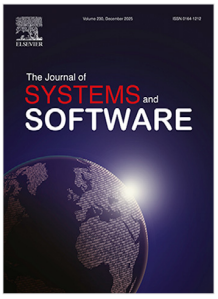
Journal Pre-proof



Architectural stability reasoning using self-awareness principles: Case of self-adaptive cloud architectures

Maria Salama, Rami Bahsoon, Rajkumar Buyya

PII: S0164-1212(26)00421-8
DOI: https://doi.org/10.1016/j.jss.2026.113188
Reference: JSS 113188

To appear in: *The Journal of Systems & Software*

Received date : 1 March 2026
Revised date : 15 October 2026
Accepted date : 9 November 2026

Please cite this article as: M. Salama, R. Bahsoon and R. Buyya, Architectural stability reasoning using self-awareness principles: Case of self-adaptive cloud architectures. *The Journal of Systems & Software* (2026), doi: https://doi.org/10.1016/j.jss.2026.113188.

This is a PDF of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability. This version will undergo additional copyediting, typesetting and review before it is published in its final form. As such, this version is no longer the Accepted Manuscript, but it is not yet the definitive Version of Record; we are providing this early version to give early visibility of the article. Please note that Elsevier's sharing policy for the Published Journal Article applies to this version, see: https://www.elsevier.com/about/policies-and-standards/sharing#4-published-journal-article. Please also note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2026 Published by Elsevier Inc.

Journal Pre-proof

Architectural Stability Reasoning using Self-Awareness Principles: Case of Self-Adaptive Cloud Architectures

Maria Salama[1], Rami Bahsoon

School of Computer Science, University of Birmingham, UK

Rajkumar Buyya

School of Computing and Information Systems, University of Melbourne, Australia

Abstract

With the increasing dependence on software, there is a pressing need for engineering long-lived software. As architectures have a profound effect on the life-span of the software and the provisioned quality of service (QoS), stable architectures are significant assets. Architectural stability tends to reflect the success of the system in supporting continuous changes without phasing-out. The *behavioural* aspect of stability is essential for seamless operation, to continuously keep the provision of quality requirements stable and prevent architecture's drifting and phasing-out. In this paper, we present a framework for reasoning about stability during runtime, leveraging self-awareness principles. The framework employs three complementary self-awareness capabilities: *goal-awareness* using runtime goals modelling for managing stability goals; *time-awareness* employing online learning for reasoning about stability in the long run; and *meta-self-awareness* using stochastic games for managing associated trade-offs. These capabilities can be deployed individually or in combination; this paper evaluates each independently against a reactive *stimulus-awareness* baseline to establish the foundational empirical evidence for their respective stability contributions. We evaluate the proposed work using the case of cloud architectures for its high dynamics during runtime. The experimental results

Email address: m.salama@cs.bham.ac.uk (Maria Salama)
[1]Corresponding author

Journal Pre-proof

Journal Pre-proof

demonstrate the feasibility of self-awareness techniques in reasoning about architectural stability during runtime operation, providing promising evidence of improvement over the reactive baseline across the chosen stability attributes.

Keywords: software architecture, cloud architecture, stability, architectural stability, self-adaptive software, self-awareness, runtime adaptation, runtime goals, reinforcement learning, Q-learning, online learning, Markov decision process, stochastic games, trade-offs management, architectural trade-offs, symbiotic simulation

1. Introduction

Modern software systems are increasingly operating in open, dynamic and uncertain environments [1]. Such challenges can have an impact on the software life-time and the quality of service (QoS) provisioned. These challenges, which are likely to persist into the foreseeable future, have motivated the need for long-lived software. An essential prerequisite for the longevity of software systems is the capability to maintain service provision with expected qualities and accommodate changes in requirements and environment.

An extensive literature survey [2] has revealed that the stability property has been considered at different levels (e.g., code, design, architecture levels) and with respect to several aspects (e.g., logical, structural, physical). This implies many different interpretations for considering stability as a software property. At the architecture level, stability has been viewed as the ability to endure with changes in requirements and the environment, while reducing the likelihood of architectural drifting and phasing-out, by avoiding ripple structural modifications (over two or more versions of the software) [3] [4]. That is an *evolutionary perspective* in considering stability, i.e., evolving the system through a number of releases [5]. Meanwhile, dynamic changes, which occur while the system is in operation, require quick and dynamic adaptations during runtime [5]. This calls for an *operational perspective* of stability that is fundamental for software architectures, to ensure seamless operation.

Journal Pre-proof

As architectures have a profound effect on the life-span of the software and QoS provision [6] [7], the architecture's behaviour tends to reflect the success of the system in constantly fulfilling end-users' requirements, as well as supporting and tolerating continuous runtime changes [8]. We argue that *architectural stability* manifests itself as a software property necessary for the operation of software systems, their dependability and longevity over time. To leverage the capabilities of software systems, it is necessary to consider *behavioural stability* to ensure that the architecture's intended behaviour is provisioned during runtime operation. This raises new questions on how to take stability-aware adaptation decisions during runtime that can keep the architecture stable in the long run.

Even though some adaptation properties have been investigated by researchers in the self-adaptive software community, such as latency of adaptation (e.g., [9] [10]), stability has not been tackled yet. On the other hand, self-awareness has been recently employed in software engineering for reasoning and engineering better adaptations [11]. But self-awareness principles have not been employed for reasoning about runtime stability.

Achieving behavioural stability for long-lived software systems calls for more intelligent reasoning about stability over the long run. We propose reasoning about stability based on self-awareness principles, enriching self-adaptive architectures with goal-, time- and meta-self-awareness capabilities, and embedding different computational intelligence techniques within these self-awareness components to support stability reasoning. The framework employs three complementary self-awareness capabilities: (i) *goal-awareness* manages stability goals using runtime goal modelling and a symbiotic relation between goals and self-awareness capabilities; (ii) *time-awareness* takes adaptation decisions converging towards stability by learning from historical information about adaptation actions and stability states; and (iii) *meta-self-awareness* manages trade-offs between stability attributes under runtime uncertainty. These capabilities can be deployed individually or in combination; this paper evaluates each independently against a reactive *stimulus-awareness* baseline—a deliberate first step

Journal Pre-proof

that provides the empirical grounding necessary before combined configurations are investigated.

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 [12] [13] [14] [15], we can evidently agree that realising stability at the architecture level should be based on the quality requirements subject to stability [12] [15] [16], where requirements are the key to long-term stability and sustainability [17] [18]. We present runtime goals modelling for stability and implement algorithms for realising symbiotic relation between runtime goals model and self-awareness.

- *Time-awareness using online learning.* Stability learning is essential for achieving stability on 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 [19] [20] [21]. 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 under runtime uncertainty. We build a runtime approach for managing trade-offs based on automatic verification of stochastic multi-player games (SMGs) using PRISM-games 2.0 [22] [23]. The approach allows reasoning about possible adaptations for multiple attributes on the long run.

- *Evaluation using the case of self-adaptive cloud architectures.* We apply the proposed framework to the case of cloud architectures, where the con-

Journal Pre-proof

tinuous satisfaction and provision of quality requirements without SLA violations in the dynamic operating environment are challenging. The self-adaptive cloud architecture is modelled and simulated by extending *CloudSim* [24]. Our work is experimentally evaluated using the *RUBiS* benchmark [25], varying the number of requests proportionally according to the *World Cup 1998* workload trend [26]. Each self-awareness capability is evaluated individually against the stimulus-awareness baseline, enabling precise characterisation of each capability's stability contribution. Experimental results demonstrate the feasibility of the proposed approaches in reasoning about stability during runtime, providing promising evidence of improvement over the reactive baseline across the chosen stability attributes; however, as the evaluation is based on a single simulation study with one workload trace, the findings should be regarded as exploratory.

Organisation. The remainder of the paper is structured as follows. Section 2 presents the relevant background, covering the main concepts, self-awareness principles and the case of self-adaptive cloud architectures. Section 3 introduces architectural stability as a software quality property. Section 4 presents the technical contributions on self-awareness techniques for stability reasoning. Section 5 reports the experimental design for evaluating the proposed approach on cloud architectures, with results presented in Section 6. Section 7 discusses the implications of the findings and threats to validity, while Section 8 reviews related work. Finally, Section 9 concludes the paper and outlines directions for future work.

2. Background

In this section, we introduce the main concepts (Section 2.1), followed by an overview of self-awareness principles (Section 2.2) which underpins the self-awareness techniques proposed for stability reasoning in this work. We then present the case of self-adaptive cloud architectures (Section 2.3), which serves as the primary evaluation context.

Journal Pre-proof

2.1. Definitions of the Main Concepts

Software Architecture. The concept of software architecture has been defined in different ways under different contexts. In our work, we adopt the definition of the ISO/IEC/IEEE Standards 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" [27]. This definition is in line with early definitions when the discipline emerged [28] [29] and with matured ones appearing later [30]. Software architectures provide abstractions for representing the structure, behaviour and key properties of a software system [29]. 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 [31] [32].

Software life cycle. The life cycle of a software system consists basically of the *development* and *operation* phases [33]. The development phase includes all activities till the decision that the software is ready for operation to deliver service, such as requirements elicitation, conceptual design, architectural design, implementation and testing [33]. The operation phase begins when the system is deployed, configured and put into operation to start delivering the actual service in the end-user's environment, cutover issues are resolved, and the product is launched [33] [27]. The former phase is known as *initial development* or *design-time*, and the latter is usually referred as *runtime*. After the development and launch of the first functioning version, the software product enters to different cycles of maintenance and evolution stages till reaching the phase-out and close-down [34] [33] [27]. During the maintenance stage, minor defects are repaired, while the system functionalities and capabilities are extended in major ways in the evolution stage [34].

Quality Attribute. The definition of a quality attribute we use is of the IEEE Standard for a Software Quality Metrics defining quality attribute as "a characteristic of software, or a generic term applying to quality factors, quality

Journal Pre-proof

sub-factors, or metric values" [35]. 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" [35].

Architecturally-significant requirements. Generally, 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) [36] [37]. Functional requirements are implemented by the individual components, while the quality requirements are dependent on the organisation and communication of these components [13]. In the software architecture discipline, the architecturally-significant requirements are considered, as not all requirements have equal effect on the architecture [38]. Architecturally-significant requirements are a subset of technically challenging requirements, technically constraining and central to the system's purpose. These requirements have significant influence on the architecture design decisions, as they should be satisfied by the architecture [38]. *Architecturally-significant functional requirements* may define the essence of the functional behaviour of the system [39], while *architecturally-significant quality requirements* are often technical in nature, such as performance targets [40] [15]. This special category of requirements, describing the key behaviours that the system should perform, plays a main role in making architectural decisions and has measurable effect on the software architecture.

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" [27]. This is determined by the state-changing operations the system can perform [27].

Self-adaptive software system. In general settings, *to adapt* means "to change a behaviour to conform to new circumstances" [41]. 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

Journal Pre-proof

functionality or performance is possible" [42] [43] [44]. 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) [45]. 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) [46]. The managed system structure could be either a non-modifiable structure or modifiable structure with/without reflection capabilities (e.g., reconfigurable software components architecture) [46]. The controller's structure is a variation of the MAPE-K loop [46].

