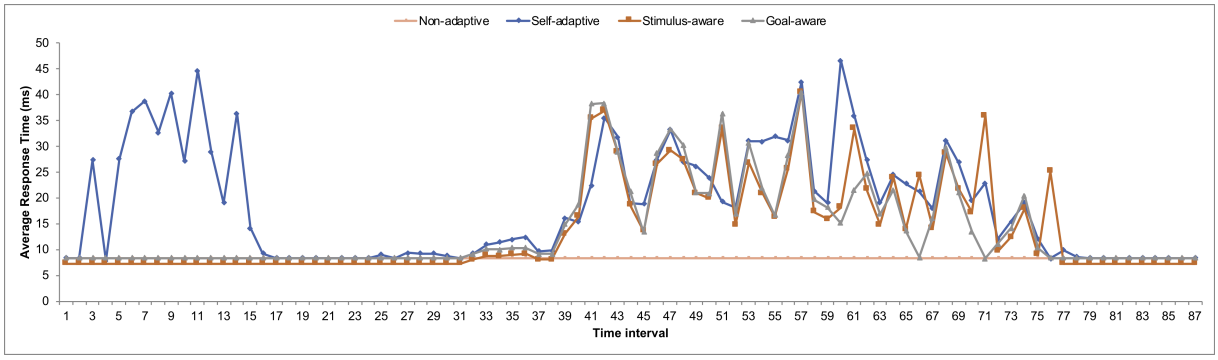

Figure D.6: Average Response Time of Service Type 2 during Time Intervals

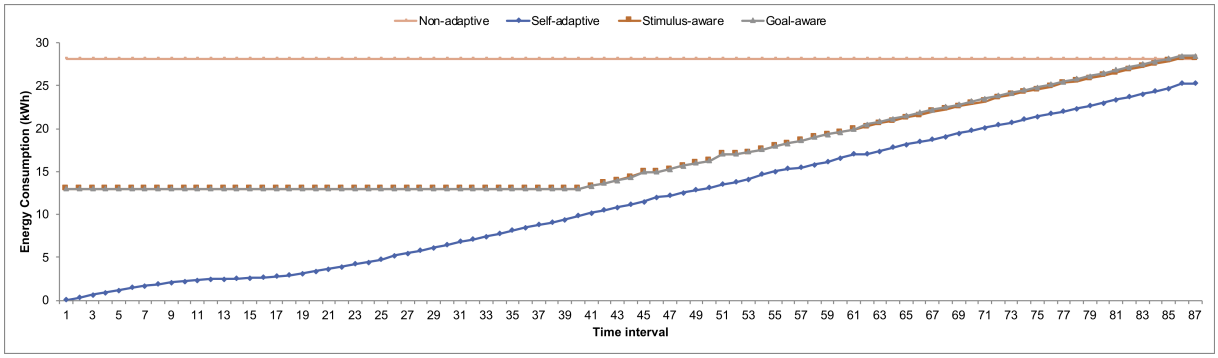

Figure D.7: Average Energy Consumption of Service Type 2 during Time Intervals

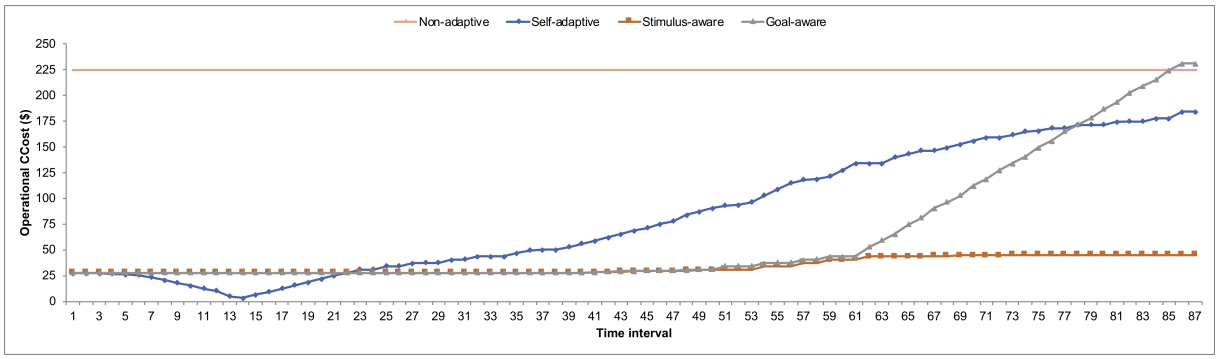

Figure D.8: Average Operational Cost of Service Type 2 during Time Intervals

by the goal-aware architecture due to proactive adaptations, followed by stimulus-aware and self-adaptive architecture (average 20.02, 20.53, 62.85 ms respectively). While achieving better performance, energy consumption (calculated based on the number of running hosts) and operational cost (calculated based on the number of running VMs) were found less on average than non-adaptive. For instance, average energy consumption is 17.42, 17.32, 11.37 kWh versus 28.14 kWh for the non-adaptive architecture, due to consolidation performed during off-peak periods and scaling during peak load only. Operational cost is found less in the case of stimulus-aware architecture (31.88 $), followed by the goal-aware (56.26 $) and self-adaptive (79.57 $) compared to non-adaptive (224.34 $). As the stimulus- and goal-aware architectures were running nearly the same number of hosts, their energy consumption was close. But, each was running a different number of

VMs, which caused the difference in operational cost. The goal-aware architecture used a higher number of VMs in pro-active adaptations.

Table D.3: Experiments Average Results

Quality Attributes	**S#**	**Architecture**			
		Non-adaptive	Self-adaptive	Stimulus-aware	Goal-aware
Response Time (ms)	1	4.17	73.73	16.54	16.00
	2	8.33	63.49	23.01	22.92
	3	5.00	58.41	19.18	18.56
	4	6.25	58.90	21.69	21.10
	5	7.08	59.74	22.24	21.54
	avg.	6.17	62.85	20.53	20.02
Energy consumption (kWh)	1	28.14	10.42	17.42	17.56
	2	28.14	11.61	17.30	17.37
	3	28.14	11.61	17.35	17.41
	4	28.14	11.61	17.29	17.39
	5	28.14	11.61	17.27	17.36
	avg.	28.14	11.37	17.32	17.42
Operational cost ($)	1	224.34	64.54	29.36	52.40
	2	224.34	84.41	34.08	64.47
	3	224.34	82.61	30.43	53.68
	4	224.34	82.61	31.65	56.29
	avg.	224.34	79.57	31.88	56.26

D.4.4 Performance Evaluation

In order to evaluate the performance of self-adaptive and self-aware architectures, we observe the processing of all service requests and compared the percentage of response time violations for different service types, as shown in Figure D.9. As expected, the goal-aware architecture has the less violation percentage (e.g. 24.40% in the case of service type 2). This is due to the proactive adaptation taken prior to violations. While the self-adaptive had better performance than stimulus-aware (e.g. 26.44% versus 28.86% in the case of service 2), the operational cost was remarkably higher in the former case starting from the peak time.

D.4.5 Evaluation of Adaptation Overhead

We evaluate the adaptation overhead by calculating the total time spent by the architecture in monitoring quality attributes, detecting violations, making and executing adaptation decisions. Figure D.10 shows the overhead of each service type and their average. As goal-aware architecture is performing pro-active adaptations, its overhead is the highest (251.62 sec on average). Stimulus-aware is close to goal-aware due to the intelligent

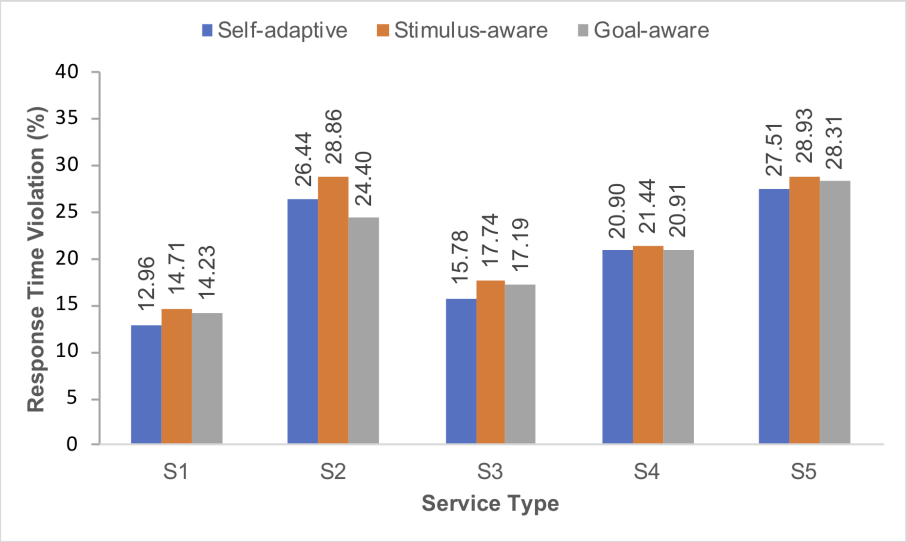

Figure D.9: Average Response Time Violations

reactions (239.47 sec). The overhead of self-adaptive is lower (164.90 sec) due to reactive adaptations, which obviously resulted in lower performance.

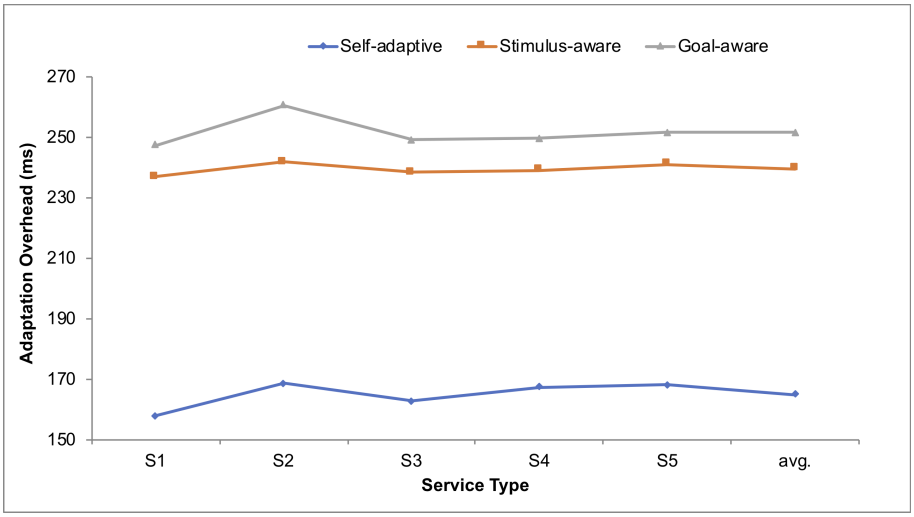

Figure D.10: Average Adaptation Overhead

D.5 Related Work

In the context of self-adaptive software systems, Abuseta et al. [610] proposed a simulation environment for testing self-adaptive systems designed around the feedback control loop proposed by IBM architecture blueprint. A review for the state-of-the-art related to self-awareness in software engineering [45] has confirmed the lack of simulation tools for designing and evaluating such systems, with the exception of the work of [611]. This work proposed a simulation environment for systems with self-aware and self-expressive

capabilities, focusing on hardware aspects and precise process chronology execution. The simulation environment suits industrial relevant system sizes of the avionic and spaceflight industry.

With respect to cloud computing, there have been some notable proposals for simulation environments. An early survey has enlisted simulation approaches used for research in cloud computing [612]. Examples include CloudSim [36] [5] a modular and extensible open-source simulator, able to model very large scale clouds, GreenCloud [613] a packet-level simulator of energy-aware cloud data centers, MDCSim [614] simulates multi-tier data centres in detail, and iCanCloud [615] [616] simulates cloud infrastructures flexibility and scalability. Other tools focused on simulating specific issues, such as power consumption and scientific workflows [612].

CloudSim has been widely adopted and used in many further extensions modelling and simulating cloud-related problems, due to its modular architecture. Examples include visually modelling and analysing cloud environments and applications (CloudAnalyst) [617], modelling parallel applications (NetworkCloudSim) [609], simulating scientific workflows (WorkflowSim) [618], concurrent and distributed cloud (Cloud2Sim) [608], adaptive scaling cloud and MapReduce simulations (Cloud^{2}Sim) [619], simulating heterogeneity in computational clouds (DynamicCloudSim) [620], and simulating containers in cloud data centres (ContainerCloudSim) [621].

Despite the influx of research in self-adaptivity and cloud computing, as well as the various simulations environments proposed so far, there is a general lack, to the best of our knowledge, of modelling and simulation environments for self-adaptive and self-aware cloud architectures.

Appendix E

Queuing-based Model for Evaluating Runtime Stability

In this appendix, we present a queuing theoretic-based model for evaluating stability during runtime. A self-adaptive software system is dynamic and exhibits probabilistic behaviour during runtime. Such behaviour is mainly due to the uncertain fluctuation of the workload at runtime, the constraints on available resources and changes in the environment. Behaviour can also be affected by prior decisions and adaptation actions.

Given the runtime dynamics and the probabilistic behaviours of such systems, a Markov-based analytical modelling can provide a generic and scalable model for this probabilistic behaviour. Based on multiple parallel dynamic queues, the model can capture instance-related information at a finer-grained level of tactics' configurations, given the heterogeneity of the environment. The model can, then, measure and predict quality attributes for a scenario of interest. Such measurements and predictions, in conjunction with the goal-awareness capability, can assist in choosing the optimal tactics and their configurations to achieve behavioural stability.

E.1 System Model

Assume that a software system is running on a computing node using m hosts (Physical Machines PMs). A PM_i, where $i = \{1, ..., m\}$ runs n_i VMs sharing computational resources. The number of running VMs varies from one PM to another according to its computational capacity. Service requests are received and processed on the infrastructure, where the workload tends to vary in the number of incoming requests, the length of each request, and quality requirements according to the end-user SLA.

We assume the total incoming workload λ will be divided among the m PMs resulting $\{\lambda_1, \lambda_2, \lambda_i, ..., \lambda_m\}$. Several algorithms have been proposed to manage the jobs placement in PMs and VMs [622] [623]. Though we follow a simple approach for requests placement, the same principle can apply to other placement mechanisms. The distribution of workload, in our case, is based on either the PM computational capacity in case PMs computational capacity are different or equally on all PMs based on their availability.

Each PM, by its turn, will distribute its workload share on its n running VMs. The workload is distributed on VMs level either based on VM computational capacity in case

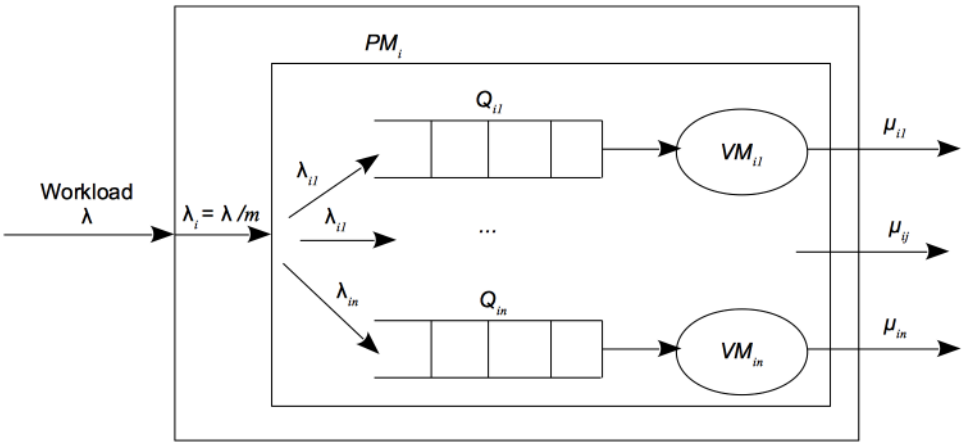

Figure E.1: Dynamic Workload Handling

the incoming request is constrained by certain computational requirements, or equally in case of no constraints. The workload is denoted by λ_{ij}, where i indicates the PM, j indicates the VM, and $j = \{1, ..., n_i\}$. For a VM_{ij}, an $m/m/1$ queue will be formed for the incoming requests to be processed, where the incoming rate of requests constitutes a Poisson process of rate $\lambda_{i/n}$ (assuming equal workload distributed on all VMs), and the service process is Markovian exponentially distributed, with parameter μ_{ij} and mean $1/\mu_{ij}$ that is handled by that VM. Thus, the total service handled by the self-aware node is $\sum_{i=1}^{m}\sum_{j=1}^{n}\mu_{ij}$. The handling of the workload in a self-aware node is illustrated in Figure E.1.

Unlike most of the prior models that have employed only single queues, we employ multiple parallel dynamic queues, where the queuing can discipline the way we analyse the workload in relation to heterogeneous environments with varying configurations of PMs, VMs, and their computational capacities. The model also features scalability into the analysis, as well as helps in tracking and predicting the behaviour at a given time instance.

For VM_{ij}, the formed queue of incoming requests can be described as a continuous time Markov chain with transition rate matrix

$$Q_{ij} = \begin{pmatrix} -\lambda_{i/n} & \lambda_{i/n} & & & \\ \mu_{ij} & -(\mu_{ij}+\lambda_{i/n}) & \lambda_{i/n} & & \\ & \mu_{ij} & -(\mu_{ij}+\lambda_{i/n}) & \lambda_{i/n} & \\ & & & & \ddots \end{pmatrix}$$

on the state space S_{ij} $\{0, 1, 2, 3, ...\}$, and the rate from state k to the state $k+1$ is denoted by $q_{k,k+1}$. Thus, $q_0 = \lambda_{i/n}$, $q_{00} = -\lambda_{i/n}$, and $q_{01} = \lambda_{i/n}$. In general, we must have

$$q_{k,k+1} \geq 0 \text{ for all } k \neq k+1 \in S_{ij}$$

where $q_{k,k+1}$ denotes the $k, k+1^{th}$ diagonal element in the Q_{ij} matrix.

Let X_t denote the number of requests in the VM_{ij} queue at time t. If $X_t = 0$, then

the next event has to be the arrival of a new request, and the time of its arrival is exponential $\lambda_{i/n}$. At run-time, the next event could be either the arrival of a new request or the departure of the request currently being processed. Thus, the time to the next event is exponentially distributed with the parameter $\lambda_{i/n} + \mu_{ij}$. Hence, $q_k = \lambda_{i/n} + \mu_{ij}$, $q_{k,k+1} = \lambda_{i/n}$, and $q_{k,k-1} = \mu_{ij}$. So, the probability of the arrival of a new request is $\lambda_{i/n} / \left(\lambda_{i/n} + \mu_{ij}\right)$, and the complementary probability $\mu_{ij} / \left(\lambda_{i/n} + \mu_{ij}\right)$ is the probability of the departure of the request currently being processed.

Having fully specified the transition rate matrix Q_{ij}, $\{X_t, t \geq 0\}$ is, then, a Markov process with the following transition rates:

$$q_{k,k+1} = \lambda_{i/n} \ , \ q_{k,k-1} = \mu_{ij} \ , \ qk, k = -\left(\lambda_{i/n} + \mu_{ij}\right) \quad \text{for all } k \geq 1$$

with an invariant distribution π, where

$$\pi_k q_{k,k+1} = \pi_{k+1} q_{k+1,k} \text{ for all } k, k+1 \tag{E.1}$$

along with the normalisation condition

$$\sum_{k=0}^{\infty} \pi_k = 1 \tag{E.2}$$

We obtain from (E.1) that

$$\pi_k = \left(\lambda_{i/n} / \mu_{ij}\right) \pi_{k-1} \text{ for all } k \geq 1$$

Denoting $\left(\lambda_{i/n} / \mu_{ij}\right)$ by ρ_{ij} , we get

$$\pi_k = \rho_{ij}^k \pi_0 \tag{E.3}$$

Substituting in (E.2), we get

$$\pi_k = \left(1 - \rho_{ij}\right) \rho_{ij}^k \ , \ k = 0, 1, 2, ..., \text{ if } \rho_{ij} < 1 \tag{E.4}$$

which represents the invariant distribution of the Markov process transition rate of our imposed problem.

E.2 Quality Model

The Markov-based analytical model allows estimating the quality of service. Given the expected workload λ, the number of PMs m, VMs, and the capacity of both of them, the model approximates different quality attributes; such as response time (R), mean queue (W), throughput (T), utilisation (ρ), cost (C) and energy consumption (E).

For the VM_{ij}, given the incoming rate of requests $\lambda_{i/n}$ and the mean service time

$1/\mu_{ij}$, the invariant queue length distribution computed in (E.4) gives us

$$P(N = n) = (1 - \rho)\rho^n,\ n = 0, 1, 2, 3, ...$$

In particular, $P(N = 0) = 1 - \rho$, that is the probability that the queue is empty is steady state. Hence, the utilisation of the VM_{ij} should be:

$$\rho_{ij} = \lambda_{i/n}/\mu_{ij}$$

Therefore, the probability for VM_{ij} to be idle can be expressed by π_0 from (E.3) as:

$$\pi_0 = 1 - \rho_{ij} = 1 - (\lambda_{i/n}/\mu_{ij})$$

By applying Little's law $E(S) = (1/\mu)/(1 - \rho)$, the following performance metrics could be deduced:
The mean response time for VM_{ij} is estimated by:

$$R_{ij} = 1/(\mu_{ij}(1 - \rho_{ij})) = 1/(\mu_{ij}\rho_{ij})$$

The mean queue length is:

$$W_{ij} = \rho_{ij}^2/(1 - \rho_{ij})$$

The mean throughput is basically the departure rate; i.e. the rate at which the requests finish being processed successfully at the VM; that is:

$$T_{ij} = \lambda_{i/n}\pi_k/\sum_{k=0}^{\infty}\lambda_{i/n}\pi_k$$

Having performance metrics of each VM independently, all performance metrics for a given PM could be deduced, as well as for the self-adaptive computing node. The mean response time for PM_i is the mean response time for the n_i VMs running on that PM. Also, the mean utilisation and the throughput can be calculated as the sum of the related measures of the n_i VMs.

On the node level, same metrics could also be calculated as the sum of related metrics for the m PMs operating on the node. Operational cost could also be calculated among the node, that is the cost of processing the incoming workload:

$$C = \sum_{i=1}^{m}\sum_{j=1}^{n} Cost(CPU)_{ij} + Cost(memory)_{ij}$$

And, the total power consumption of all running PMs, given the varying number of VMs and their allocated CPU threads, would be:

$$C = \sum_{i=1}^{m} E_i$$

As an architectural tactic represents codified knowledge about the relationship between architectural decisions and quality attributes [624], our analytical model can accommodate

the impact of a diverse range of tactics on the stability of these quality attributes, as follows.

- Tactics related to PMs, such as horizontal scaling and consolidation, are reflected in our model by varying the value of m PMs. That is, scaling with a certain number of PMs will be reflected in our model when dividing the incoming workload λ on more PMs; i.e. $m + 1$. This would influence the stability of performance (response time) and greenability (energy consumption).

- Tactics related to VMs, such as vertical scaling and consolidation, are reflected in our model by increasing or decreasing the total value of n VMs. This influences the average latency of processing the incoming requests.

- Tactics related to computational capacity; i.e., CPU threads of a specific VM_{ij}; are reflected in the increase or decrease of the corresponding service rate μ_{ij}, and hence influence the throughput. Also, the utilisation of VMs, determined by our model, allows consolidating the less utilised VMs (e.g. x VMs are less than 10% utilised) and re-checking the performance metrics given the new number of VMs $(n - x)$.