Architectural Tactic. An architectural tactic is a design decision that influences the achievement of a quality attribute [47]. Tactics manipulate the mechanisms that mediate between a stimulus and the system's response, and are the building blocks from which architectural patterns are constructed. In the context of self-adaptive systems, tactics are operationalised at runtime as adaptation actions executed in response to violations of stability objectives.

2.2. Self-Awareness and Self-Expression

As self-adaptive software systems are increasingly becoming heterogeneous with dynamic requirements and complex trade-offs [48], 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 reinterpreted in the context of software engineering to realise autonomic behaviour for software exhibiting these characteristics [49] [50], with the aim of improving the quality of adaptation and seamlessly managing these trade-offs.

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

Journal Pre-proof

concerns are dynamically analysed and managed at runtime using knowledge from awareness.

The self-awareness framework is depicted in Figure 1. 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" [49] [50]. 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" [51]. Different levels of self-awareness, called *capabilities*, were identified to better assist the self-adaptive process [51] [50]:

- *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.

Reviewing self-awareness in software engineering [11], we found it has been employed in software systems for reasoning and engineering better adaptations with guaranteed functionalities and quality of service during runtime (e.g., [52], [49]), online service composition without human intervention [53], realising intelligent software systems with sophisticated abilities [54], and dealing with real-world situations and dynamic environments of modern software systems to respond to such fluctuating environment and associated uncertainty [55] [56].

Journal Pre-proof

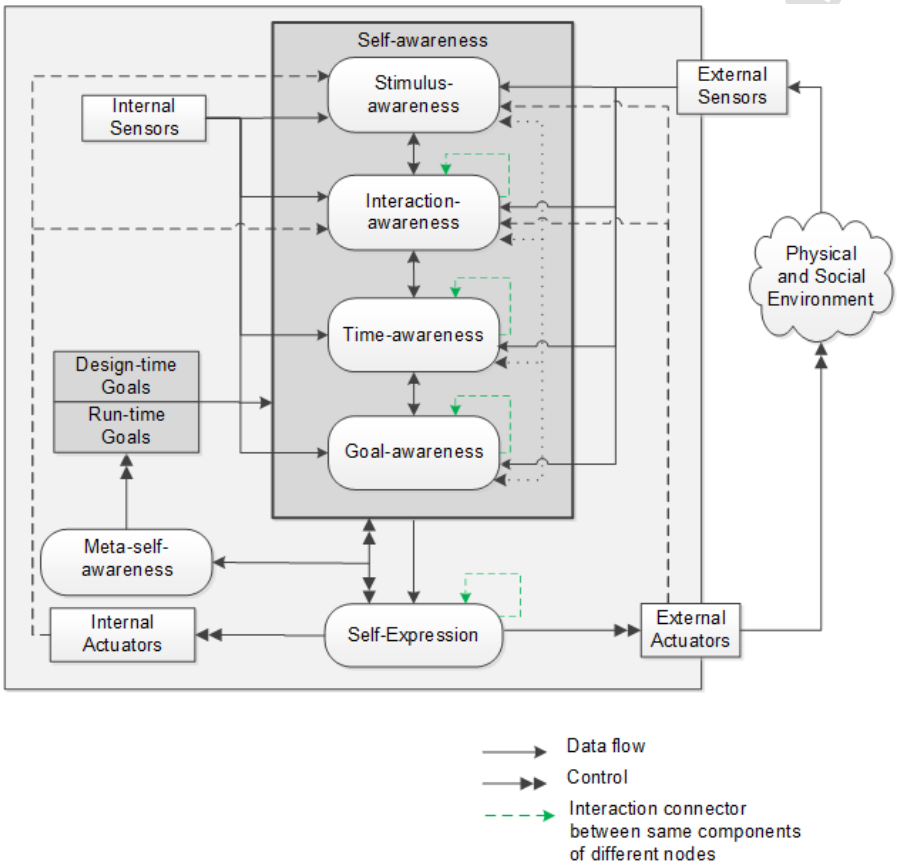


Figure 1: Self-Awareness Framework

Yet, behavioural stability has not been explored in self-adaptation using self-awareness principles.

2.3. Self-Adaptive Cloud Architectures

Cloud-based software architectures serve as the motivating case and illustrative example of this work, as they represent a canonical instance of dynamism, unpredictability and uncertainty in software systems [57]. Cloud architectures operate under continuously changing conditions—including fluctuations in workload, end-user quality requirements and peak demand events [58] [59]—and rely extensively on self-adaptation (e.g., reconfiguration, resource provisioning, re-

Journal Pre-proof

deployment) to regulate quality requirement satisfaction at runtime [60] [59].

The self-adaptation process is intended to make system behaviour converge towards intended quality requirements without SLA violation [59]; however, maintaining stability under continuous runtime adaptation remains challenging, as an unstable architecture risks degrading the system to unacceptable states [46].

The economic model of clouds, namely pay-as-you-go, imposes additional challenges: providers are required to maximise SLA profit whilst minimising operational costs [57], and face monetary penalties for SLA violations that directly affect profitability. These pressures motivate the stabilisation of provisioned quality requirements as both a technical and economic concern.

Energy consumption constitutes a further stability-relevant dimension. With increasing IT system usage and growing environmental awareness, energy consumption has emerged as a factor affecting software quality and sustainability [61]. Reducing energy consumption offers potential financial savings alongside ecological benefits [61], and environmental requirements must therefore be considered and traded off against business and financial constraints [61].

Together, these three dimensions—response time, operational cost and energy consumption—define the stability attributes evaluated in the cloud architecture case studied in this work.

3. Architectural Stability

Generally, the notion of "stability" refers to the resistance to change and the tendency to recover from perturbations. 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. As a software quality property, stability is defined in the ISO/IEC 9126 standards for software quality model [62] as one of the sub-characteristics of the maintainability characteristic of the software —along with analysability, changeability and testability —as "the capability of the software product to avoid unexpected effects from modifications

Journal Pre-proof

of the software" [62]. For general application purposes, the standard does not specify particular features or aspects of stability [63].

Reviewing the state-of-the-art on stability in software engineering [2], we have found that stability has been considered at different levels, i.e., at the code level (e.g., [64]), the requirements level (e.g., [65]), the design level ([66] [67] [68] [69]) and the architecture level ([70] [3] [71] [72]). At each level, stability has been considered in relation to several aspects from different perspectives, and thus interpreted in many ways according to the adopted perspective. For instance, stability at the code level has been interpreted as "the resistance to the potential ripple effect that the program would have when it is modified" [64], that is considering the *logical* and performance (i.e., *behavioural*) aspects of stability from the *maintenance* perspective. Design stability has been referred to "the extent to which the structure of the design is preserved throughout the evolution of the software from one release to the next" [66], where the *logical* and *structural* aspects of stability are considered from *evolutionary* perspective.

Architectural stability has been considered in terms of ripple structural modifications over two or more versions of the software, as a *structural* aspect with respect to architecturally-relevant changes carried from *evolutionary* ([70] [3]) and maintenance perspectives ([72]). This has been referred to the extent to which the architecture's structure is capable of accommodating evolutionary changes without re-designing the architecture or making ripple modifications [70] [3]. Among different perspectives, the *structural* aspect of stability is the one mostly considered at the architecture level [2].

Considering runtime dynamics of software systems, the structure and the behaviour of the software may be affected when adaptations are taking place during runtime [73]. In this context, we distinguish between the *structural* and *behavioural* aspects of stability. We also posit that an *operational* perspective (for the runtime operation of the software) is essential, different from the *evolutionary* perspective (over two or more versions of the software). The notion of stability, we are seeking, can be regarded at the *architecture* level considering the *behavioural* aspect from an *operational* perspective. As such, we define stability

Journal Pre-proof

as *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.* A stable architecture from the *operational* perspective is the one capable of continuously fulfilling the architecturally-significant quality requirements during runtime, where the architecture can return to the equilibrium state, following a perturbation due to changes in quality requirements, workload patterns or in the operational environment. Conversely, an unstable architecture is one that, when perturbed from equilibrium, will show deviation from the expected behaviour. So, stability of the architecture is essential to examine the behaviour with time following a perturbation during runtime.

Given the runtime dynamics, self-adaptive architectures need to keep the quality requirements provision stable, as well as maintain the quality of adaptation [46] [59] [8]. The former form of stability corresponds to the quality requirements desired to be delivered as adaptation goals and how constantly they are delivered without SLA violations. The latter is the degree in which the adaptation process converges towards the adaptation objective. Both aspects, mapped to each other, are used to evaluate architectural stability [46] [59] [8]. These two aspects together define what is referred to throughout this paper as *stability attributes*: the set of measurable properties used to evaluate architectural stability, encompassing both quality requirements subject to specified stability objectives and adaptation properties that characterise the adaptation process itself.

4. A Self-Awareness Assisted Framework for Reasoning about Architectural Stability

We employ the different awareness capabilities for reasoning about architectural stability. Goal-awareness incorporates a runtime goals modelling component for managing stability goals and realising the symbiotic relation between the self-awareness component and goals during runtime (discussed in Section 4.1). Time-awareness implements an online learning algorithm to assist

Journal Pre-proof

in making adaptation decisions leading to stability using historical information (discussed in Section 4.2). Meta-self-awareness is assisted by a probabilistic game-theoretic approach for managing trade-offs between different stability attributes (discussed in Section 4.3).

Within the framework, each capability can be deployed individually, as the framework is modular by design. When deployed in combination, these capabilities interact in a complementary manner: goal-awareness provides the knowledge representation layer that drives adaptation decisions, time-awareness accumulates historical information from goal satisfaction traces to inform those decisions over time, and meta-self-awareness reasons across multiple stability attributes simultaneously to resolve conflicts and synthesise optimal adaptation strategies under uncertainty. Stimulus-awareness underpins all configurations as the baseline reactive trigger mechanism, providing the monitoring and execution layer upon which the higher-level capabilities build. Whilst the framework supports both individual deployment and combined operation—as well as dynamic switching between capabilities at runtime in response to evolving operational conditions—this paper focuses on evaluating each capability individually against the stimulus-awareness baseline. This is a deliberate first step: characterising the individual stability contribution of each capability in isolation establishes the precise empirical grounding necessary to understand how capabilities may complement or interfere with one another before combined configurations are introduced. The evaluation of combined and dynamically switched capability configurations builds naturally on the individual profiles established here.

4.1. Goal-Awareness for Managing Stability Goals

Goal-Oriented Requirements Engineering (GORE) has become a widely used paradigm for eliciting, modelling, analysing and reasoning about system requirements [74] [75]. Goals are objectives to be achieved by the system under consideration [76]; that is, prescriptive statements of intent whose satisfaction requires the cooperation of different software components and elements of the system's environment [77] [55]. Goals range from high-level objectives to fine-

Journal Pre-proof

grained technical prescriptions that can be assigned as responsibilities to individual components [55] [76]. Goals therefore provide a rationale for requirements and enable low-level design and operational details to be traced back to high-level concerns [76]. This traceability makes goal models particularly relevant to self-adaptive systems, where requirements satisfaction must be assessed not only at design time but also during operation.

Runtime requirements models—denoting requirements models that are used at runtime—play a key role in supporting the monitoring of requirements satisfaction and consequent adaptation during runtime. Runtime goal models, which extend design-time goals, have been proposed to analyse the runtime behaviour of a system with respect to requirements satisfaction and to refine the goal specification model, its assumptions and operationalisation decisions [78] [79] [80]. Runtime goals have also been employed in self-adaptive software to address uncertainty [81].

We propose the *SAwGoals@run.time* component for managing stability goals. As runtime goals drive the architecture in reasoning about adaptation during runtime [82], *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) to provide fine-grained dynamic knowledge representation of stability goals, enabling effective use of the different levels of self-awareness; (ii) to monitor the satisfaction of stability goals and the performance of adaptation tactics; (iii) to support better-informed decisions regarding the optimal tactic for realising architectural stability; and (iv) to accumulate historical information that updates the knowledge available 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, trace history for time-awareness, and information about the execu-

Journal Pre-proof

tion 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 operates more effectively in the presence of historical information about the effectiveness of operationalisation decisions. In the case of instantiation, it is imperative that the designer consider what-if analysis, simulation or scenarios to test the suitability of the choice. Models that support decision-making under uncertainty may also be appropriately employed. Given relevant information about goals and the operating environment, conflict management between goals during runtime is handled by meta-self-awareness capabilities.

4.1.1. 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 quantifiable 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 functions.* $f(G)$ defines the measures for assessing levels of goal satisfaction with respect to values defined in SLAs of different end-users (e.g., objective functions for performance are response time 15 ms and 25 ms for dedicated and shared clients respectively).

Journal Pre-proof

- *Set of tactics.* $T(G) \in \mathcal{T}$ to be used in case of violation of the goal. The goal's semantics are 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 a description and informal definition of when to apply the tactic and how to execute it.
- *Object.* The element 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.* Characterises the state of the operating environment after applying the tactic.
- *Variations.* 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, the client issuing the service request r.
- *Objective function.* 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.

Journal Pre-proof

Journal Pre-proof

- *Actual value.* v, the degree of satisfaction achieved after the execution of the tactic T, as measured by the Architecture Evaluator.
- *Set of environment runtime goals.* G_e, 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, 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.

4.1.2. Algorithms for Realising Symbiotic Relation Between Runtime Goals and Self-awareness

As end-users' requirements change during runtime, there is a need to maintain the synchronisation between the goals model and the architecture [82]. 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-offs 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 2, 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.

Journal Pre-proof

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.

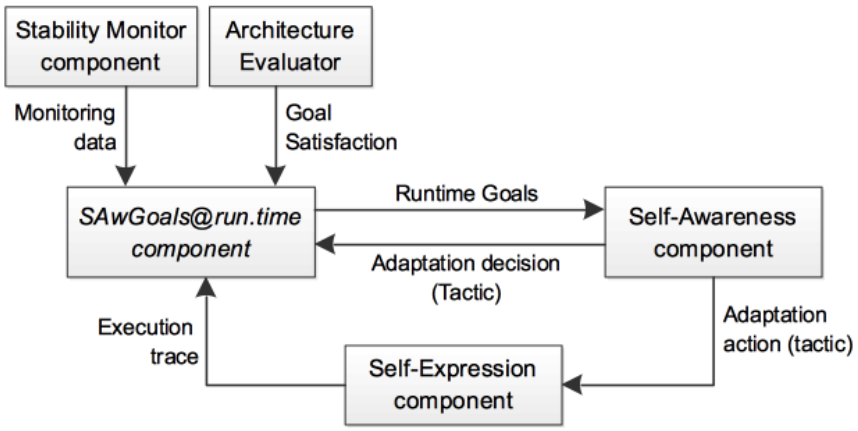


Figure 2: Symbiotic relation between Runtime Goals and Self-awareness

To realise the symbiotic relation, we provide algorithms to process the Runtime Goal Instance (Algorithm 1) and construct the Goal History (Algorithm 2).

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

Journal Pre-proof

Algorithm 1 Process Runtime 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 GoalHistory is enabled then
9:             get goal history 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
```

Tactic selection in Algorithm 1 is governed by the adaptation rules defined in the *SAwGoals@run.time* component, which associate each quality requirement with a prioritised set of tactics. When a violation of goal G is detected, the set of candidate tactics $T(G)$ is identified from the adaptation rules associated with that goal. The tactic with the highest priority that satisfies the current pre-conditions is selected for execution. Goal weights w reflect the relative importance of stability goals and are used to order the handling of concurrent violations: when multiple goals are violated simultaneously, the goal with the highest weight determines which violation is addressed first and which tactics are prioritised accordingly.

The goal history $\mathsf{H}(G)$, retrieved at line 9 of Algorithm 1 when goal history is enabled, provides the accumulated record of past tactic executions and their associated goal satisfaction outcomes across time instances. Within goal-awareness, this history serves two purposes: (i) enabling proactive checking for possible future violations by examining trends in goal satisfaction over time, and (ii) assessing the impact of previously executed tactics on goal satisfaction to inform the selection of the most effective tactic for the current violation. The history is maintained independently of time-awareness; whilst it could in principle be used by time-awareness to reason about adaptation actions in the

Journal Pre-proof

future, this combination is not considered in the current investigation.

Illustrative example. Consider a self-aware node managing two stability goals at time instance t_i: a response time goal G_1 with weight $w_1 = 0.5$ and an energy consumption goal G_2 with weight $w_2 = 0.2$. Both goals are violated simultaneously: the monitored response time exceeds its objective, and energy consumption exceeds its threshold. Since $w_1 > w_2$, the response time goal G_1 is addressed first. The adaptation rules associated with G_1 define two candidate tactics in priority order: dynamic scheduling (priority 1) and vertical scaling (priority 2). The pre-conditions of dynamic scheduling are checked first; if satisfied, it is selected and executed. If goal history $\mathsf{H}(G_1)$ is enabled, the historical record is consulted to check whether dynamic scheduling has previously improved goal satisfaction under similar conditions. If it has, the selection is confirmed. If it has not, the tactic is skipped for the current violation and the next candidate in priority order whose pre-conditions are satisfied (here, vertical scaling) is selected. The priorities defined in the adaptation rules are static and are not modified by the history; the history only determines whether the highest-priority candidate is confirmed or bypassed at selection time. If no candidate has a favourable record, the highest-priority tactic whose pre-conditions are satisfied is executed by default, so that a violation is never left unaddressed. Once the selected tactic is executed, the execution trace and resulting goal satisfaction value $v(G_1)$ are logged to $\mathsf{H}(G_1)$, so that the outcome, whether favourable or not, informs subsequent selections. The process then repeats for the energy consumption goal G_2, with its own candidate tactics evaluated in priority order.

Algorithm 2: Constructing 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.

Journal Pre-proof

Algorithm 2 Construct 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
```

4.2. 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 "Q-learning", a form of model-free reinforcement learning technique. The technique does not require a model of the system (i.e., prior 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 [19] [20] [21].

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) [19] [20] [21].

Time-awareness is characterised by its use of accumulated historical information to improve adaptation decisions; the fixed-interval trigger adopted in this work is a separate design choice. The two address different questions: the learning component determines *which* configuration to adopt, whereas the trigger determines *when* adaptation is considered. Q-learning is selected over a rule-based mechanism such as that of goal-awareness because it requires no prior

Journal Pre-proof

specification of rules or tactic priorities. It learns the mapping from observed stability states to architecture configurations (e.g., $\langle pm, vm \rangle$) from experience, and it can cope with stochastic state transitions. A rule-based mechanism, by contrast, depends on the designer anticipating the appropriate tactic for each condition in advance. The fixed interval is adopted because Q-learning updates are defined over discrete decision epochs: a regular interval ensures that each observed reward is attributable to the preceding action and that all state transitions span a comparable period, which keeps the learning process well-formed. It also bounds the adaptation overhead. The fixed interval is therefore a property of the instantiation evaluated here rather than a requirement of reinforcement learning (its consequences for stability are examined in Section 7.2).

4.2.1. Learning Model

Formally, the learning process involves a set of states S and a set of adaptation actions A, where each state $s \in S$ presents 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. The Q-Learning algorithm has a function that calculates the quality of a state-action combination:

$$Q : S \times A \rightarrow \mathbb{R} \tag{1}$$

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

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

Journal Pre-proof

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:

$$Q(s_t, a_t) \leftarrow (1-\alpha)\, Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) \right) \tag{2}$$

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 newly acquired information overrides old information, and γ is the discount factor ($0 < \gamma \leq 1$) that determines the importance of future rewards. If $\alpha = 1$, the algorithm uses only the most recent information, discarding historical values entirely; and if $\alpha = 0$, the algorithm performs no learning and uses only historical information. If $\gamma = 0$, the algorithm considers only immediate rewards; and if $\gamma = 1$, it maximises long-term cumulative reward, which is the objective in the case of stability.

4.2.2. 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.

Journal Pre-proof

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 2. Then, the new state and new reward are observed, and the Q-matrix is updated with the new reward value.

Algorithm 3 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:
  $S = \{s_1\langle rt, c, \ldots\rangle, s_2\langle rt, c, \ldots\rangle, \ldots\}$
  $A = \{a_1\langle pm, vm, \ldots\rangle, a_2\langle pm, vm, \ldots\rangle, \ldots\}$
  $Q[s][a] \leftarrow 0 \quad \forall s \in S, a \in A$
  get current state s
  while system is online **do**
    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'
    $s \leftarrow s'$
  end while
end procedure

The Q-matrix is initialised with all entries set to zero prior to the start of the simulation, reflecting the absence of prior knowledge of action quality before runtime observations begin. Continuous monitoring data for each quality requirement is mapped to a discrete state space by defining a fixed set of representative values centred on the stability objective, with additional values generated at uniform steps above and below the objective. Specifically, for each stability goal g with objective v_g and discretisation step δ_g, five discrete values are generated:

$$\{v_g - 2\delta_g,\ v_g - \delta_g,\ v_g,\ v_g + \delta_g,\ v_g + 2\delta_g\}$$

Journal Pre-proof

The full state space is the Cartesian product of the per-goal value sets. At runtime, the monitored value for each stability goal is mapped to the nearest discrete state value.