Aiming to stabilise a certain quality attribute, the impact of related tactic could be predicted under different configurations of the tactic, in order to select the optimal configuration. Unlike prior related work, which considered a case of homogeneity, we consider the heterogeneity of environment in PMs, VMs, and their computational capacity. The proposed model is capable to model the sensitivity of quality parameters behaviour with different scenarios varying the number of PMs, computational capacities of PMs, number of VMs, allocated CPU threads and requests constraints. Besides, our model allows measuring the cost and energy consumption of the self-adaptive computing node under these different scenarios. Also, information from self-awareness capabilities is employed in our model. More specifically, we rely on the goal-awareness level in informing the adaptation process to select the adaptation tactic that converges towards the adaptation goal. This influences the deduced performance metrics, and consequently leas to the choice of the optimal tactics.

Bibliography

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

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

[3] 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," School of Computer Science, University of Birmingham, Technical Report, 2014.

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

[5] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[6] 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.

[7] 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.

[8] 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.

[9] A. Mahdy and M. E. Fayad, "A software stability model pattern," in *9th Conference on Pattern Language of Programs (PLoP02)*, 2002, Conference Proceedings.

[10] 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.

[11] J. Wang and M. N. Huhns, "Using simulations to assess the stability and capacity of cloud computing systems," in *48th Annual Southeast Regional Conference*. ACM, 2010, Conference Proceedings, pp. 1–6.

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

[13] 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.

[14] M. Alenezi and F. Khellah, "Architectural stability evolution in open-source systems," in *International Conference on Engineering & MIS*. ACM, 2015, Conference Proceedings, pp. 1–5.

[15] 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.

[16] F. Dantas and A. Garcia, "Software reuse versus stability: Evaluating advanced programming techniques," in *Brazilian Symposium on Software Engineering*, 2010, Conference Proceedings, pp. 40–49.

[17] D. Garlan, "Software architecture: A roadmap," in *Conference on The Future of Software Engineering*, 2000, pp. 91–101.

[18] D. Garlan, "Software architecture: a travelogue," in *International Conference on Future of Software Engineering*. ACM, 2014, pp. 29–39.

[19] R. Bahsoon, W. Emmerich, and J. Macke, "Using real options to select stable middleware-induced software architectures," *IEE Proceedings - Software*, vol. 152, no. 4, pp. 167–186, 2005.

[20] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.

[21] 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.

[22] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, 2009, pp. 1–26.

[23] 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.

[24] 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.

[25] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas, "A framework for evaluating quality-driven self-adaptive software systems," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2011, Conference Proceedings, pp. 80–89.

[26] 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: Elsevier (Morgan Kaufmann), 2017, pp. 17–43.

[27] 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.

[28] D. Weyns and T. Ahmad, *Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7957, book section 22, pp. 249–265.

[29] 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.

[30] 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.

[31] 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.

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

[33] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[34] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed method approaches*, 2nd ed. Thousand Oaks, California London: Sage, 2003.

[35] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of Cloud Computing," *ACM Communications*, vol. 53, no. 4, pp. 50–58, 2010.

[36] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities," in *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2009, pp. 1–11.

[37] 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.

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

[39] R. Bahsoon and W. Emmerich, *Architectural Stability*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5872, book section 43, pp. 304–315.

[40] M. Salama, "Stability of self-adaptive software architectures," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Doctoral Symposium*, 2015, pp. 886–889.

[41] M. Salama and R. Bahsoon, "Quality-driven architectural patterns for self-aware cloud-based software," in *IEEE 8th International Conference on Cloud Computing (CLOUD), Applications Track (acceptance rate 14%)*, 2015, pp. 844–851.

[42] M. Salama and R. Bahsoon, "A taxonomy for architectural stability," in *31st ACM/SIGAPP Symposium on Applied Computing (SAC), Software Architecture: Theory, Technology, and Applications Track (SATTA)*, 2016, pp. 1354–1357.

[43] M. Salama, R. Bahsoon, and N. Bencomo, "Managing trade-offs in self-adaptive software architectures: A systematic mapping study," 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. 249–297.

[44] 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.

[45] A. Elhabbash, M. Salama, R. Bahsoon, and P. Tino, "Self-awareness in software engineering: A systematic literature review," *(submitted for publication)*, 2017.

[46] M. Salama, R. Bahsoon, and P. Lago, "Stability in software engineering: Survey of the state-of-the-art and research directions," *(submitted for publication)*, 2017.

[47] M. Salama and R. Bahsoon and R. Buyya, "Modelling and simulation environment for self-adaptive and self-aware cloud architectures," *(submitted for publication)*, 2018.

[48] M. Salama and R. Bahsoon and R. Buyya, "A reference architecture and modelling principles for architectural stability based on self-awareness: Case of cloud architectures," *(submitted for publication)*, 2018.

[49] M. Salama and R. Bahsoon and R. Buyya, "Architectural stability reasoning using self-awareness principles: Case of self-adaptive cloud architectures," *(submitted for publication)*, 2018.

[50] M. Salama, R. Bahsoon, and P. Lago, "A framework for evaluating architectural stability," *(submitted for publication)*, 2018.

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

[52] 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.

[53] 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.

[54] 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.

[55] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[56] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2003.

[57] 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.

[58] 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.

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

[60] 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.

[61] B. I. Witt, *Software Architecture and Design : Principles, models, and methods*. New York: New York : Van Nostrand Reinhold, 1994.

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

[63] I. Sommerville, *Software Engineering*. Boston, Mass. London: Boston, Mass. London : Pearson Education, 2011.

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

[65] R. Laddaga, "Self-adaptive software," DARPA BAA, Technical Report 98-12, 1997.

[66] A. C. Meng, *On Evaluating Self-Adaptive Software.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 65–74.

[67] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 639–649.

[68] G. Procaccianti, P. Lago, and G. A. Lewis, "Green architectural tactics for the cloud," in *IEEE/IFIP Conference on Software Architecture (WICSA)*, 2014, pp. 41–44.

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

[70] Z. J. Wang, D. C. Zhan, and X. F. X., "STCIM: a dynamic granularity oriented and stability based component identification method," *SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–14, 2006.

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

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

[73] 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.

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

[75] 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.

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

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

[78] 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.

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

[80] L. D. Berkovitz, *Optimal control theory.* New York: Springer-Verlag, 1974.

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

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

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

[84] E. W. Dijkstra, "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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

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

[93] 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.

[94] 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.

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

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

[97] 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.

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

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

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

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] D. A. Tamburri, P. Lago, and H. V. Vliet, "Uncovering latent social communities in software development," *IEEE Software*, vol. 30, no. 1, pp. 29–36, 2013.

[110] 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.

[111] N. L. Soong, "A program stability measure," in *Annual Conference (ACM)*. ACM, 1977, Conference Proceedings, pp. 163–173.

[112] S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545–552, 1980.

[113] S. S. Yau and S. C. Chang, "Estimating logical stability in software maintenance," in *IEEE Computer Society's International Computer Software & Applications Conference*, 1984, Conference Proceedings, pp. 109–119.

[114] S. S. Yau and J. S. Collofello, "Design stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, pp. 849–856, 1985.

[115] Software Engineering Standards Committee of the IEEE Computer Society, "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1–40, 1998.

[116] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC 9126-1 - Software engineering – Product quality – Part 1: Quality model," ISO/IEC, Geneva, Switzerland, Report ISO/IEC 9126-1:2001, 2001.

[117] M. Jazayeri, "On architectural stability and evolution," in *7th Ada-Europe International Conference on Reliable Software Technologies*. Springer-Verlag, 2002, Conference Proceedings, pp. 13–23.

[118] D. Grosser, H. A. Sahraoui, and P. Valtchev, "Predicting software stability using case-based reasoning," in *17th IEEE International Conference on Automated Software Engineering (ASE)*, 2002, Conference Proceedings, pp. 295–298.

[119] D. Grosser, H. A. Sahraoui, and P. Valtchev, "An analogy-based approach for predicting design stability of java classes," in *9th International Software Metrics Symposium*, 2003, Conference Proceedings, pp. 252–262.

[120] R. Bahsoon and W. Emmerich, "Evaluating software architectures for stability: A real options approach," in *25th International Conference on Software Engineering, Doctoral Symposium*, 2003, Conference Proceedings.

[121] R. Bahsoon and W. Emmerich, "ArchOptions: a real options-based model for predicting the stability of software architectures," in *5th Workshop on Economics-driven Software Engineering Research (EDSER), co-located with 25th International Conference of Software Engineering (ICSE)*, 2003, Conference Proceedings.

[122] R. Bahsoon and W. Emmerich, "Evaluating software architectures: development, stability, and evolution," in *ACS/IEEE International Conference on Computer Systems and Applications, Book of Abstracts*, 2003, Conference Proceedings, p. 47.

[123] R. Bahsoon and W. Emmerich, "Evaluating architectural stability with real options theory," in *20th IEEE International Conference on Software Maintenance*, 2004, Conference Proceedings, pp. 443–447.

[124] R. Bahsoon and W. Emmerich, "Requirements for evaluating architectural stability," in *IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2006, Conference Proceedings, pp. 1143–1146.

[125] R. Bahsoon and W. Emmerich, "Architectural stability and middleware: An architecture-centric evolution perspective," in *ECOOP Workshop on Architecture-Centric Evolution*, 2006, Conference Proceedings.

[126] M. O. Elish and D. Rine, "Investigation of metrics for object-oriented design logical stability," in *7th European Conference on Software Maintenance and Reengineering*, 2003, Conference Proceedings, pp. 193–200.

[127] S. Bouktif, D. Azar, D. Precup, H. Sahraoui, and B. Kegl, "Improving rule set based software quality prediction: A genetic algorithm-based approach," *Journal of Object Technology*, vol. 3, no. 4, pp. 227–241, 2004.

[128] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *26th International Conference on Software Engineering (ICSE)*, 2004, Conference Proceedings, pp. 282–291.

[129] M. O. Elish and D. Rine, "Indicators of structural stability of object-oriented designs: A case study," in *29th Annual IEEE/NASA Software Engineering Workshop*, 2005, Conference Proceedings, pp. 183–192.

[130] D. Kelly, "A study of design characteristics in evolving software using stability as a criterion," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 315–329, 2006.

[131] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, *On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4609, book section 9, pp. 176–200.

[132] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2007, Conference Proceedings, pp. 30–39.

[133] A. Molesini, "On the quantitative analysis of architecture stability in aspectual decompositions," in *7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, A. F. Garcia, C. v. F. G. Chavez, and T. V. Batista, Eds., vol. 0, 2008, Conference Proceedings, pp. 29–38.

[134] A. Molesini, A. F. Garcia, C. v. F. G. Chavez, and T. V. Batista, "Stability assessment of aspect-oriented software architectures: A quantitative study," *Journal of Systems and Software*, vol. 83, no. 5, pp. 711–722, 2010.

[135] J. P. Correia, Y. Kanellopoulos, and J. Visser, "A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics," in *IEEE International Conference on Software Maintenance*, 2009, Conference Proceedings, pp. 61–70.

[136] L. Yu and S. Ramaswamy, "Measuring the evolutionary stability of software systems: case studies of linux and freebsd," *IET Software*, vol. 3, no. 1, pp. 26–36, 2009.

[137] F. Dantas, "Reuse vs. maintainability: revealing the impact of composition code properties," in *33rd International Conference on Software Engineering*. ACM, 2011, Conference Proceedings, pp. 1082–1085.

[138] H. Cui, J. Simsa, Y. H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, "Parrot: A practical runtime for deterministic, stable, and reliable threads," in *24th ACM Symposium on Operating Systems Principles*. 2522735: ACM, 2013, Conference Proceedings, pp. 388–405.

[139] S. Bouktif, H. Sahraoui, and F. Ahmed, "Predicting stability of open-source software systems using combination of bayesian classifiers," *ACM Transactions on Management Information Systems*, vol. 5, no. 1, pp. 1–26, 2014.

[140] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of gof design patterns on stability: A case study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 781–802, 2015.

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

[142] 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.

[143] 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.

[144] 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.

[145] 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.

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

[147] Standards Committee of the IEEE Reliability Society, "IEEE Recommended Practice on Software Reliability," *IEEE Std 1633-2016 (Revision of IEEE Std 1633-2008)*, pp. 1–261, 2017.

[148] 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.

[149] A. Chomchumpol and T. Senivongse, "Stability measurement model for service-oriented systems," in *9th Malaysian Software Engineering Conference (MySEC)*, 2015, Conference Proceedings, pp. 54–59.

[150] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., 1998.

[151] 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.

[152] W. Jiao, "Measurements for adaptation level and efficiency of adaptive software systems," in *18th International Conference on Engineering of Complex Computer Systems*, 2013, Conference Proceedings, pp. 37–45.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] 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.

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

[159] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar, "Input-output robustness for discrete systems," in *10th ACM International Conference on Embedded Software*. ACM, 2012, Conference Proceedings, pp. 217–226.

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

[161] 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.

[162] 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.

[163] J. M. Conejero, E. Figueiredo, A. Garcia, J. Hernández, and E. Jurado, "On the relationship of concern metrics and requirements maintainability," *Information and Software Technology*, vol. 54, no. 2, pp. 212–238, 2012.

[164] 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.

[165] 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.

[166] 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.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] 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.

[173] 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.

[174] 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.

[175] 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.

[176] 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.

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

[178] 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.

[179] E. Constantinou and I. Stamelos, "Architectural stability and evolution measurement for software reuse," in *30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2015, Conference Proceedings, pp. 1580–1585.

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

[181] A. Sangpuwong and P. Muenchaisri, *Comparison of Stability Models in Incremental Development*. Springer International Publishing, 2014, pp. 322–331.

[182] 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.

[183] L. M. Smith and M. H. Samadzadeh, "Measuring complexity and stability of web programs," *Structured Programming*, vol. 13, no. 1, pp. 35–50, 1992.

[184] S. Black, "Computing ripple effect for software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 4, pp. 263–279, 2001.

[185] S. Black, "Deriving an approximation algorithm for automatic computation of ripple effect measures," *Information and Software Technology*, vol. 50, no. 7–8, pp. 723–736, 2008.

[186] J. Bevan and E. J. Whitehead, "Identification of software instabilities," in *10th Working Conference on Reverse Engineering (WCRE)*, 2003, Conference Proceedings, pp. 134–144.

[187] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[188] M. K. Chawla and I. Chhabra, "Sqmma: Software quality model for maintainability analysis," in *8th Annual ACM India Conference*. ACM, 2015, Conference Proceedings, pp. 9–17.

[189] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, *The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation*. Boston, MA: Springer US, 2008, pp. 237–248.

[190] J. Krinke, "Is cloned code more stable than non-cloned code?" in *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, Conference Proceedings, pp. 57–66.

[191] J. Krinke, "Is cloned code older than non-cloned code?" in *Proceedings of 5th International Workshop on Software Clones*. ACM, 2011, Conference Proceedings, pp. 28–33.

[192] N. Göde and J. Harder, "Clone stability," in *15th European Conference on Software Maintenance and Reengineering*, 2011, Conference Proceedings, pp. 65–74.

[193] J. Harder and N. Göde, "Cloned code: stable code," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1063–1088, 2013.

[194] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *ACM SIGAPP Applied Computing Review*, vol. 12, no. 3, pp. 20–36, 2012.

[195] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative stability of cloned and non-cloned code: an empirical study," in *27th Annual ACM Symposium on Applied Computing*. ACM, 2012, Conference Proceedings, pp. 1227–1234.

[196] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, "Is cloned code really stable?" *Empirical Software Engineering*, 2017.

[197] M. S. Rahman and C. K. Roy, "A change-type based empirical study on the stability of cloned code," in *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, Conference Proceedings, pp. 31–40.

[198] S. Kabinna, W. Shang, C. P. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, Conference Proceedings, pp. 326–337.

[199] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015, Conference Proceedings, pp. 86–89.

[200] W. Li, L. Etzkorn, C. Davis, and J. Talburt, "An empirical study of object-oriented system evolution," *Information and Software Technology*, vol. 42, no. 6, pp. 373–381, 2000.

[201] M. P. Robillard, "Tracking concerns in evolving source code: An empirical study," in *22nd IEEE International Conference on Software Maintenance*, 2006, Conference Proceedings, pp. 479–482.

[202] M. Ortu, G. Destefanis, M. Orru, R. Tonelli, and M. L. Marchesi, "Could micro patterns be used as software stability indicator?" in *IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, 2015, Conference Proceedings, pp. 11–12.

[203] D. Threm, L. Yu, S. Ramaswamy, and S. D. Sudarsan, "Using normalized compression distance to measure the evolutionary stability of software systems," in *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, Conference Proceedings, pp. 112–120.

[204] D. Hou and X. Yao, "Exploring the intent behind API evolution: A case study," in *18th Working Conference on Reverse Engineering*, 2011, Conference Proceedings, pp. 131–140.

[205] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2013, Conference Proceedings, pp. 70–79.

[206] D. Pfahl, A. Al-Emran, and G. Ruhe, *Simulation-based stability analysis for software release plans*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 262–273.

[207] D. Pfahl, A. Al-Emran, and G. Ruhe, "A system dynamics simulation model for analyzing the stability of software release plans," *Software Process: Improvement and Practice*, vol. 12, no. 5, pp. 475–490, 2007.

[208] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. 76, no. 12, pp. 1210–1222, 2011.

[209] Y. Eom and B. Demsky, "Self-stabilizing Java," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 287–298, 2012.

[210] A. Bracciali, P. Mancarella, K. Stathis, and F. Toni, *Engineering Stable Multi-agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 322–334.

[211] H. Cui, J. Wu, C. C. Tsai, and J. Yang, "Stable deterministic multithreading through schedule memoization," in *9th Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 10. UseniX Association, 2010, Conference Proceedings.

[212] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu, "Making parallel programs reliable with stable multithreading," *Communications of the ACM*, vol. 57, no. 3, pp. 58–69, 2014.

[213] 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.

[214] 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.

[215] 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.

[216] 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.

[217] C. Jones, "Strategies for managing requirements creep," *Computer*, vol. 29, no. 6, pp. 92–94, 1996.

[218] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[219] C. Ting, *The Control and Measure of Requirements Stability in Software Project*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 387–394.

[220] 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.

[221] Software Engineering Standards Subcommittee of the Technical Committee on Software Engineering of the IEEE Computer Society, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," *IEEE Std 982.2-1988*, 1988.

[222] S. Anderson and M. Felici, *Controlling Requirements Evolution: An Avionics Case Study*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 361–370.

[223] S. Anderson and M. Felici, *Requirements Evolution From Process to Product Oriented Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 27–41.

[224] T. Nakatani, T. Tsumaki, M. Tsuda, M. Inoki, S. Hori, and K. Katamine, "Requirements maturation analysis by accessibility and stability," in *18th Asia-Pacific Software Engineering Conference*, 2011, Conference Proceedings, pp. 357–364.

[225] T. Nakatani, K. Katamine, M. Tsuda, and T. Tsumaki, "Estimation of the maturation type of requirements from their accessibility and stability," *IEICE Transactions on Information and Systems*, vol. 97, no. 5, pp. 1039–1048, 2014.

[226] D. Bush and A. Finkelsteiin, "Environmental scenarios and requirements stability," in *International Workshop on Principles of Software Evolution*. ACM, 2002, Conference Proceedings, pp. 133–137.

[227] D. Bush and A. Finkelsteiin, "Requirements stability assessment using scenarios," in *11th IEEE International Conference on Requirements Engineering (RE)*. IEEE Computer Society, 2003, Conference Proceedings, p. 23.

[228] M. Brünink, "Autonomous compliance monitoring of non-functional properties," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, Conference Proceedings, pp. 795–798.

[229] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[230] K. Sethi, C. Yuanfang, S. Wong, A. F. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*, 2009, Conference Proceedings, pp. 269–272.

[231] K. Farias, A. Garcia, and C. Lucena, "Effects of stability on model composition effort: An exploratory study," *Software & Systems Modeling*, vol. 13, no. 4, pp. 1473–1494, 2014.

[232] J. Bansiya, *Evaluating Structural and Functional Stability*. John Wiley & Sons, Inc., 1999, pp. 599–616.

[233] J. Bansiya, "Evaluating framework architecture structural stability," *ACM Computing Surveys*, vol. 32, no. 1es, p. 18, 2000.

[234] M. Mattsson and J. Bosch, "Characterizing stability in evolving frameworks," in *Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*, 1999, Conference Proceedings, pp. 118–130.

[235] M. Mattsson and J. Bosch, "Stability assessment of evolving industrial object-oriented frameworks," *Journal of Software Maintenance*, vol. 12, no. 2, pp. 79–102, 2000.

[236] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, 2005.

[237] M. Alshayeb and W. Li, "An empirical study of system design instability metric and design evolution in an agile software process," *Journal of Systems and Software*, vol. 74, no. 3, pp. 269–274, 2005.

[238] M. Alshayeb, M. Naji, M. O. Elish, and J. Al-Ghamdi, "Towards measuring object-oriented class stability," *IET Software*, vol. 5, no. 4, pp. 415–424, 2011.

[239] H. M. Olague, L. H. Etzkorn, W. Li, and G. Cox, "Assessing design instability in iterative (agile) object-oriented projects," *Journal of Software Maintenance and Evolution*, vol. 18, no. 4, pp. 237–266, 2006.

[240] D. Azar and J. Vybihal, "An ant colony optimization algorithm to improve software quality prediction models: Case of class stability," *Information and Software Technology*, vol. 53, no. 4, pp. 388–393, 2011.

[241] M. Alshayeb, "The impact of refactoring on class and architecture stability," *Journal of Research and Practice in Information Technology*, vol. 43, no. 4, pp. 269–284, 2011.

[242] E. Figueiredo, B. Silva, C. Sant'Anna, A. Garcia, J. Whittle, and D. Nunes, "Crosscutting patterns and design stability: An exploratory analysis," in *IEEE 17th International Conference on Program Comprehension (ICPC)*, 2009, Conference Proceedings, pp. 138–147.

[243] J. M. Conejero, E. Figueiredo, A. F. Garcia, J. Hernández, and E. Jurado, *Early Crosscutting Metrics as Predictors of Software Instability*, ser. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2009, vol. 33, book section 9, pp. 136–156.

[244] M. Elish, "Do structural design patterns promote design stability?" in *30th Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2006, Conference Proceedings, pp. 215–220.

[245] M. Alshayeb, "On the relationship of class stability and maintainability," *IET Software*, vol. 7, no. 6, pp. 339–347, 2013.

[246] A. Baqais, M. Amro, and M. Alshayeb, "Analysis of the correlation between class stability and maintainability," in *7th International Conference on Computer Science and Information Technology (CSIT)*, 2016, Conference Proceedings, pp. 1–4.

[247] M. E. Fayad and A. Altman, "An introduction to software stability," *Communications of the ACM*, vol. 44, no. 9, p. 95, 2001.

[248] M. E. Fayad, "Accomplishing software stability," *Communications of the ACM*, vol. 45, no. 1, pp. 111–115, 2002.

[249] M. E. Fayad, "How to deal with software stability," *Communications of the ACM*, vol. 45, no. 4, pp. 109–112, 2002.

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

[251] C. C. Chiang, "Software stability in software reengineering," in *IEEE International Conference on Information Reuse and Integration*, 2007, Conference Proceedings, pp. 719–723.

[252] A. Mahdy, M. E. Fayad, D. Hamza, and P. Tugnawat, "Stable and reusable model-based architectures," in *Workshop on Model-based Software Reuse, co-located with 16th European Conference on Object-Oriented Programming (ECOOP)*, A. Speck, E. Pulvermüller, M. Clauß, R. V. D. Straeten, and R. Reussner, Eds., 2002, Conference Proceedings.

[253] M. E. Fayad, S. Wu, and M. Nabavi, "Stable model-based software reuse," in *Workshop on Model-based Software Reuse, co-located with 16th European Conference on Object-Oriented Programming (ECOOP)*, A. Speck, E. Pulvermüller, M. Clauß, R. V. D. Straeten, and R. Reussner, Eds., 2002, Conference Proceedings.