Illustrative example. Consider a node with two stability goals: response time (objective 25 ms, step $\delta = 10$ ms) and operational cost (objective \$50, step $\delta = \$10$). The discrete values are $\{5, 15, 25, 35, 45\}$ ms and $\${30, 40, 50, 60, 70\}$, respectively, giving 25 states in the Cartesian product. Let the action set contain three configurations $\langle pm, vm\rangle$: $a_1 = \langle 2, 4\rangle$, $a_2 = \langle 2, 6\rangle$, and $a_3 = \langle 3, 6\rangle$. At the scheduled adaptation instant, the monitored values are 38 ms and \$48, which map to the nearest discrete state $s = \langle 35, 50\rangle$. Suppose the current Q-values for this state are $Q(s, a_1) = 0.8$, $Q(s, a_2) = 1.4$, and $Q(s, a_3) = 1.1$. The algorithm selects a_2, the action with the highest value, and the architecture is reconfigured accordingly. At the next scheduled instant, the observed state is $s' = \langle 25, 50\rangle$ with reward $r = 1.0$, and $\max_{a'} Q(s', a') = 1.6$. With $\alpha = 0.5$ and $\gamma = 0.9$, the update is $Q(s, a_2) \leftarrow (1 - 0.5) \times 1.4 + 0.5 \times (1.0 + 0.9 \times 1.6) = 1.92$, so the evidence that a_2 led towards the stability objectives raises its value for state s. Note that no adaptation is considered between the two instants, even if the monitored response time deteriorates in the interim.

4.3. Meta-Self-Awareness for Managing Trade-offs between Stability Attributes

We investigate the use of stochastic game theory to reason about trade-offs between quality requirements and to select adaptation tactics that maximise long-term architectural welfare. The central idea is that an architectural decision — specifically, the selection of an adaptation tactic — is analogous to a game strategy, and that quality requirements with uncertain future outcomes act as underlying assets for evaluating those decisions. Rather than optimising for a single quality requirement in isolation, the proposed approach evaluates adaptation tactics for their aggregate utility across multiple stability attributes and selects tactics in a way that supports principled trade-off management under the uncertainty of the runtime environment. This provides a quantitative basis for tactic selection with respect to runtime workload uncertainty, quality

Journal Pre-proof

requirements and environmental changes, and allows reasoning about the long-term consequences of adaptation decisions for the overall stability and welfare of the architecture. Architectural tactics are intended to impart utility to the architecture [83]; here, that utility is conceived as an aggregate across multiple quality requirements rather than with respect to any single one.

4.3.1. 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-player games (SMGs) [22]. A natural fit for modelling systems that exhibit probabilistic behaviour is adopting a game-theoretic perspective [22] [84]. In particular, *stochastic games* can be 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 verification of quantitative properties in probabilistic temporal logic [84]. 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 [84]. 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 [84]:

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

PRISM-games' properties specification are written using a probabilistic temporal logic with rewards called rPATL [84] [22]. rPATL is an extension of the logic PATL [85], which is itself an extension of ATL [86], a widely used logic

Journal Pre-proof

for reasoning about multi-player games and multi-agent systems [84]. 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 [22]. rPATL is a CTL-style branching time temporal logic that incorporates the coalition operator $\langle\langle C\rangle\rangle$ of ATL [86], the probabilistic operator $\mathsf{P}_{\bowtie q}$ of PCTL [87], and the reward operator $\mathsf{R}^{r}_{\bowtie x}$ from [88] for reasoning goals related to reward/cost measures [84]. 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 choosing 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 4.3.2) and strategy synthesis (section 4.3.3). Then, we describe the model specification (in section 4.3.4).

4.3.2. Stochastic Multi-Player Game Model

We model the self-adaptive system and its environment as two players of a 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 1. (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

Journal Pre-proof

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

$$A(s) = \{a \in A | \Delta(s, a) \neq \bot\}$$

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 2. (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 $\mathcal{G}$.

where 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 3. (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 4. (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 $A(s)$.

Journal Pre-proof

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 $\lambda.s \in \Omega^+_{\mathcal{G}}$.

4.3.3. Strategy Synthesis

Reasoning about *strategies* is a fundamental aspect of SMGs model checking. rPATL queries check for the existence of a strategy that is able to optimise an objective or satisfy a given probability/reward bound [84]. Model checking also supports optimal *strategy synthesis* [84] 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 [84]. Such strategies are guaranteed to achieve the optimal expected rewards [84].

We perform strategy synthesis using *multi-objective queries* supported by PRISM-games 2.0, by computing Pareto set or optimal strategies for managing trade-offs between multi-objective properties [23]. Multi-objective queries are expressed as Boolean combination of reward-based objectives with appropriate weights [23], which allows reasoning about long-run average reward. Generally, higher weights are given to the stability of quality requirements (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).

A known limitation of probabilistic model checking approaches, including PRISM-games, is susceptibility to state-space explosion as the number of players, actions and properties grows. In the formulation adopted here, the state space of the SMG is determined by the number of architecture configuration variables (e.g., PM and VM counts and capacities), the number of adaptation

Journal Pre-proof

tactics available to the system player, and the number of stability attributes encoded as reward structures. As the number of stability attributes q and tactics $|A_{sys}|$ grows, the state space expands accordingly and verification time may become prohibitive. Mitigation strategies include compositional verification, property-directed state-space reduction and the substitution of approximate or model-free reasoning techniques, where full model checking is computationally infeasible—an option supported by the modular design of the framework (as discussed in Section 7.3).

4.3.4. Model Specification

Our formal model is implemented using PRISM-games 2.0 [84] [23]. 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 for 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:

$$[\mathtt{a}]\ \mathtt{C_a} \wedge \neg \mathtt{goal} \wedge \mathtt{t} = sys\ ->\ \mathtt{prob}_\mathtt{a}^1 :\ \mathtt{update}_\mathtt{a}^1 \wedge \mathtt{t}' = env\ +\ \ldots\ +\ \mathtt{prob}_\mathtt{a}^n :\ \mathtt{update}_\mathtt{a}^n \wedge \mathtt{t}' = env \quad (3)$$

where $\mathtt{C_a}$ is the constraints for executing the tactic $\mathtt{a}$ (e.g., increasing capacity of a physical machine (PM) to accommodate more 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 $s \in S_{sys}$. The command includes the possible updates $\mathtt{update}_\mathtt{a}^\mathtt{i}$, corresponding to one probabilistic outcome for the execution of $\mathtt{a}$, along with their associated probabilities $\mathtt{prob}_\mathtt{a}^\mathtt{i}$. And the turn is given back to the *env* player by the control variable $\mathtt{t}'$.

Journal Pre-proof

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:

$$[\mathtt{a}]\ \mathtt{C}^{\mathtt{e}}_{\mathtt{a}} \wedge \neg \mathtt{end} \wedge \mathtt{t} = env\ ->\ \mathtt{prob}^{\mathtt{1}}_{\mathtt{a}}\ :\ \mathtt{update}^{\mathtt{1}}_{\mathtt{a}} \wedge \mathtt{t}' = sys\ +\ \ldots\ +\ \mathtt{prob}^{\mathtt{n}}_{\mathtt{a}}\ :\ \mathtt{update}^{\mathtt{n}}_{\mathtt{a}} \wedge \mathtt{t}' = sys \quad (4)$$

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 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 1: Players definition in PRISM-games 2.0

```
player env environment [] endplayer
player sys system [], [] endplayer
const TURN_SYS, TURN_ENV;
global t:[TURN_SYS..TURN_ENV] init TURN_ENV;
```

Environment. The environment module (encoding shown in Listing 2) allows to obtain 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 number of VMs,

Journal Pre-proof

changing 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 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 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;
```

Journal Pre-proof

```
total_pm_cap: [1..MAX_TOTAL_PM_CAP] init 1;
[] (t=TURN_ENV)&(!end)&
        (d_vm<MAX_VM_CHANGE)&
        (d_pm<MAX_PM_CHANGE)&
        (d_vm+current_vm_num<MAX_TOTAL_VM_NUM)&
        (d_pm+current_pm_num<MAX_TOTAL_PM_NUM) ->
        (current_vm_num=current_vm_num+d_vm)&
        (current_pm_num=current_pm_num+d_pm)&
        (total_vm_cap=current_vm_num*cap)&
        (total_pm_cap=current_pm_num*cap)&
        (t'=TURN_SYS);
endmodule
```

System. The system module models the behaviour of the system, including the adaptation controller and the execution of adaptation tactics (Listing 3). This is parametrised by the constants: (i) MIN_PM_NUM and MAX_PM_NUM which 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 requirements (respnse_time, 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 [89]) to compute them based on the current architecture configuration and the request arrivals.

Listing 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;