[254] S. Wu, "Implementation method for business objects in software stability modeling," in *IEEE International Conference on Information Reuse and Integration*, 2007, Conference Proceedings, pp. 730–733.

[255] E. R. Naganathan and X. P. Eugene, "Software Stability Model (SSM) for building reliable real time computing systems," in *3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009, Conference Proceedings, pp. 431–435.

[256] E. Yavari and M. E. Fayad, "A stable software model for mri visual analyzer," in *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2013, Conference Proceedings, pp. 324–325.

[257] M. E. Fayad and C. A. Flood, "Unified software engineering reuse (user) using stable analysis, design and architectural patterns," in *Future Technologies Conference (FTC)*, 2016, Conference Proceedings, pp. 706–711.

[258] H. S. Hamza and M. E. Fayad, "Model-based software reuse using stable analysis patterns," in *Workshop on Model-based Software Reuse, co-located with 16th European Conference on Object-Oriented Programming (ECOOP)*, A. Speck, E. Pulvermüller, M. Clauß, R. V. D. Straeten, and R. Reussner, Eds., 2002, Conference Proceedings.

[259] H. S. Hamza, "Towards stable software analysis patterns," in *Companion of the 17th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2002, Conference Proceedings, pp. 110–111.

[260] M. E. Fayad and S. Das, "The visualization stable analysis pattern," in *IEEE International Conference on Information Reuse and Integration*, 2007, Conference Proceedings, pp. 701–706.

[261] M. E. Fayad and S. Das, "The classification stable analysis pattern," in *IEEE International Conference on Information Reuse and Integration*, 2007, Conference Proceedings, pp. 707–712.

[262] M. E. Fayad, J. Rajagopalan, and A. Ranganath, "Anylog stable design pattern," in *5th IEEE Workshop on Mobile Computing Systems and Applications*, 2003, Conference Proceedings, pp. 566–571.

[263] R. Goverdhana and M. E. Fayad, "Any transaction stable design pattern," in *IEEE International Conference on Information Reuse and Integration (IRI)*, 2004, Conference Proceedings, pp. 54–59.

[264] M. E. Fayad and H. Kilaru, "AnyInformationHiding: a stable design pattern," in *IEEE International Conference on Information Reuse and Integration (IRI)*, 2005, Conference Proceedings, pp. 108–113.

[265] S. K. Singh and M. E. Fayad, "The AnyCorrectiveAction stable design pattern," in *17th Conference on Pattern Languages of Programs*. ACM, 2010, Conference Proceedings, pp. 1–20.

[266] M. E. Fayad and T. Sujatha, "The learning stable analysis pattern," in *IEEE International Conference on Information Reuse and Integration (IRI)*, 2005, Conference Proceedings, pp. 597–602.

[267] M. E. Fayad and C. A. Flood, "Using reputation stable analysis patterns as model based software reuse," in *Future Technologies Conference (FTC)*, 2016, Conference Proceedings, pp. 725–730.

[268] H. S. Hamza and M. E. Fayad, "Engineering and reusing stable atomic knowledge (SAK) patterns," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2003, Conference Proceedings, pp. 308–309.

[269] H. S. Hamza, A. Mahdy, M. E. Fayad, and M. Cline, *Extracting Domain-Specific and Domain-Neutral Patterns Using Software Stability Concepts*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2817, book section 18, pp. 191–201.

[270] A. M. Mahdy, H. S. Hamza, M. E. Fayad, and M. Cline, "Identifying domain patterns using software stability," in *IEEE International Conference on Information Reuse and Integration (IRI)*, 2004, Conference Proceedings, pp. 18–23.

[271] H. S. Hamza, "SODA: a stability-oriented domain analysis method," in *Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. ACM, 2004, Conference Proceedings, pp. 220–221.

[272] H. S. Hamza, "Separation of concerns for evolving systems: a stability-driven approach," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[273] H. S. Hamza, "A semi-automated approach for analyzing, separating, and modeling of concerns in evolving systems," in *Companion to the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2005, Conference Proceedings, pp. 220–221.

[274] P. E. Xavier and E. R. Naganathan, "Productivity improvement in software projects using 2-dimensional Probabilistic sSoftware Stability Model (PSSM)," *SIGSOFT Software Engineering Notes*, vol. 34, no. 5, pp. 1–3, 2009.

[275] E. R. Naganathan and P. E. Xavier, "Architecting autonomic computing systems through Probabilistic Software Stability Model (PSSM)," in *2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. ACM, 2009, Conference Proceedings, pp. 643–648.

[276] D. Van Landuyt, S. Op de beeck, E. Truyen, and W. Joosen, "Domain-driven discovery of stable abstractions for pointcut interfaces," in *8th ACM international conference*

on Aspect-oriented software development. ACM, 2009, Conference Proceedings, pp. 75–86.

[277] D. Van Landuyt, S. Op de beeck, E. Truyen, and W. Joosen, *Domain-Driven Discovery of Stable Abstractions for Pointcut Interfaces.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–52.

[278] D. Van Landuyt, E. Truyen, and W. Joosen, "Automating the discovery of stable domain abstractions for reusable aspects," in *ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, 2009, Conference Proceedings, pp. 1–7.

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

[280] 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.

[281] C. Canal and G. Salaün, *Stability-Based Adaptation of Asynchronously Communicating Software.* Springer International Publishing, 2016, pp. 321–336.

[282] M. H. Samadzadeh and S. J. Khan, "Stability, coupling, and cohesion of object-oriented software systems," in *22nd Annual ACM Computer Science Conference on Scaling up: Meeting the challenge of complexity in real-world computing applications.* ACM, 1994, Conference Proceedings, pp. 312–319.

[283] U. S. Poornima and V. Suma, *An Investigation on Coupling and Cohesion as Contributory Factors for Stable System Design and Hence the Influence on System Maintainability and Reusability.* Springer International Publishing, 2015, pp. 645–650.

[284] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," in *8th European Conference on Software Maintenance and Reengineering (CSMR)*, 2004, Conference Proceedings, pp. 223–232.

[285] R. Vasa, J. G. Schneider, and O. Nierstrasz, "The inevitable stability of software change," in *IEEE International Conference on Software Maintenance*, 2007, Conference Proceedings, pp. 4–13.

[286] F. A. Fontana and S. Maggioni, "Metrics and antipatterns for software quality evaluation," in *IEEE 34th Software Engineering Workshop*, 2011, Conference Proceedings, pp. 48–56.

[287] R. C. Martin, *Agile software development: principles, patterns, and practices.* New Jersey: Prentice Hall, 2002.

[288] J. Chow and E. Tempero, "Stability of Java interfaces: a preliminary investigation," in *2nd International Workshop on Emerging Trends in Software Metrics.* ACM, 2011, Conference Proceedings, pp. 38–44.

[289] S. Jenkins and S. R. Kirk, "Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution," *Information Sciences*, vol. 177, no. 12, pp. 2587–2601, 2007.

[290] J. Ruohonen, S. Hyrynsalmi, and V. Leppänen, "Exploring the stability of software with time-series cross-sectional data," in *2nd International Workshop on Software Architecture and Metrics*. IEEE Press, 2015, Conference Proceedings, pp. 41–47.

[291] M. Alenezi and F. Khellah, "Evolution impact on architecture stability in open-source projects," *International Journal of Cloud Applications and Computing*, vol. 5, no. 4, pp. 24–35, 2015.

[292] A. AbuHassan and M. Alshayeb, "A metrics suite for UML model stability," *Software & Systems Modeling*, 2016.

[293] S. A. Tonu, A. Ashkan, and L. Tahvildari, "Evaluating architectural stability using a metric-based approach," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, Conference Proceedings, pp. 10 pp.–270.

[294] L. Aversano, M. Molfetta, and M. Tortorella, "Evaluating architecture stability of software projects," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, Conference Proceedings, pp. 417–424.

[295] F. Handani and S. Rochimah, "Relationship between features volatility and software architecture design stability in object-oriented software: Preliminary analysis," in *International Conference on Information Technology Systems and Innovation (ICITSI)*, 2015, Conference Proceedings, pp. 1–5.

[296] E. Figueiredo, I. Galvao, S. S. Khan, A. F. Garcia, C. Sant'Anna, A. Pimentel, A. L. Medeiros, L. Fernandes, T. V. Batista, R. Ribeiro, P. van den Broek, M. Aksit, S. Zschaler, and A. Moreira, "Detecting architecture instabilities with concern traces: An exploratory study," in *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*, 2009, Conference Proceedings, pp. 261–264.

[297] A. L. Medeiros, E. Figueiredo, I. Galvao, A. F. Garcia, T. V. Batista, and C. Sant'Anna, "Concern-based assessment of architectural stability: A comparative study," in *4th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, 2010, Conference Proceedings, pp. 130–139.

[298] R. L. Nord, I. Ozkaya, R. S. Sangwan, and R. J. Koontz, "Architectural dependency analysis to understand rework costs for safety-critical systems," in *Companion Proceedings of 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, Conference Proceedings, pp. 185–194.

[299] E. Constantinou and I. Stamelos, "Identifying evolution patterns: A metrics-based approach for external library reuse," *Software: Practice and Experience*, vol. 47, no. 7, pp. 1027–1039, 2017.

[300] M. Mirakhorli and J. Cleland-Huang, "Modifications, tweaks, and bug fixes in architectural tactics," in *IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, Conference Proceedings, pp. 377–380.

[301] S. Rafiliu, P. Eles, and Z. Peng, "Stability conditions of on-line resource managers for systems with execution time variations," in *23rd Euromicro Conference on Real-Time Systems*, 2011, Conference Proceedings, pp. 151–161.

[302] S. Rafiliu, P. Eles, and Z. Peng, "Stability of adaptive feedback-based resource managers for systems with execution time variations," *Real-Time Systems*, vol. 49, no. 3, pp. 367–400, 2013.

[303] J. Porter, G. Hemingway, N. Kottenstette, G. Karsai, and J. Sztipanovits, "Online stability validation using sector analysis," in *10th ACM international conference on Embedded software*. ACM, 2010, Conference Proceedings, pp. 29–38.

[304] 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.

[305] T. Weis and A. Wacker, "Self-stabilizing embedded systems," in *Workshop on Organic computing*. ACM, 2011, Conference Proceedings, pp. 59–66.

[306] J. Heo and T. Abdelzaher, "Adaptguard: guarding adaptive systems from instability," in *6th international conference on Autonomic computing*. ACM, 2009, Conference Proceedings, pp. 77–86.

[307] S. Yerramalla, B. Cukic, M. Mladenovski, and E. Fuller, "Stability monitoring and analysis of learning in an adaptive system," in *International Conference on Dependable Systems and Networks (DSN)*, 2005, Conference Proceedings, pp. 70–79.

[308] G. Mencagli and M. Vanneschi, "QoS-control of structured parallel computations: A predictive control approach," in *IEEE 3rd International Conference on Cloud Computing Technology and Science*, 2011, Conference Proceedings, pp. 296–303.

[309] G. Mencagli, M. Vanneschi, and E. Vespa, "A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 9, no. 1, pp. 1–27, 2014.

[310] G. Mencagli and M. Vanneschi, "Analysis of control-theoretic predictive strategies for the adaptation of distributed parallel computations," in *1st ACM workshop on Optimization techniques for resources management in clouds*. ACM, 2013, Conference Proceedings, pp. 33–40.

[311] G. Mencagli, M. Vanneschi, and E. Vespa, "Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments," in *International Conference on High Performance Computing & Simulation (HPCS)*, 2013, Conference Proceedings, pp. 11–18.

[312] N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili, "Formal analysis of policy-based self-adaptive systems," in *ACM Symposium on Applied Computing*. ACM, 2010, Conference Proceedings, pp. 2536–2543.

[313] 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.

[314] 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.

[315] 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.

[316] 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.

[317] 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.

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

[319] 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.

[320] 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.

[321] 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

[322] 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.

[323] 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

[324] 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.

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

[326] 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.

[327] 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.

[328] 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.

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

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

[331] 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.

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

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

[334] 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.

[335] 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.

[336] 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.

[337] 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.

[338] 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.

[339] 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.

[340] 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.

[341] 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.

[342] 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.

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

[344] 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.

[345] 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.

[346] 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.

[347] 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.

[348] 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.

[349] 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.

[350] 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.

[351] 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.

[352] 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.

[353] 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 (SEAMS)*. ACM, 2011, pp. 218–227.

[354] 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.

[355] 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.

[356] 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.

[357] 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

[358] 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.

[359] 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.

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

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

[362] 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.

[363] 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.

[364] 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.

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

[366] 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.

[367] 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.

[368] 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.

[369] 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.

[370] 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.

[371] 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.

[372] 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.

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

[374] 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.

[375] 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.

[376] H. Koziolek, "Sustainability evaluation of software architectures: A systematic review," in *Joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. ACM, pp. 3–12.

[377] 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.

[378] 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.

[379] 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.

[380] 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.

[381] A. van Lamsweerde, "Requirements engineering in the year 00: A research perspective," in *22nd International Conference on Software Engineering (ICSE)*. 337184: ACM, 2000, pp. 5–19.

[382] 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.

[383] 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.

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

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

[386] 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.

[387] 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.

[388] 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.

[389] 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.

[390] 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, book section 5, pp. 122–145.

[391] 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.

[392] 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.

[393] 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.

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

[395] 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.

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

[397] 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.

[398] 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.

[399] 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.

[400] 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.

[401] 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.

[402] 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.

[403] 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.

[404] R. Laddaga, "Creating robust software through self-adaptation," *IEEE Intelligent Systems and their Applications*, vol. 14, no. 3, pp. 26–29, 1999.

[405] J. C. Georgas, "Knowledge-based architectural adaptation management for self-adaptive systems," in *27th International Conference on Software Engineering*. ACM, 2005, pp. 658–658.

[406] 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.

[407] 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.

[408] A. P. Engelbrecht, *Computational intelligence: An introduction*. Chichester: J. Wiley & Sons, 2002.

[409] P. R. Lewis, *Computational Self-awareness and Learning Machines*. London, UK: Imperial College Press, 2014, pp. 267–280.

[410] 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.

[411] 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.

[412] G. Chatzikonstantinou and K. Kontogiannis, "Run-time requirements verification for reconfigurable systems," *Information and Software Technology*, vol. 75, pp. 105–121, 2016.

[413] 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.

[414] R. Calinescu, M. Autili, J. Cámara, A. Di Marco, S. Gerasimou, P. Inverardi, A. Perucci, N. Jansen, J. P. Katoen, M. Kwiatkowska, and et. al., "Synthesis and verification of self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017, pp. 337–373.

[415] 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.

[416] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *33rd International Conference on Software Engineering*. ACM, 2011, pp. 341–350.

[417] 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.

[418] 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.

[419] 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.

[420] 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.

[421] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and

T. Vogel, "Software engineering meets control theory," in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Press, 2015, pp. 71–82.

[422] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC/IEEE 42010 – Systems and software engineering – Architecture description," ISO/IEC, Report ISO/IEC/IEEE 42010:2011(E), 2011.

[423] B. Tekinerdogan and H. Sozer, "Variability viewpoint for introducing variability in software architecture viewpoints," in *WICSA/ECSA Companion Volume*. ACM, 2012, pp. 163–166.

[424] H. Koning and H. van Vliet, "A method for defining IEEE Std 1471 viewpoints," *Journal of Systems and Software*, vol. 79, no. 1, pp. 120–131, 2006.

[425] G. Qing and P. Lago, "On service-oriented architectural concerns and viewpoints," in *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*, 2009, pp. 289–292.

[426] P. Kruchten, R. Capilla, and J. C. Dueas, "The decision view's role in software architecture practice," *IEEE Software*, vol. 26, no. 2, pp. 36–42, 2009.

[427] R. Kazman and L. Bass, "Toward deriving software architectures from quality attributes," Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-94-TR-010, 1994.

[428] P. Narman, M. Buschle, J. Konig, and P. Johnson, "Hybrid probabilistic relational models for system quality analysis," in *14th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 2010, pp. 57–66.

[429] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC 9126-1 – Information technology – Software product quality – Quality model," ISO/IEC, Report ISO/IEC 9126-1:2001, 2000.

[430] T. Hall and N. E. Fenton, "Implementing effective software metrics programs," *IEEE Software*, vol. 14, no. 2, pp. 55–65, 1997.

[431] R. E. Park, W. B. Goethert, and W. A. Florac, "Goal-driven software measurement. a guidebook," Software Engineering Institute, Carnegie Mellon University, Report Technical Report CMU/SEI-96-HB-002, 1996.

[432] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal question metric (gqm) approach," in *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[433] C. Calero and M. Piattini, "Introduction to green in software engineering," in *Green in Software Engineering*, C. Calero and M. Piattini, Eds. Springer, 2015, pp. 3–27.

[434] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–47, 2014.

[435] 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.

[436] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[437] C. P. Robert, *The Bayesian choice: from decision-theoretic foundations to computational implementation*. Springer Science & Business Media, 2007.

[438] U. B. Kjaerulff and A. L. Madsen, *Bayesian Networks and Influence Diagrams: A guide to construction and analysis*, ser. Information Science and Statistics. New York London: Springer, 2008.

[439] N. Bencomo, A. Belaggoun, and V. Issarny, "Bayesian artificial intelligence for tackling uncertainty in self-adaptive systems: The case of dynamic decision networks," in *2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2013, pp. 7–13.

[440] R. E. Neapolitan, *Learning Bayesian Networks*. Pearson Prentice Hall, 2004.

[441] J. Pearl, "Graphical models, causality, and intervention," *Statistical Science*, vol. 8, no. 3, pp. 266–273, 1993.

[442] F. V. Jensen, *Bayesian Networks and Decision Graphs*. Springer-Verlag New York, Inc., 2001.

[443] J. Pearl, "Fusion, propagation, and structuring in belief networks," *Artificial Intelligence*, vol. 29, no. 3, pp. 241–288, 1986.

[444] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.

[445] J. Q. Smith, *Bayesian Decision Analysis: Principles and Practice*. Cambridge University Press, 2010.

[446] P. Lago, P. Avgeriou, and P. Kruchten, "Fifth international workshop on sharing and reusing architectural knowledge (shark)," in *ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, vol. 2, 2010, pp. 437–438.

[447] F. V. Jensen, *An Introduction to Bayesian Networks*. London: UCL Press, 1996.

[448] C. Ghezzi and G. Tamburrelli, "Reasoning on non-functional requirements for integrated services," in *17th IEEE International Requirements Engineering Conference (RE)*, 2009, pp. 69–78.

[449] M. Hölzl and T. Gabor, "Reasoning and learning for awareness and adaptation," in *Software Engineering for Collective Autonomic Systems: The ASCENS Approach*, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer International Publishing, 2015, pp. 249–290.

[450] H. Aydt, S. J. Turner, W. Cai, and M. Y. H. Low, "Research issues in symbiotic simulation," in *Winter Simulation Conference (WSC)*, 2009, pp. 1213–1222.

[451] S. J. Turner, "Symbiotic simulation and its application to complex adaptive systems (keynote)," in *IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2011, pp. 3–3.

[452] B. Tjahjono and J. Xu, "Linking symbiotic simulation to enterprise systems: Framework and applications," in *Winter Simulation Conference (WSC)*, 2015, pp. 823–834.

[453] IBM, "An architectural blueprint for autonomic computing," Technical Report, 2003.

[454] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[455] Amazon Web Services, Inc., "Amazon EC2 Instance Types," accessed: 2017-10-01. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[456] "Rice University Bidding System (RUBiS)." [Online]. Available: www.rubis.ow2.org

[457] M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, 2000.

[458] T. Chen and R. Bahsoon, "Symbiotic and sensitivity-aware architecture for globally-optimal benefit in self-adaptive cloud," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 85–94.

[459] A. Ekárt and S. Z. Neméth, "Stability analysis of tree structured decision functions," *European Journal of Operational Research*, vol. 160, no. 3, pp. 676–695, 2005.

[460] S. Parsons, R. Bahsoon, P. R. Lewis, and X. Yao, "Towards a better understanding of self-awareness and self-expression within software systems," University of Birmingham, School of Computer Science, Tech. Rep. CSR-11-03, April 2011.

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

[462] F. J. Affonso and E. Y. Nakagawa, "A reference architecture based on reflection for self-adaptive software," in *VII Brazilian Symposium on Software Components, Architectures and Reuse*, 2013, pp. 129–138.

[463] T. Nya, S. Stilkerich, P. R. Lewis, and X. Yao, "Self-aware and self-expressive systems," *Awareness Magazine*, 2014.

[464] 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.

[465] A. van Lamsweerde, *Requirements Engineering: From system goals to UML models to software specifications*. Wiley, 2009.

[466] 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.

[467] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 60–69.

[468] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[469] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior," in *9th International Workshop on Software Specification and Design*, 1998, pp. 50–59.

[470] D. Garlan, "A 10-year perspective on software engineering self-adaptive systems (keynote)," in *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Press, 2013, pp. 2–2.

[471] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein, "Requirements-aware systems: A research agenda for re for self-adaptive systems," in *18th IEEE International Requirements Engineering Conference (RE)*, 2010, pp. 95–103.

[472] S. Kim, K. Dae-Kyoo, L. Lunjin, and P. Soo-Yong, "A tactic-based approach to embodying non-functional requirements into software architectures," in *12th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, 2008, pp. 139–148.

[473] S. Kim, D.-K. Kim, L. Lu, and S. Park, "Quality-driven architecture development using architectural tactics," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1211–1231, 2009.

[474] F. Bachmann, L. Bass, and R. Nord, "Modifiability tactics," Software Engineering Institute, Carnegie Mellon University, Report Technical Report CMU/SEI-2007-TR-002, 2007.

[475] R. Champagne and S. Gagne, "Towards automation of performance architectural tactics application," in *9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2011, pp. 157–160.

[476] A. Sanchez, A. Aguiar, L. S. Barbosa, and D. Riesco, "Analysing tactics in architectural patterns," in *35th Annual IEEE Software Engineering Workshop (SEW)*, 2011, pp. 32–41.

[477] M. Mirakhorli, J. Carvalho, J. Cleland-Huang, and P. Mader, "A domain-centric approach for recommending architectural tactics to satisfy quality concerns," in *3rd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, 2013, pp. 1–8.

[478] D. Garlan, C. Shang-Wen, H. An-Cheng, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[479] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds. Springer Berlin Heidelberg, 2013, vol. 7475, book section 4, pp. 76–107.

[480] A. van Lamsweerde, "From system goals to software architecture," in *Formal Methods for Software Architectures*, ser. Lecture Notes in Computer Science, M. Bernardo and P. Inverardi, Eds. Springer Berlin Heidelberg, 2003, vol. 2804, book section 2, pp. 25–43.

[481] E. Cavalcante, T. Batista, N. Bencomo, and P. Sawyer, "Revisiting goal-oriented models for self-aware systems-of-systems," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2015, pp. 231–234.

[482] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes, "Goal-based modeling of dynamically adaptive system requirements," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, 2008, pp. 36–45.

[483] M. Vrbaski, G. Mussbacher, D. Petriu, and D. Amyot, "Goal models as run-time entities in context-aware systems," in *7th Workshop on Models@run.time*. ACM, 2012, pp. 3–8.

[484] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[485] P. Dayan and C. J. C. H. Watkins, "Reinforcement learning: A computational perspective," in *Encyclopedia of Cognitive Science*. John Wiley & Sons, Ltd, 2006.

[486] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[487] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis, "Automatic verification of competitive stochastic systems," *Formal Methods in System Design*, vol. 43, no. 1, pp. 61–92, 2013.

[488] M. Kwiatkowska, D. Parker, and C. Wiltsche, "Prism-games 2.0: A tool for multi-objective strategy synthesis for stochastic games," in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, M. Chechik and J. F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 560–566.