```

Journal Pre-proof

```
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;

response_time, 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) [23], which allow expressing properties of autonomous systems that run for long periods of time and specify measures, such as energy consumption per time unit [23].

The effect of adaptation strategies on stability goals is encoded using a reward structure that assigns real-values of stability goals [23]. 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 $\mathtt{P}_{\geq 1}[\mathtt{R}(\mathtt{path})\{\text{“}\mathtt{r}\text{”}\}_{\geq \mathtt{v}}[\mathtt{S}]]$.

Journal Pre-proof

Illustrative example. Consider a game state in which the system's response time (35 ms) violates its objective of 25 ms, while energy consumption (24 kWh) is close to its 25 kWh threshold. At the environment's turn, a disturbance is introduced (e.g., a VM failure, modelled as $d_{vm} = 1$), after which the turn passes to the system. The system player can choose between two tactics. Tactic a_1 (`increase_vm_num`) reduces response time to 22 ms with probability 0.8 and to 30 ms with probability 0.2, but raises energy consumption to approximately 26 kWh. Tactic a_2 (`increase_vm_cap`) reduces response time to 24 ms with probability 0.7 and to 32 ms with probability 0.3, with energy remaining at approximately 24.5 kWh. Neither tactic dominates the other: a_1 is better for response time, whereas a_2 is better for energy. The multi-objective query $\langle\langle sys \rangle\rangle \, (\mathsf{R}\{\text{response_time}\}_{\leq 25}[\mathsf{C}] \wedge \mathsf{R}\{\text{energy}\}_{\leq 25}[\mathsf{C}])$ asks whether the system has a strategy that keeps both long-run average rewards within their targets whatever the environment does. PRISM-games explores the game, computes the Pareto set of achievable trade-offs and synthesises a memoryless deterministic strategy σ_{sys} that maps each state to a tactic. Because response time carries the higher weight, the strategy favours a_1 in states where response time is violated and energy headroom remains, and switches to a_2 in states where energy is close to its threshold, so that neither attribute is optimised at the expense of the other over the long run.

5. Experimental Design and Implementation

The main objective of the experimental evaluation is examining stability attributes when using different self-awareness capabilities for reasoning about behavioural stability, and assessing the associated overhead. In this section, we present the experimental design and implementation used to evaluate stability reasoning in self-aware cloud architectures. We first describe the architecture of the instantiated cloud node, including the stability attributes, objectives, adaptation tactics and adaptation rules used in the evaluation (Section 5.1). We then present the architecture implementation, including the simulation-based evaluation approach, the CloudSim platform and the implementation of the

Journal Pre-proof

self-awareness components (Section 5.2). Finally, we detail the experimental setup, including the evaluation strategy, benchmarks, deployment configuration and the configuration of self-awareness components (Section 5.3).

5.1. Architecture of the instantiated cloud node

Instantiated cloud node. We instantiated the architecture of a cloud node by configuring the self-aware node architecture presented in Section 2.2 for the cloud context. Figure 3 shows the instantiated cloud node, embedding the self-awareness components within a cloud datacenter equipped with a broker responsible for managing end-user requests. The cloud node represents one datacenter (with physical and virtual resources) and contains a datacenter broker responsible for managing end-user requests. To this end, this architecture dynamically performs architecture-based adaptations, which 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, except interaction-awareness, to focus on a single cloud node with no interaction with other nodes.

Stability attributes and objectives. Table 1 lists the stability attributes together with their weights, metrics and stability objectives. As described in the case of cloud architectures in section 2.3, the stability attributes, to be taken into consideration in this case (as defined in [58], [8]), include: (i) quality requirements specified in end-users SLAs, (ii) environmental restrictions, (iii) economic constraints, and (iv) quality of adaptation. Regarding the quality requirements, we consider performance (measured by response time from the time the end-user submits the request till the cloud submits the response back to the user in milliseconds). For the environmental aspect, we use the greenability property [61] [90] measured by the amount of energy consumed for running hosts in kWh. For the economic constraints, we define the operational cost by the cost of computational resources (CPUs, memory, storage and bandwidth for running VMs). Regarding the adaptation properties, we consider the frequency of adaptation (number of adaptation cycles performed) and adaptation overhead (time

Journal Pre-proof

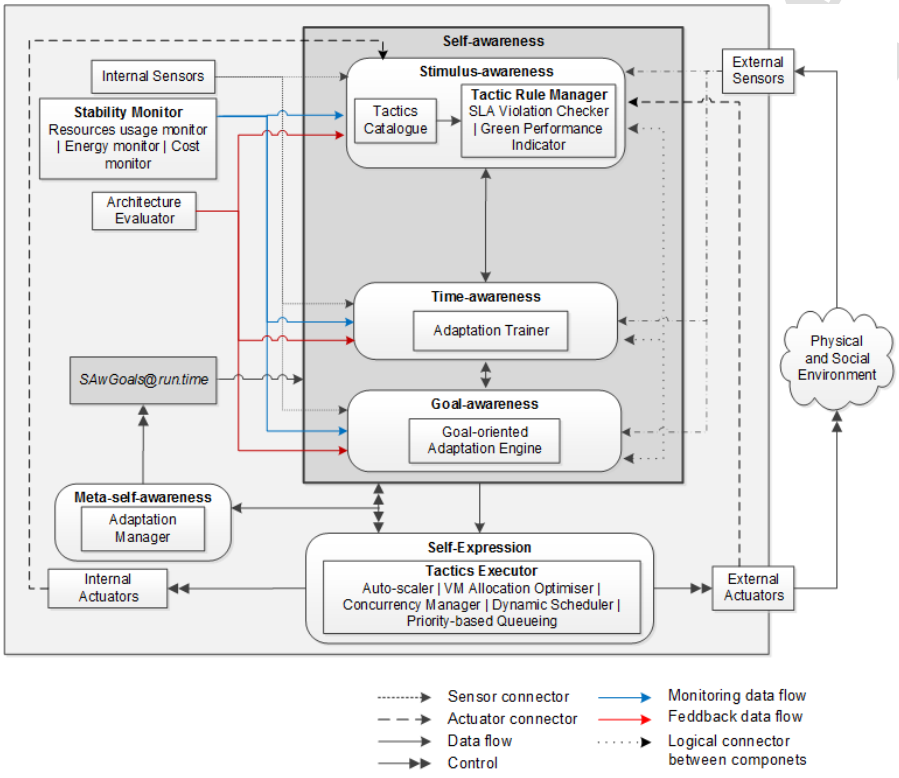


Figure 3: Architecture of the instantiated cloud node

consumed in the adaptation process) and resources overshoot (computational resources used in the adaptation process) [46] [59].

The stability objectives defined for the quality requirements — 25 ms for response time, 25 kWh for energy consumption, and $50 for operational cost — were deliberately set as demanding targets, consistent with high-stress SLA requirements, in order to stress-test the architectures beyond conditions typically encountered in production environments. These values are grounded in prior work on cloud-based software systems [58][91][92][8], rather than chosen arbitrarily. The attribute weights reflect the relative importance of the quality requirements subject to stability objectives: end-user quality requirements are given the highest weight (0.50), followed by environmental and economic constraints (0.20 each), and quality of adaptation (0.10). As these weights may vary

Journal Pre-proof

with the application scenario, domain and operational context, the framework provides architects with the flexibility to adjust both weights and objectives accordingly.

Table 1: Stability attributes used in experimental evaluation

Stability Attribute	**Objective**	**Metric**	**Weight**
Response time	25	ms	0.50
Greenability	25	kWh	0.20
Operational cost	50	$	0.20
Adaptation properties			**0.10**
Frequency of adaptation	-	cycles	
Adaptation overhead	-	ms	
Resources overshoot	-	PMs, VMs	

Adaptation tactics. Table 2 lists the adaptation tactics employed in the evaluation and their descriptions. We define the catalogue of architectural tactics to fulfil the stability attributes subject to consideration. We base this work on the architectural tactics by Bass et al. [47], including: (i) horizontal scaling (increasing or decreasing the number of physical machines), (ii) vertical scaling (increasing or decreasing the number of virtual machines or their CPU capacities), (iii) virtual machines consolidation (running the virtual machines on fewer 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), and (v) dynamic priority scheduling (scheduling policy is implemented, where the scheduler handles requests according to a scheduling policy).

Adaptation rules. Table 3 lists the adaptation rules embedded in the stimulus-awareness component, associating each tactic with its related stability attributes and priority. The priority assigned to each adaptation rule reflects the order in which tactics are attempted when a violation of the associated stability attribute is detected. When a violation occurs, the stimulus-awareness component iterates through the candidate tactics for the violated attribute in ascending priority order, selecting the first tactic whose pre-conditions are satisfied given the cur-

Table 2: Adaptation tactics used in experimental evaluation

Tactic	**Description**	**Object**	**Limits**	**Variations**
Vertical scaling	Increase the number of virtual machines (VMs) or increase their CPU capacity.	VMs	Maximum CPU capacity of the hosts running in the datacentre.	Add one or more VMs; increase the CPU capacity of running VMs.
Vertical de-scaling	Decrease the number of virtual machines (VMs) or reduce their CPU capacity.	VMs	At least one VM must remain running.	Remove one or more VMs; reduce the CPU capacity of running VMs.
Horizontal scaling	Increase the number of running hosts.	Hosts	Maximum number of hosts available in the datacentre.	Add one or more hosts.
Horizontal de-scaling	Decrease the number of running hosts.	Hosts	At least one host must remain running.	Remove one or more hosts.
VM consolidation	Shut down hosts running the fewest VMs and migrate their VMs to other active hosts.	Hosts, VMs	At least one host and one VM must remain running.	Consolidate VMs by removing one or more underutilised hosts.
Concurrency	Process different event streams using separate threads or create additional threads to process different activity sets.	Datacentre scheduler	Maximum CPU capacity of the hosts running in the datacentre.	Single-threaded or multi-threaded execution.
Dynamic scheduling	Apply a scheduling policy through which the scheduler handles requests according to runtime conditions and task priorities.	Datacentre scheduler	Maximum number of running hosts and VMs.	Earliest-deadline-first scheduling; least-slack-time scheduling; single queueing; multiple queueing; multiple dynamic queueing.

Journal Pre-proof

Journal Pre-proof

Journal Pre-proof

rent architectural state. Lower-overhead tactics are assigned higher priority—for instance, dynamic scheduling (priority 1) and concurrency (priority 2) are attempted before the more resource-intensive vertical scaling (priority 3) and horizontal scaling (priority 4) for response time violations—reflecting a design preference for minimising resource consumption whilst addressing violations. If the pre-conditions of a higher-priority tactic are not satisfied, the next tactic in the priority ordering is evaluated until a suitable tactic is found.

Table 3: Adaptation Rules used in experimental evaluation

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

5.2. Architecture implementation

Simulation-based evaluation. As research in Cloud Computing is experimental in nature [24], it is 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 [24], so that repeatable and scalable experimentation is manageable. However, the simulation results can be used to guide the selection approaches in real-world scenarios.

Simulation platform. To conduct the experimental evaluation, we implemented the architecture using the widely adopted *CloudSim* simulation platform for cloud environments [24]. The *CloudSim* simulation toolkit [24] is a general-purpose cloud simulation environment and remains among the most sophisticated discrete-event simulators for cloud systems [93]. Due to its modular architecture, it has been widely adopted and extended for modelling and simulating cloud-related problems. The simulation was built using Java JDK 1.8

Journal Pre-proof

and was run on a 2.9 GHz Intel Core i5 16 GB RAM computer.

Implementation of self-awareness components. The architecture embeds the Stability Monitor component for monitoring stability. Components necessary for checking possible violations of stability attributes are implemented in the stimulus-awareness component. Tactics are defined in the Tactics Catalogue component in the stimulus-awareness component. Management components of tactics were configured into the Tactic Executor for running the tactics, e.g., auto-scaler. The proposed self-awareness capabilities are implemented in their corresponding components: *SAwGoals@run.time* for goal-awareness, goals management and history maintenance for runtime goal reasoning, and online learning for time-awareness. Regarding the trade-offs management, we implemented the case in PRISM-Games 2.0.beta3 running on the same machine with OS X10.13.4. The outcome strategies were then exported to the simulator for performing adaptations.

5.3. Experimental Setup

Evaluation strategy. The experimental design evaluates each self-awareness capability independently against the stimulus-awareness baseline. This isolated evaluation strategy was adopted to characterise the individual stability contribution of each capability without confounding effects from capability interactions, thereby establishing a clear empirical baseline for each configuration. The evaluation of combined capability configurations, in which two or more self-awareness capabilities operate simultaneously or are switched dynamically at runtime, is beyond the scope of the current study.

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 [25] and the *World Cup 1998* trend [26] in our experiments. The *RUBiS* benchmark [25] 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

Journal Pre-proof

(read and write intensive services, e.g., PutBid, RegisterItem, RegisterUser). To fit the simulation parameters, we mapped the different services of the *RUBiS* benchmark into Million Instructions Per Second (MIPS), as listed in Table 4. To simulate a realistic workload within the capacity of our testbed, we varied the number of requests proportionally according to the *World Cup 1998* workload trend [26]. We compressed the trend such that the fluctuation of one day (86,400 seconds) in the trend corresponds to one time instance of 864 seconds in our experiments. This setup can generate up to 700 parallel requests during one time instance, which is sufficient to challenge stability. The fluctuation of the workload is depicted in Figure 4.

Table 4: Types of service requests used in experimental evaluation

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	3	70% browsing, 30% bidding	12,000