[489] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis, "Prism-games: A model checker for stochastic multi-player games," in *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, N. Piterman and S. A. Smolka, Eds. Springer Berlin Heidelberg, 2013, pp. 185–191.

[490] T. Chen and J. Lu, "Probabilistic alternating-time temporal logic and model checking algorithm," in *4th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, vol. 2, 2007, pp. 35–39.

[491] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *Journal of the ACM (JACM)*, vol. 49, no. 5, pp. 672–713, 2002.

[492] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Foundations of Software Technology and Theoretical Computer Science: 15th Conference Bangalore, India, December 18–20, 1995 Proceedings*, P. S. Thiagarajan, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 499–513.

[493] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, M. Bernardo and V. Issarny, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 53–113.

[494] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1467–1493, 2013.

[495] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue, "Learning revised models for planning in adaptive systems," in *International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 63–71.

[496] Z. Ding, Y. Zhou, and M. Zhou, "Modeling self-adaptive software systems with learning petri nets," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 4, pp. 483–498, 2016.

[497] D. Han, J. Xing, Q. Yang, J. Li, and H. Wang, "Handling uncertainty in self-adaptive software using self-learning fuzzy neural network," in *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2016, pp. 540–545.

[498] K. Dongsun and P. Sooyong, "Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software," in *ICSE Workshop*

on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2009, pp. 76–85.

[499] J. Teich, "From dynamic reconfiguration to self-reconfiguration: Invasive algorithms and architectures," in *International Conference on Field-Programmable Technology (FPT)*, 2009, pp. 11–12.

[500] R. Mirandola and P. Potena, "Self-adaptation of service based systems based on cost/quality attributes tradeoffs," in *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (Synasc)*, 2010, pp. 493–501.

[501] D. Perez-Palacin, R. Mirandola, and J. Merseguer, "Qos and energy management with petri nets: A self-adaptive framework," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2796–2811, 2012.

[502] X. Peng, B. Chen, Y. Yu, and W. Zhao, "Self-tuning of software systems through dynamic quality tradeoff and value-based feedback control loop," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2707–2719, 2012.

[503] L. W. Shen, X. Peng, and W. Y. Zhao, "Quality-driven self-adaptation: Bridging the gap between requirements and runtime architecture by design decision," in *IEEE 36th Annual Computer Software and Applications Conference (COMPSAC)*, X. Bai, F. Belli, E. Bertino, C. K. Chang, A. Elci, C. Seceleanu, H. Xie, and M. Zulkernine, Eds., 2012, pp. 185–194.

[504] S. W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[505] J. Camara, D. Garlan, B. Schmerl, and A. Pandey, "Optimal planning for architecture-based self-adaptation via model checking of stochastic games," in *30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2015, pp. 428–435.

[506] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "Iso/iec 42030 – architecture evaluation (draft)," ISO/IEC, Report ISO/IEC JTC 1/SC 7/WG 42 N0153, 2016.

[507] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC 42030 – systems and software engineering, architecture evaluation," ISO/IEC, Report WD3, 2013.

[508] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC 19501 – Information technology – Open Distributed Processing – Unified Modeling Language (UML)," ISO/IEC, Report ISO/IEC 19501:2005, 2005.

[509] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), "ISO/IEC 33001:2015 Information technology – Process assessment – Concepts and terminology," ISO/IEC, Report, 2015. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=54175

[510] P. Avgeriou, P. Lago, J. Grundy, I. Mistrik, and J. Hall, *Relating Software Requirements and Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[511] R. Bahsoon and W. Emmerich, "Architectural stability," University College London, Technical Report, 2006.

[512] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.

[513] 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.

[514] 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.

[515] 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.

[516] P. Spirtes, C. Glymour, and R. Scheines, *Causation, prediction, and search*, 2nd ed. Cambridge, Mass.: MIT Press, 2000.

[517] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. Kerminen, "A linear non-gaussian acyclic model for causal discovery," *Journal of Machine Learning Research*, vol. 7, pp. 2003–2030, 2006.

[518] K. Fukuzawa and T. Kobayashi, "Specifying and evaluating software architectures based on 4+1 View Model," in *Engineering Information Systems in the Internet Context*, ser. IFIP — The International Federation for Information Processing, C. Rolland, S. Brinkkemper, and M. Saeki, Eds. Springer US, 2002, vol. 103, book section 3, pp. 31–51.

[519] "The openstack cloud platform." [Online]. Available: http://openstack.org/

[520] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[521] M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the scalability of a large web-based shopping system," *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 44–69, 2001.

[522] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: the search for invariants," *ACM SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1, pp. 126–137, 1996.

[523] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao, "Characterization of a large web site population with implications for content delivery," *World Wide Web*, vol. 9, no. 4, pp. 505–536, 2006.

[524] L. Guo, S. Chen, Z. Xiao, and X. Zhang, "Analysis of multimedia workloads with implications for internet streaming," in *14th international conference on World Wide Web*. ACM, 2005.

[525] K. Sripanidkulchai, B. Maggs, and H. Zhang, "An analysis of live streaming workloads on the internet," in *4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 41–54.

[526] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, 2007.

[527] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, pp. 222–235, 2014.

[528] JabRef Development Team, *JabRef*. [Online]. Available: http://www.jabref.org

[529] 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.

[530] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A formal approach to autonomic systems programming: The scel language," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 2, pp. 7:1–7:29, 2014.

[531] T. Chen and R. Bahsoon, "Toward a smarter cloud: Self-aware autoscaling of cloud configurations and resources," *Computer*, vol. 48, no. 9, pp. 93–96, 2015.

[532] M. Hölzl and T. Gabor, *Reasoning and Learning for Awareness and Adaptation*. Springer International Publishing, 2015, pp. 249–290.

[533] P. Andras and B. G. Charlton, "Self-aware software - will it become a reality?" in *Self-Star Properties in Complex Information Systems: Conceptual and Practical Foundations*, ser. Lecture Notes In Computer Science, 2005, vol. 3460, pp. 229–259.

[534] R. Sterritt and M. Hinchey, "Why computer-based systems should be autonomic," in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, 2005, pp. 406–412.

[535] D. B. Abeywickrama, F. Zambonelli, and N. Hoch, "Towards simulating architectural patterns for self-aware and self-adaptive systems," in *IEEE 6th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, 2012, pp. 133–138.

[536] D. B. Abeywickrama, N. Hoch, and F. Zambonelli, "Simsota: Engineering and simulating feedback loops for self-adaptive systems," in *International C* Conference on Computer Science and Software Engineering (C3S2E)*. ACM, 2013, pp. 67–76.

[537] N. Šerbedžija, T. Bureš, and J. Keznikl, "Engineering autonomous systems," in *17th Panhellenic Conference on Informatics (PCI)*. ACM, 2013, pp. 128–135.

[538] D. Dannenhauer, M. T. Cox, S. Gupta, M. Paisner, and D. Perlis, "Toward meta-level control of autonomous agents," *Procedia Computer Science*, vol. 41, pp. 226 – 232, 2014, 5th Annual International Conference on Biologically Inspired Cognitive Architectures (BICA).

[539] R. Gioiosa, G. Kestor, D. J. Kerbyson, and A. Hoisie, "Cross-layer self-adaptive/self-aware system software for exascale systems," in *IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2014, pp. 326–333.

[540] E. Riccobene and P. Scandurra, "Formal modeling self-adaptive service-oriented applications," in *30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2015, pp. 1704–1710.

[541] H. Giese, T. Vogel, A. Diaconescu, S. Götz, N. Bencomo, K. Geihs, S. Kounev, and K. L. Bellman, "State of the art in architectures for self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017, pp. 237–275.

[542] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli, "A roadmap towards sustainable self-aware service systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2010, pp. 10–19.

[543] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "A generalized software framework for accurate and efficient management of performance goals," in *International Conference on Embedded Software (EMSOFT)*, 2013, pp. 1–10.

[544] E. E. Veas, K. Kiyokawa, and H. Takemura, "Self-aware framework for adaptive augmented reality," in *International Conference on Augmented Tele-existence (ICAT)*. ACM, 2005, pp. 70–77.

[545] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani, "On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles," in *5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, 2011, pp. 108–113.

[546] N. Bicocchi, D. Fontana, and F. Zambonelli, "A self-aware, reconfigurable architecture for context awareness," in *IEEE Symposium on Computers and Communications (ISCC)*, 2014, pp. 1–7.

[547] M. Salama and R. Bahsoon, "Quality-driven architectural patterns for self-aware cloud-based software," in *IEEE 8th International Conference on Cloud Computing*, 2015, pp. 844–851.

[548] E. Gerbert-Gaillard, S. Chollet, and P. Lalanda, "Model-driven approach for self-aware pervasive systems," in *IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, 2016, pp. 1–6.

[549] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Baehr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 432–452, 2017.

[550] P. Lalanda, E. Gerber-Gaillard, and S. Chollet, "Self-aware context in smart home pervasive platforms," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 119–124.

[551] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "Seec: A general and extensible framework for self-aware computing," Computer Science and Artificial Intelligence Laboratory MIT, Technical Report MIT-CSAIL-TR-2011-046, 2011.

[552] N. K. Thanigaivelan, E. Nigussie, S. Virtanen, and J. Isoaho, "Towards self-aware approach for mobile devices security," in *Computer Network Security*, J. Rak, J. Bay, I. Kotenko, L. Popyack, V. Skormin, and K. Szczypiorski, Eds. Springer International Publishing, 2017, pp. 171–182.

[553] M. Mitchell, "Self-awareness and control in decentralized systems." in *AAAI Spring Symposium: Metacognition in Computation*, 2005, pp. 80–85.

[554] E. Vassev and M. Hinchey, "Knowledge representation and reasoning for intelligent software systems," *Computer*, vol. 44, no. 8, pp. 96–99, 2011.

[555] E. Vassev and M. Hinchey, "Awareness in software-intensive systems," *Computer*, vol. 45, no. 12, pp. 84–87, 2012.

[556] C. Landauer and K. L. Bellman, "An architecture for self-awareness experiments," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 255–262.

[557] C. H. Huang, J. S. Shen, and P. A. Hsiung, "A self-adaptive hardware/software system architecture for ubiquitous computing applications," in *Ubiquitous Intelligence and Computing*, ser. Lecture Notes in Computer Science, Z. Yu, R. Liscano, G. L. Chen, D. Q. Zhang, and X. S. Zhou, Eds., vol. 6406. Nokia; Ind Corp China, 2010, pp. 382–396, 7th International Conference on Autonomic and Trusted Computing, NW Polytechn Univ, Xian, PEOPLES R CHINA, OCT 26-29, 2010.

[558] I. Breskovic, C. Haas, S. Caton, and I. Brandic, "Towards self-awareness in cloud markets: A monitoring methodology," in *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2011, pp. 81–88.

[559] F. Zambonelli, G. Castelli, L. Ferrari, M. Mamei, A. Rosi, G. D. Marzo, M. Risoldi, A. E. Tchao, S. Dobson, G. Stevenson, J. Ye, E. Nardini, A. Omicini, S. Montagna, M. Viroli, A. Ferscha, S. Maschek, and B. Wally, "Self-aware pervasive service ecosystems," *Procedia Computer Science*, vol. 7, pp. 197 – 199, 2011.

[560] G. Castelli, M. Mamei, A. Rosi, and F. Zambonelli, "Engineering pervasive service ecosystems: The sapere approach," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 1, pp. 1:1–1:27, 2015.

[561] Y. Su, F. Shi, S. Talpur, Y. Wang, S. Hu, and J. Wei, "Achieving self-aware parallelism in stream programs," *Cluster Computing - The Journal of Networks Software Tools and Applications*, vol. 18, no. 2, SI, pp. 949–962, 2015.

[562] I. Alzuru, A. Matsunaga, M. Tsugawa, and J. A. B. Fortes, "Selfie: Self-aware information extraction from digitized biocollections," in *IEEE 13th International Conference on e-Science (e-Science)*, 2017, pp. 69–78.