composition of the two	4	50% browsing, 50% bidding	15,000
service patterns	5	30% browsing, 70% bidding	17,000

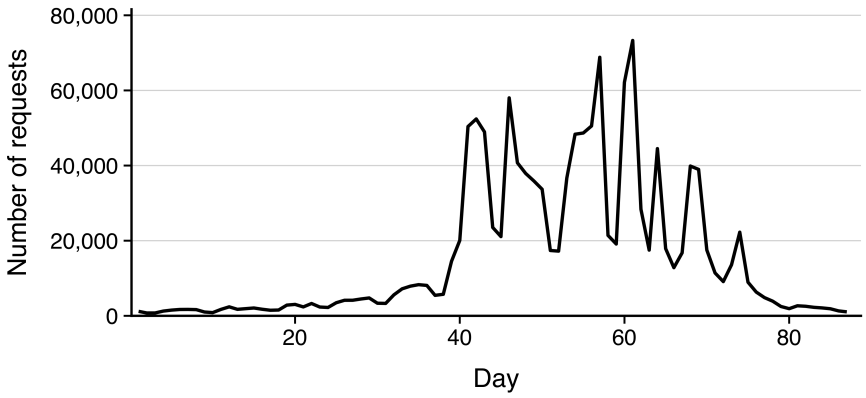


Figure 4: The *World Cup 1998* workload trend

Initial deployment configuration. The initial deployment configuration is summarised in Table 5. The initial deployment consists of 10 running hosts IBM

Journal Pre-proof

x3550 server, each with the configuration of 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 per core [94] and their energy consumption is calculated using power models of [94]. 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 [95]. 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.

Table 5: Initial deployments used in the experimental evaluation

Configuration	
No. of hosts	10 running (max. 1000)
Host type	IBM x3550 server
Host Specs	2 x Xeon X5675 3067 MHz, 6 cores, 256 GB RAM
No. of VMs	15
VMs type	General Purpose Amazon EC2 Instances m4.large, m4.xlarge, m4.2xlarge
VMs Capacity	5 x 2 core CPU 8 GB RAM, 5 x 4 core CPU 16 GB RAM, 5 x 8 core CPU 32 GB RAM

Time-awareness configuration. The Q-learning algorithm underpinning time-awareness was configured with a learning rate of $\alpha = 1$ and a discount factor of $\gamma = 0.8$. The learning rate $\alpha = 1$ corresponds to a fully online, non-averaging update regime in which each Q-matrix update is based entirely on the most recently observed reward. The discount factor $\gamma = 0.8$ reflects a strong but discounted preference for long-term reward accumulation over immediate rewards, consistent with the stability objective of maintaining quality requirements provision over the full simulation period. The discretisation step δ_g was set to 20 ms

Journal Pre-proof

for response time, 15 kWh for energy consumption and $10 for operational cost, yielding five discrete values per quality requirement centred on the respective stability objective and a total state space of $5 \times 5 \times 5 = 125$ states.

Meta-self-awareness configuration. The meta-self-awareness component was configured with the three quality requirements and seven adaptation tactics, defining a bounded architecture configuration space within which the PRISM-games model remains tractable. The computational cost of model checking was not systematically measured as part of this evaluation; the characterisation of verification time and state-space size across varying experimental configurations warrants dedicated investigation.

6. Results and Analysis

We report and analyse the experimental results over 30 runs per service request type. Stability attributes were examined at each time interval (864 seconds) whilst executing the full workload. We compare the three enhanced self-aware architectural configurations—goal-awareness, time-awareness, and meta-self-awareness—against the stimulus-awareness configuration, which serves as the baseline. This baseline configuration represents a conventional reactive self-adaptive MAPE architecture, in which adaptation is triggered by monitored stimuli without additional self-awareness reasoning [96]. We first present the overall stability outcomes across response time, energy consumption and operational cost (Section 6.1). We then analyse the temporal dynamics of these stability attributes across the simulation period (Section 6.2). Next, we examine response time stability in detail using distribution statistics and violation behaviour (Section 6.3). Finally, we evaluate adaptation behaviour in terms of adaptation frequency and overhead (Section 6.4).

6.1. Overall Stability Attributes

Figure 5 provides an overview of average stability attributes across all architectural configurations and service request types, presenting the mean values of each stability attribute. Together, the three subplots establish the high-level

Journal Pre-proof

ranking of architectural performance before subsequent sections present detailed results.

Response time. Overall, the self-awareness capabilities outperformed the stimulus-awareness baseline in maintaining response time (the highest-priority attribute) within the stability objective. The stimulus-awareness baseline configuration records the highest average response time across all service types, ranging from approximately 39 ms (S#5) to 48 ms (S#1) — exceeding the 25 ms stability objective in every case and producing violation rates between 25.5% and 46.3%. All three self-aware configurations improve upon this baseline. Meta-self-awareness configuration reduces the overall average to approximately 27 ms, converging toward the objective from an elevated starting point of 33 ms at S#1. Goal-awareness and time-awareness configurations achieve overall averages of approximately 24 ms and 22 ms respectively, representing the best improvements over the baseline. However, the numerical similarity between goal-awareness and time-awareness at the average level conceals fundamentally different distributional profiles (as demonstrated in sections 6.2 and 6.3).

Energy consumption. All architectural configurations, including the stimulus-awareness baseline configuration, exceed the 25 kWh energy objective (a deliberately challenging objective) on average over the full simulation period. Goal-awareness configuration records the highest average energy consumption (approximately 28.5 kWh) and the highest violation rates (45.5%–46.6%), driven by its elevated adaptation frequency, examined further in section 6.4. Time-awareness configuration achieves the lowest average energy expenditure (approximately 22.9 kWh) and the lowest violation rates (36.4%–37.5%), outperforming the baseline (approximately 24–25 kWh) on this attribute. Meta-self-awareness configuration records an average similar to the baseline. The energy profiles are highly consistent across service types, indicating that architectural configuration rather than service-type characteristics is the primary determinant of energy consumption.

Journal Pre-proof

Operational cost. Operational cost, reflecting the number of running virtual machines, remains well below the \$50 stability objective on average across all architectural configurations and service types, with a 0% cost violation rate recorded across every service type for all four architectural configurations. Whilst goal-awareness exhibits a pronounced non-monotonic cost profile across service types—peaking at approximately \$23 for S#2, as a consequence of its propensity to deploy resource-intensive adaptation actions when goal satisfaction is threatened by concentrated demand surges—this remains well within the \$50 stability objective. The stimulus-awareness baseline, time-awareness and meta-self-awareness configurations maintain a near-flat, low-cost profile below \$8 across all service types.

6.2. Temporal dynamics of Stability Attributes

Figures 6, 7 and 8 demonstrate how architectural behaviour evolves as cumulative workload intensity increases across the simulation period. For the clarity of the figures, we exclude time intervals prior to 30, as these early intervals contain relatively few requests and would otherwise skew the visual distributions (as shown in Figure 4). Figure 6 presents violin plots of response time distributions across grouped time intervals for each service request type. Figures 7 and 8 present cumulative average energy consumption and operational cost respectively.

Response time dynamics. As shown in Figure 6, all architectural configurations, including the stimulus-awareness baseline, exhibit a bimodal response time pattern across the simulation period, with mass concentrated at a lower mode representing efficient within-objective operation and an upper mode reflecting demand-driven overload. During the low-workload phase (intervals 30–39), all architectural configurations maintain response times well below the stability objective, with narrow violin bodies concentrated in the sub-10 ms range. As workload intensity escalates from interval 40 onward, architectural configurations diverge sharply.

Journal Pre-proof

The stimulus-awareness baseline configuration exhibits progressive broadening and upward extension of distributions from intervals 40–44 onwards, producing the widest and most consistently elevated violin bodies throughout the mid- and high-workload phases — the visual expression of its high standard deviations and violation rates. Goal-awareness configuration follows a broadly similar temporal trajectory but with more restrained upper-mode extension, consistent with its lower standard deviations. Meta-self-awareness configuration shows a comparable pattern with moderate distributional spread during the active workload phase.

Time-awareness configuration exhibits a qualitatively distinct temporal pattern that warrants specific attention. During intervals 30–64, it maintains an unusually compact, low-valued distribution — with medians frequently below 5 ms — substantially outperforming both the baseline and all other self-aware configurations. From the 65–69 interval group onward, however, time-awareness configuration undergoes an abrupt regime shift: violin bodies extend dramatically into the 40–80 ms range and medians rise sharply above the stability objective. This delayed but severe transition reflects the architecture's fixed-schedule adaptation mechanism, which is unable to respond when workload escalation outpaces the inter-adaptation interval. The resulting tail-heavy late-interval distributions represent the most extreme individual response time events observed across the entire experiment — worse than the baseline in the peak workload phase, despite superior typical-case performance.

Energy consumption dynamics. Energy consumption (Figure 7) increases monotonically for all architectural configurations across all service types. The stimulus-awareness baseline and meta-self-awareness configurations cluster at approximately 24–25 kWh average energy, crossing the 25 kWh threshold at approximately interval 50–55. Goal-awareness configuration crosses earliest — approximately at interval 45 — and reaches a cumulative total of approximately 37–38 kWh by interval 70, approximately 25% higher than the baseline. Time-awareness configuration crosses the threshold latest and records the lowest cu-

Journal Pre-proof

mulative totals (approximately 29–30 kWh by interval 70), representing a meaningful improvement over the baseline configuration. Profiles are highly consistent across S#1 to S#5, suggesting that the self-awareness configuration, rather than service-type characteristics, is the main factor shaping energy expenditure.

Operational cost dynamics. Operational cost trajectories (Figure 8) reveal a sharp divergence between goal-awareness configuration and all other architectural configurations, including the stimulus-awareness baseline. The baseline maintains a near-flat, low cost profile below $8 throughout the simulation for all service types — a profile matched closely by time-awareness and meta-self-awareness. Goal-awareness configuration diverges markedly for S#2, rising steeply from approximately $14 at interval 30 to over $30 by interval 70, and exhibits escalating costs for S#3, S#4, and S#5 as well. Notably, despite its elevated cost relative to the baseline, goal-awareness configuration does not breach the $50 stability objective in the temporal trajectory for any service type, with threshold violations arising only from episodic spikes rather than sustained accumulation.

6.3. Response Time Stability: Distribution Statistics and Violation Behaviour

Response time is the QoS attribute most sensitive to workload variation in the experimental setting, and the one for which architectural differences relative to the stimulus-awareness baseline are most pronounced. This section analyses the distributional statistics and violation rates of response time in detail, as depicted in Figure 9 (mean, median, and standard deviation heatmaps) and Figure 10 (violation rate heatmap) respectively.

Mean response time. The mean (shown in Figure 9a) reproduces the overall ranking established in section 6.1. The stimulus-awareness baseline configuration records the highest means across all service types (38.9–48.1 ms). All three self-aware configurations reduce the mean substantially: meta-self-awareness to 21.9–32.9 ms, goal-awareness to 20.0–28.1 ms and time-awareness to 18.4–25.6 ms. The reduction from the baseline is particularly large for time-awareness

Journal Pre-proof

(approximately 50% lower mean for several service types), confirming that self-awareness in the temporal domain provides the strongest average-case improvement in response time.

Median response time. The median (Figure 9b) reveals a markedly different picture that is critical to interpreting architectural stability. The stimulus-awareness baseline configuration records medians of 12.6–16.8 ms — paradoxically higher than those of time-awareness (3.6–8.6 ms) and meta-self-awareness (8.0–11.3 ms), but lower than goal-awareness (10.0–16.7 ms). The divergence between mean and median is most extreme for time-awareness, where the mean exceeds the median (by a factor of four to six depending on the service type). This is the statistical signature of a heavily right-skewed bimodal distribution: time-awareness configuration operates efficiently well below the stability objective for the majority of requests, but suffers periodic severe failure events that inflate the mean disproportionately. For the stimulus-awareness baseline configuration, by contrast, the mean-median gap is smaller, reflecting a more uniformly elevated distribution rather than a bimodal one.

Standard deviation. The standard deviation (Figure 9c) provides the most direct quantification of distributional instability. The stimulus-awareness baseline configuration records by far the highest values across all service types (106.6–157.9 ms), reflecting extreme response time variability that renders consistent QoS provision impossible. All three self-aware configurations substantially reduce distributional spread relative to the baseline. Meta-self-awareness configuration reduces standard deviation to 59.8–107.7 ms, though its S#1 value (107.7 ms) remains comparable to the baseline and is an order of magnitude greater than its own median (8.6 ms) — indicating severe unpredictability despite an apparently acceptable typical-case performance. Time-awareness configuration reduces to 55.5–66.3 ms, driven by catastrophic tail events rather than general variability. Goal-awareness configuration achieves the greatest reduction, with the lowest standard deviations of any configuration (29.7–36.3 ms), indicating that its violations, whilst more frequent than those of time-awareness,

Journal Pre-proof

are bounded and therefore more operationally predictable than those of either the baseline configuration or the other self-aware configurations.

Violation rates. Figure 10 presents the proportion of requests that violated the response time stability objective across architectural configurations and service request types. The stimulus-awareness baseline configuration records the highest rates across all service types (25.5%–46.3%): in more than one in three requests on average, the baseline fails to meet the stability objective, with the peak rate of 46.3% for S#2 approaching every other request. Goal-awareness configuration reduces violation rates relative to the baseline but remains elevated (17.2%–33.7%), with particularly high values for S#2 (32.2%) and S#5 (33.7%). Meta-self-awareness configuration achieves a lower violation rate range (15.7%–18.7%), indicating consistent improvement over the baseline across all service types. Time-awareness configuration achieves the greatest reduction from the baseline (8.8%–14.7%), with best-case rates of 8.8% for S#1 and S#3. However, this numerical advantage is purchased at the cost of severe localised instability during peak workload phases — a mode of failure absent in both the baseline and the other self-aware architectural configurations (as presented in section 6.2).

Taken together, the statistical and violation rate analyses reveal that response time stability cannot be adequately characterised by either violation rates or distributional spread alone. A two-dimensional view — encompassing both the frequency of violations and the severity and predictability of their distribution — is necessary to distinguish, for example, goal-awareness configuration (higher violation rate, bounded distribution) from time-awareness (lower violation rate, severe tail events), and to assess the extent to which self-aware architectural configurations genuinely improve upon the reactive stimulus-awareness baseline configuration.

6.4. Adaptation Behaviour: Frequency and Overhead

Given the direct impact of adaptation frequency on architectural stability, we evaluate the number of adaptation cycles taken by each architectural con-

Journal Pre-proof

figuration. Figure 11 presents the average adaptation frequency (number of adaptation cycles triggered per run) and average adaptation overhead (the total time of the adaptation process in ms) across architectural configurations and service request types. These properties directly underpin the stability patterns observed in previous sections, and explain why self-aware architectural configurations improve upon — or in some respects diverge from — the stimulus-awareness baseline.

Adaptation frequency. As shown in Figure 11a, the stimulus-awareness baseline configuration triggers approximately 104 adaptation cycles per run on average — a frequency comparable to goal-awareness (approximately 126 cycles) and meta-self-awareness (approximately 119 cycles) configurations, both of which adapt more frequently as their self-aware logic detects goal deviation or meta-level instability. Time-awareness configuration records a substantially lower frequency of approximately 46 cycles on average, reflecting its time-triggered design in which adaptations are initiated only at fixed intervals rather than in continuous response to monitored conditions. Frequency peaks at S#2 for goal-awareness (approximately 135 cycles) and meta-self-awareness (approximately 131 cycles) configurations, the service type that produces the highest violation rates for both architectural configurations, suggesting that S#2's demand profile is particularly challenging for condition-triggered adaptation mechanisms.

Adaptation overhead. As shown in Figure 11b, the stimulus-awareness baseline configuration incurs an adaptation overhead of approximately 124 ms per cycle — comparable to goal-awareness (approximately 128 ms), the highest of all architectural configurations. Meta-self-awareness configuration records a moderate overhead of approximately 104 ms, reflecting the additional introspective processing required by its meta-level reasoning layer. Time-awareness configuration achieves the lowest overhead at approximately 75 ms, contributing to its energy and cost efficiency. Overhead values are stable across service types for all architectural configurations, confirming that overhead is an intrinsic property of the adaptation mechanism rather than a workload-dependent variable.

Journal Pre-proof

The product of frequency and overhead directly explains the energy and cost profiles presented in section 6.2. The stimulus-awareness baseline configuration combines high frequency with high overhead, yet still produces the worst response time stability — confirming that the volume of adaptation activity alone is not sufficient for stability, and that the stimulus-driven trigger mechanism is poorly matched to the temporal dynamics of the *World Cup 1998* workload, where demand surges can escalate faster than reactive feedback loops can compensate. Goal-awareness configuration incurs even higher resource expenditure than the baseline by adding self-awareness to an already expensive adaptation cycle, but its goal-oriented feedback mechanism succeeds in bounding response time deviations more effectively. Time-awareness configuration achieves energy and cost efficiency through fewer, lower-overhead adaptations, but the inflexibility of its fixed schedule produces the catastrophic late-interval instability documented in section 6.2. Meta-self-awareness configuration moderates both frequency and overhead relative to goal-awareness, and reduces standard deviation substantially relative to the baseline, but does not fully resolve the predictability challenges inherent in the reactive trigger it inherits.

7. Discussion and Threats to Validity

This section summarises the experimental findings (Section 7.1), discusses their implications for self-aware architectural design (Section 7.2), and examines the customisation and applicability of the proposed framework (Section 7.3), followed by an analysis of threats to the validity of the evaluation (Section 7.4).

7.1. Summary of Findings

The results characterise the four architectural configurations along two primary dimensions of stability: the provision of quality requirements against specified stability objectives, and the properties of the adaptation process itself. Self-awareness capabilities improve upon the reactive baseline on average, but the degree and nature of improvement are highly sensitive to the chosen self-awareness capabilities.

Journal Pre-proof

Stimulus-awareness. Stimulus-awareness is the weakest performer overall, failing primarily on response time—the highest-priority stability attribute. It records the highest violation rates across all service types (25.5%–46.3%) and the highest standard deviations (106.6–157.9 ms). Although its energy consumption (approximately 24–25 kWh) and operational cost (below $8 with a 0% cost violation rate) are moderate, these do not offset its fundamental instability on the primary attribute. The underlying cause is structural: a reactive trigger mechanism is ill-matched to the bursty dynamics of the *World Cup 1998* workload, where demand surges escalate faster than reactive feedback loops can compensate.

Time-awareness. Time-awareness is the strongest performer on aggregate metrics but the most dangerous in worst-case behaviour. It achieves the lowest response time violation rates (8.8%–14.7%), the lowest energy consumption (approximately 22.9 kWh) with violation rates of 36.4%–37.5% and a near-flat operational cost profile—yet its fixed-schedule mechanism produces the most extreme individual events observed across the entire experiment. These typical-case values are deceptive: the underlying distribution is heavily right-skewed and bimodal, with the mean exceeding the median by a factor of four to six. An additional contributing factor is knowledge maturity: early-interval decisions, made when the Q-matrix has not yet converged, are substantially weaker than those made in later intervals, driving the abrupt regime shift observed from interval 65–69 onward.

Meta-self-awareness. Meta-self-awareness achieves consistent but modest improvement over the baseline. Its response time violation rates (15.7%–18.7%) are reliably lower than the baseline, but its standard deviations remain high (59.8–107.7 ms)—indicating that meta-level reasoning moderates but does not resolve the unpredictability introduced by the reactive trigger it inherits. Energy and cost profiles offer no meaningful improvement over the baseline.

Journal Pre-proof

Goal-awareness. Goal-awareness is the most predictably stable architecture, though at the highest resource cost. Whilst its response time violation rates (17.2%–33.7%) are higher than those of time-awareness and meta-self-awareness, it achieves the lowest standard deviations of any self-awareness capability (29.7–36.3 ms)—confirming that its failures are bounded and proportional, a property of greater operational value than a lower violation rate accompanied by severe exceedances. The cost of this predictability is elevated energy consumption (approximately 28.5 kWh) with violation rates of 45.5%–46.6% and the highest operational costs of any self-awareness capability with a pronounced non-monotonic cost profile peaking at approximately $23 for S#2. Whilst goal-awareness records the highest operational costs across all architectures, all values remain well within the $50 cost stability objective, resulting in a 0% cost violation rate across all service types.

7.2. Implications for Self-Aware Architectural Design

The observed stability profiles carry distinct implications for the design of self-aware architectural configurations, particularly with respect to the choice of adaptation trigger mechanism, the role of knowledge accumulation and the trade-offs between adaptation overhead and stability predictability. The results reported here reflect specific algorithmic choices and implementation decisions for each self-awareness capability; alternative implementations may yield materially different stability profiles.

Adaptation trigger strategies. Adaptation activity alone is not sufficient for stability. Stimulus-awareness combines high adaptation frequency (approximately 104 cycles per run) with high per-cycle overhead (approximately 124 ms), yet produces the worst response time stability of any architectural configuration. This confirms that a reactive trigger—one that responds to monitored conditions without anticipatory or goal-oriented reasoning—is structurally ill-suited to workloads characterised by rapid, concentrated demand surges. The choice of trigger is therefore at least as consequential as adaptation frequency: a poorly matched trigger produces diminishing returns regardless of how often it fires.

Journal Pre-proof

The three self-awareness capabilities each employ a distinct trigger mechanism that shapes their stability profile: goal-awareness uses a condition-triggered mechanism driven by continuous goal satisfaction monitoring, enabling it to respond promptly to violations at the cost of elevated adaptation frequency; time-awareness uses a fixed-interval trigger that achieves resource efficiency through scheduled adaptation cycles but is unable to respond to demand surges that occur between intervals; and meta-self-awareness inherits the reactive trigger of stimulus-awareness, augmenting it with probabilistic game-theoretic reasoning to manage trade-offs, but remaining subject to the same responsiveness limitations as the reactive baseline. The choice of trigger should therefore be informed by the temporal dynamics of the expected workload and the relative priority of stability attributes, as discussed in the self-awareness configuration guidance below.

Goal-oriented adaptation. Goal-awareness achieves the most predictable stability profile of any architecture: the lowest response time standard deviations (29.7–36.3 ms), bounded and proportional failures, and a 0% cost violation rate across all service types—making it the most operationally reliable configuration evaluated. These advantages stem from its goal-oriented feedback mechanism, which continuously monitors goal satisfaction and selects tactics accordingly, bounding response time deviations even under concentrated demand surges. However, this predictability comes at the cost of elevated resource expenditure: its higher adaptation frequency (approximately 126 cycles) and overhead (approximately 128 ms) translate into the highest resource consumption of any architecture. Goal-awareness is therefore well-suited to settings where operational predictability and bounded SLA violations are prioritised over resource efficiency, and is shaped by the specific instantiation evaluated here—the *SAw-Goals@run.time* component, which realises a symbiotic relation between runtime goal models and self-awareness capabilities through fine-grained dynamic knowledge representation and continuous history accumulation.

Journal Pre-proof