[563] A. Griffiths, "ldquo;self rdquo;-conscious objects in object-z," in *Technology of Object-Oriented Languages and Systems (TOOLS 25)*, 1997, pp. 210–224.

[564] A. Bronstein, J. Das, M. Duro, R. Friedrich, G. Kleyner, M. Mueller, S. Singhal, and I. Cohen, "Self-aware services: using bayesian networks for detecting anomalies in internet-based services," in *IEEE/IFIP International Symposium on Integrated Network Management*, 2001, pp. 623–638.

[565] A. G. Beltran, P. Milligan, and P. Sage, "Heterogeneity-aware distributed access structure," in *5th IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2005, pp. 152–153.

[566] R. Abbott and C. Sun, "Abstraction abstracted," in *2nd International Workshop on The Role of Abstraction in Software Engineering (ROA)*. ACM, 2008, pp. 23–30.

[567] O. Nierstrasz, M. Denker, T. Gîrba, A. Lienhard, and D. Röthlisberger, *Change-Enabled Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–79.

[568] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards self-aware performance and resource management in modern service-oriented systems," in *IEEE International Conference on Services Computing (SCC)*, 2010, pp. 621–624.

[569] S. Kounev, F. Brosig, and N. Huber, "Self-aware qos management in virtualized infrastructures," in *8th ACM International Conference on Autonomic Computing (ICAC)*. ACM, 2011, pp. 175–176.

[570] S. Kounev, F. Brosig, and N. Huber, "The descartes modeling language," Department of Computer Science, University of Wuerzburg, Technical Report, 2014.

[571] M. D. Santambrogio, H. Hoffmann, J. Eastep, and A. Agarwal, "Enabling technologies for self-aware adaptive systems," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2010, pp. 149–156.

[572] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "A model-based approach to designing self-aware it systems and infrastructures," *Computer*, vol. 49, no. 7, pp. 53–61, 2016.

[573] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "Seec: A framework for self-aware computing," 2010.

[574] E. Gerbert-Gaillard and P. Lalanda, "Self-aware model-driven pervasive systems," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2016, pp. 221–222.

[575] K. L. Bellman, C. Landauer, P. Nelson, N. Bencomo, S. Götz, P. R. Lewis, and L. Esterle, *Self-modeling and Self-awareness*. Springer International Publishing, 2017, pp. 279–304.

[576] E. Gerbert-Gaillard, P. Lalanda, S. Chollet, and J. Demarchez, "A self-aware approach to context management in pervasive platforms," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, pp. 256–261.

[577] A. Egyed, "Architecture differencing for self management," in *1st ACM SIGSOFT Workshop on Self-managed Systems*, ser. WOSS '04. New York, NY, USA: ACM, 2004, pp. 44–48.

[578] F. Faniyi, R. Bahsoon, A. Evans, and R. Kazman, "Evaluating security properties of architectures in unpredictable environments: A case for cloud," in *9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2011, pp. 127–136.

[579] A. Elhabbash, R. Bahsoon, and P. Tino, "Towards self-aware service composition," in *IEEE International Conference on High Performance Computing and Communications, IEEE 6th International Symposium on Cyberspace Safety and Security, IEEE 11th International Conference on Embedded Software and Systems (HPCC,CSS,ICESS)*, 2014, pp. 1275–1279.

[580] M. Autili, K. L. Bellman, A. Diaconescu, L. Esterle, M. Tivoli, and A. Zisman, *Transition Strategies for Increasing Self-awareness in Existing Types of Computing Systems*. Springer International Publishing, 2017, pp. 305–336.

[581] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and S. Kounev, "Architectural concepts for self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017, pp. 109–147.

[582] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and K. L. Bellman, "Generic architectures for individual self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017, pp. 149–189.

[583] E. Vassev and M. Hinchey, *Knowledge Representation for Adaptive and Self-aware Systems*. Springer International Publishing, 2015, pp. 221–247.

[584] J. O. Kephart, M. Maggio, A. Diaconescu, H. Giese, H. Hoffmann, S. Kounev, A. Koziolek, P. R. Lewis, A. Robertsson, and S. Spinner, *Reference Scenarios for Self-aware Computing*. Springer International Publishing, 2017, pp. 87–106.

[585] K. Tammemäe, A. Jantsch, A. Kuusik, J. Preden, and E. Õunapuu, *Self-Aware Fog Computing in Private and Secure Spheres*. Springer International Publishing, 2018, pp. 71–99.

[586] R. Birke, J. Cámara, L. Y. Chen, L. Esterle, K. Geihs, E. Gelenbe, H. Giese, A. Robertsson, and X. Zhu, *Self-aware Computing Systems: Open Challenges and Future Research Directions*. Springer International Publishing, 2017, pp. 709–722.

[587] L. Esterle, K. L. Bellman, S. Becker, A. Koziolek, C. Landauer, and P. R. Lewis, *Assessing Self-awareness.* Springer International Publishing, 2017, pp. 465–481.

[588] N. Herbst, S. Becker, S. Kounev, H. Koziolek, M. Maggio, A. Milenkoski, and E. Smirni, *Metrics and Benchmarks for Self-aware Computing Systems.* Springer International Publishing, 2017, pp. 437–464.

[589] J. O. Kephart, A. Diaconescu, H. Giese, A. Robertsson, T. Abdelzaher, P. R. Lewis, A. Filieri, L. Esterle, and S. Frey, *Self-adaptation in Collective Self-aware Computing Systems.* Springer International Publishing, 2017, pp. 401–435.

[590] S. Kounev, P. R. Lewis, K. L. Bellman, N. Bencomo, J. Camara, A. Diaconescu, L. Esterle, K. Geihs, H. Giese, S. Götz, P. Inverardi, J. O. Kephart, and A. Zisman, *The Notion of Self-aware Computing.* Springer International Publishing, 2017, pp. 3–16.

[591] S. Linkola, A. Kantosalo, T. Männistö, and H. Toivonen, "Aspects of self-awareness: An anatomy of metacreative systems," in *8th International Conference on Computational Creativity (ICCC)*, 2017.

[592] J. Walter, A. Di Marco, S. Spinner, P. Inverardi, and S. Kounev, *Online Learning of Run-Time Models for Performance and Resource Management in Data Centers.* Springer International Publishing, 2017, pp. 507–528.

[593] D. Budgen, M. Turner, P. Brereton, and B. A. Kitchenham, "Using mapping studies in software engineering," in *20th Annual Meeting of the Psychology of Programming Interest Group (PPIG).* Lancaster University, 2008, pp. 195–204.

[594] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using mapping studies as the basis for further research: A participant-observer case study," *Journal Information and Software Technology*, vol. 53, no. 6, pp. 638–651, 2011.

[595] D. Ardagna, C. Ghezzi, and R. Mirandola, *Rethinking the Use of Models in Software Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5281, book section 1, pp. 1–27.

[596] H. Inoue, Y. Li, and S. Mitra, "VAST: Virtualization-assisted concurrent autonomous self-test," in *IEEE International Test Conference (ITC)*, 2008, pp. 1–10.

[597] J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan, "User guidance of resource-adaptive systems," in *3rd International Conference on Software and Data Technologies (ICSOFT)*, J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, Eds., 2008, pp. 36–44.

[598] J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan, *A Software Infrastructure for User-Guided Quality-of-Service Tradeoffs*, ser. Communications in Computer and Information Science. Springer, 2009, vol. 47, pp. 48–61.

[599] C. Landauer, "Abstract infrastructure for real systems: Reflection and autonomy in real time," in *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, 2011, pp. 102–109.

[600] D. Menasce, H. Gomaa, S. Malek, and J. P. Sousa, "Sassy: A framework for self-architecting service-oriented systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, 2011.

[601] D. Perez-Palacin, R. Mirandola, and J. Merseguer, "Enhancing a qos-based self-adaptive framework with energy management capabilities," 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. 165–170.

[602] S. S. Andrade and R. J. De A Macedo, "A search-based approach for architectural design of feedback control concerns in self-adaptive systems," in *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2013, pp. 61–70.

[603] C. Sandionigi, D. Ardagna, G. Cugola, and C. Ghezzi, "Optimizing service selection and allocation in situational computing applications," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 414–428, 2013.

[604] S. S. Andrade and R. J. D. Macedo, "Do search-based approaches improve the design of self-adaptive systems ? a controlled experiment," in *28th Brazilian Symposium on Software Engineering (SBES)*, 2014, pp. 101–110.

[605] D. Perez-Palacin, R. Mirandola, and J. Merseguer, "On the relationships between qos and software adaptability at the architectural level," *Journal of Systems and Software*, vol. 87, pp. 1–17, 2014.

[606] A. Sutcliffe, *An Architecture Framework for Self-Aware Adaptive Systems*. Boston: Morgan Kaufmann, 2014, pp. 59–80.

[607] S. S. Andrade and R. J. De A Macedo, "Assessing the benefits of search-based approaches when designing self-adaptive systems: A controlled experiment," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, pp. 1–27, 2015.

[608] P. Kathiravelu and L. Veiga, "Concurrent and distributed cloudsim simulations," in *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014, pp. 490–493.

[609] S. K. Garg and R. Buyya, "NetworkCloudSim: Modelling parallel applications in cloud simulations," in *4th IEEE International Conference on Utility and Cloud Computing (UCC)*, 2011, pp. 105–113.

[610] Y. Abuseta and K. Swesi, "Towards a framework for testing and simulating self adaptive systems," in *6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2015, pp. 70–76.

[611] T. D. Nya, S. C. Stilkerich, and P. R. Lewis, "A modelling and simulation environment for self-aware and self-expressive systems," in *IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops (SASOW)*, 2013, pp. 65–70.

[612] G. Sakellari and G. Loukas, "A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing," *Simulation Modelling Practice and Theory*, vol. 39, no. 0, pp. 92–103, 2013.

[613] D. Kliazovich, P. Bouvry, and S. Khan, "GreenCloud: A packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1263–1283, 2012.

[614] S. H. Lim, B. Sharma, G. Nam, E. K. Kim, and C. R. Das, "Mdcsim: A multi-tier data center simulation, platform," in *IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–9.

[615] A. Nunez, J. L. Vazquez-Poletti, A. C. Caminero, G. G. Castane, J. Carretero, and I. M. Llorente, "icancloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, 2012.

[616] A. Nunez, J. L. Vazquez-Poletti, A. C. Caminero, J. Carretero, and I. M. Llorente, "Design of a new cloud computing simulation platform," in *Computational Science and Its Applications - ICCSA 2011: International Conference, Santander, Spain, June 20-23, 2011. Proceedings, Part III*, B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. O. Apduhan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 582–593.

[617] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications," in *24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 446–452.

[618] W. Chen and E. Deelman, "Workflowsim: A toolkit for simulating scientific workflows in distributed environments," in *IEEE 8th International Conference on E-Science (e-Science)*, 2012, pp. 1–8.

[619] P. Kathiravelu and L. Veiga, "An adaptive distributed simulator for cloud and mapreduce algorithms and architectures," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, 2014, pp. 79–88.

[620] M. Bux and U. Leser, "Dynamiccloudsim: Simulating heterogeneity in computational clouds," *Future Generation Computer Systems*, vol. 46, pp. 85–99, 2015.

[621] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers," *Software: Practice and Experience*, vol. 47, no. 4, p. 505–52, 2016.

[622] W. Yean-Fu and C. Chih-Lung, "Load balancing job assignment for cluster-based cloud computing," in *6th International Conference on Ubiquitous and Future Networks (ICUFN)*, 2014, pp. 199–204.

[623] M. Paul and G. Sanyal, "Survey and analysis of optimal scheduling strategies in cloud environment," in *World Congress on Information and Communication Technologies (WICT)*, 2011, pp. 789–792.

[624] F. Bachmann, L. Bass, and M. Klein, "Illuminating the fundamental contributors to software architecture quality," Software Engineering Institute, Carnegie Mellon University, Report CMU/SEI-2002-TR-025, 2002.