Time-triggered learning. Time-awareness achieves the most favourable aggregate stability profile of any architectural configuration: the lowest response time violation rates (8.8%–14.7%), the lowest energy consumption and the lowest adaptation overhead—making it the most resource-efficient configuration evaluated. These advantages stem directly from its fixed-interval trigger, which initiates fewer, lower-overhead adaptation cycles than the condition-triggered architectures. However, time-awareness also illustrates the risks of inflexible adaptation scheduling combined with immature knowledge. Its fixed-interval trigger, whilst achieving energy and cost efficiency through approximately 46 cycles with approximately 75 ms overhead, renders it unable to respond when workload escalation outpaces the inter-adaptation interval. A critical compounding factor is knowledge maturity: the Q-learning algorithm underpinning time-awareness operates as an online learner, and solution quality is substantially lower during early intervals—before the Q-matrix has converged—than during later intervals where sufficient historical information has accumulated. This interplay between knowledge maturity and solution quality is a non-trivial design challenge; whilst its implications for self-awareness fundamentals have been studied [97], its specific consequences for architectural stability warrant dedicated future investigation. Designers adopting time-triggered architectures must therefore account not only for schedule granularity, but also for the evolving quality of the knowledge underpinning adaptation decisions. The combination of a fixed-interval trigger with a learning-based decision mechanism, rather than the rule-based mechanism of goal-awareness, also merits discussion. The two components contribute differently: learning from experience yields configurations that are, on average, well matched to the stability objectives without rules being specified in advance, which accounts for the low violation rates, whereas the fixed interval leaves surges that escalate between decision epochs unaddressed until the next adaptation instant, which accounts for the tail failures. Reinforcement learning is therefore helpful under transient workloads only to the extent that the trigger allows its decisions to be applied in time. The fixed interval is a design choice that keeps learning epochs well-formed and overhead bounded, not

Journal Pre-proof

a requirement of reinforcement learning; condition-triggered learning, or using the goal-awareness trigger with time-awareness selecting the configuration, are promising alternatives that warrant further investigation (Section 9). Beyond adaptation triggering, knowledge unavailability, decay, incompleteness and interference between competing knowledge sources represent additional concerns that can materially degrade the quality of adaptation and, by extension, stability.

Meta-level reasoning. Meta-self-awareness achieves consistent and reliable improvement over the reactive baseline across all stability attributes: its response time violation rates (15.7%–18.7%) are reliably lower than the baseline across all service types, and its probabilistic game-theoretic approach—stochastic multi-player games modelled and verified using PRISM-games—enables principled management of stability trade-offs under runtime uncertainty, representing a more sophisticated reasoning strategy than those employed by goal-awareness or time-awareness individually. However, meta-self-awareness also highlights the limitations of layering meta-level reasoning atop a reactive trigger without addressing the trigger's underlying responsiveness. Its moderate adaptation frequency (approximately 119 cycles) and overhead (approximately 104 ms) reflect the additional processing imposed by its introspective reasoning layer, yet this investment does not fully resolve the distributional unpredictability that the reactive trigger produces. Furthermore, probabilistic model checking approaches, such as PRISM-games, are subject to state-space explosion as the number of stability attributes and adaptation tactics grows; whilst the experimental configuration evaluated here remains tractable, the scalability of this approach under more complex settings warrants careful consideration and compositional or approximate verification strategies may be necessary alternatives in such cases. Meta-level self-awareness therefore requires a correspondingly responsive trigger mechanism to realise its full potential for stability improvement.

Knowledge as a first-class design concern. The results collectively suggest that knowledge representation, monitoring, accumulation and management should be

Journal Pre-proof

treated as first-class concerns in self-aware architectural design. The framework evaluated here demonstrates promising feasibility in reasoning about stability at runtime, benefiting from more intelligent adaptations whilst avoiding unnecessary overhead and resource overshoot. Realising its full potential—particularly where multiple self-awareness capabilities may be combined or switched dynamically at runtime—requires additional treatment of how knowledge evolves, degrades and interacts across capability boundaries. An assurance and testing layer for the adaptation intelligence itself is a further important consideration, particularly in time-critical systems with strict service guarantees and dependability requirements.

Combined operation and capability collaboration. Whilst this paper evaluates each capability individually, the framework is configurable to support any combination of capabilities—two or more—or all three operating together. When all three capabilities are deployed in combination, they interact in a complementary and layered manner. Goal-awareness provides the foundational knowledge representation layer, maintaining a live runtime goal model that continuously monitors stability objective satisfaction and accumulates goal history. Time-awareness builds upon this layer by drawing on the accumulated goal history to inform learning-based adaptation decisions, progressively improving the quality of tactic selection as knowledge matures. Meta-self-awareness operates at the highest level, reasoning across multiple stability attributes simultaneously to resolve conflicts between competing goals and synthesise optimal adaptation strategies under runtime uncertainty, using the stability state information maintained by the lower-level capabilities.

Self-awareness configuration guidance. The stability profiles characterised in this study provide practical guidance for selecting the most appropriate capability for a given operational scenario. Goal-awareness is best suited to settings where operational predictability and bounded SLA violations are the primary concern, and where the cost of elevated adaptation frequency and resource consumption can be tolerated—for instance, in latency-sensitive services with strict

Journal Pre-proof

Table 6: Self-Awareness configuration guidance by operational scenario

Scenario	**Goal-Awareness**	**Time-Awareness**	**Meta-Self-Awareness**
Strict SLA / latency-sensitive	Recommended	Not recommended	Suitable
Resource- / energy-constrained	Not recommended	Recommended	Suitable
Multiple conflicting stability attributes	Suitable	Not recommended	Recommended
Regular / predictable workload	Suitable	Recommended	Suitable
Dynamic / shifting priorities	Suitable	Suitable	Recommended

contractual obligations. Time-awareness is best suited to settings where resource efficiency and energy conservation are prioritised, and where workload patterns are sufficiently regular to be accommodated within a fixed-interval schedule—for instance, in batch-oriented or periodically loaded services. Meta-self-awareness is best suited to settings where multiple stability attributes must be managed simultaneously under runtime uncertainty, and where principled trade-off reasoning is required—for instance, in multi-tenant environments with heterogeneous SLA profiles. Table 6 summarises the recommended deployment scenarios for each capability. Deciding which capability to activate at a particular point in time could be governed by a meta-level switching policy—itself a form of meta-self-awareness—that monitors the current operational context and selects the most appropriate capability or combination thereof. For instance, during stable low-workload phases, time-awareness may be sufficient to maintain stability efficiently; during periods of concentrated demand surges, goal-awareness may be activated to bound SLA violations; and during phases where multiple attributes are simultaneously stressed, meta-self-awareness may be engaged to manage trade-offs. The design of such switching policies is discussed further in Section 7.3 in the context of framework customisation.

Journal Pre-proof

Journal Pre-proof

7.3. Customisation and Applicability

The framework evaluated in this work is designed to be generic with respect to both the stability goals it targets and the self-awareness capabilities it employs. Below we discuss the principal dimensions along which the framework can be extended or reconfigured.

Configurable stability goals. Whilst the evaluation focuses on three stability attributes—response time, energy consumption, and operational cost—the framework is not inherently constrained to these objectives. Additional stability goals relevant to a given application domain can be incorporated, provided that appropriate metrics and objective functions are defined. Future work should, however, examine the scalability of the underlying reasoning techniques as the number of stability attributes grows and inter-attribute conflicts become more prevalent during operation.

Configurable self-awareness capabilities. The framework supports different self-awareness capabilities in a configurable manner. The evaluation has demonstrated distinct stability profiles across four architectural configurations, each implementing different reasoning techniques, and the framework can in principle be further customised to support one or multiple capabilities simultaneously, or to switch between them dynamically at runtime in response to evolving operational conditions. Such flexibility is architecturally valuable where workload characteristics or stability requirements shift over the system lifetime. Dynamic capability switching does, however, introduce non-trivial challenges with respect to knowledge representation, accumulation and management: the knowledge state associated with one capability—for instance, the Q-matrix accumulated by time-awareness—may not transfer directly to another, and coordination across capability boundaries requires dedicated treatment.

Extensibility of reasoning techniques. Where existing reasoning methods are insufficient to align the adaptation model with specific stability objectives, alternative or complementary techniques can be substituted. For instance, where

Journal Pre-proof

stochastic game-theoretic approaches may be computationally prohibitive at scale, model-free or approximation-based techniques could be adopted, provided that the resulting adaptation decisions remain consistent with the stability goal specifications. The framework's modular design—in which reasoning components are associated with specific self-awareness capabilities—facilitates such substitution without requiring wholesale architectural change.

Applicability beyond cloud environments. The framework's applicability extends beyond the cloud-based service provisioning context evaluated here. The stability definition adopted—maintaining a fixed level of operation within specified tolerances—is domain-agnostic, and the stability attributes under evaluation can be reconfigured to reflect the priorities of other application domains. Future evaluations at larger scale, including industrial deployments, will provide further evidence of the framework's practical viability and of the generalisability of the stability findings reported here.

7.4. Threats to Validity

Several threats to the validity of this work warrant consideration. We distinguish between internal validity, concerning the evaluation design and the extent to which it supports the conclusions drawn (Section 7.4.1), and external validity, concerning the generalisability of the findings beyond the evaluated setting (Section 7.4.2). Where applicable, we also describe the mitigation measures adopted.

7.4.1. Internal Validity

Subjectivity in attribute and threshold selection. The selection of stability attributes and their associated thresholds is subject to an element of subjectivity, as these choices were informed by the authors' background and domain knowledge. To mitigate this, the evaluation case was grounded in prior work [58] [91] [92] [8], which provides a well-established basis for the chosen attributes and lends confidence that the case study is representative of cloud-based software systems in practice. Furthermore, the stability thresholds adopted—25 ms for

Journal Pre-proof

response time, 25 kWh for energy consumption and $50 for operational cost—were deliberately set as demanding targets, consistent with high-stress SLA requirements, in order to stress-test the architectures beyond conditions typically encountered in production environments.

Controlled experimental setting. The experiments were conducted in a controlled environment and did not account for runtime switching between service types or dynamic changes to stability goals across different end-user profiles. Whilst this represents a departure from some real-world operational scenarios, the use of a real-world workload trend and the RUBiS benchmark provides a sufficiently realistic approximation of practical conditions. The controlled setting also permitted systematic stress-testing of architectural behaviour under concentrated, high-intensity demand—a form of evaluation that would be difficult to reproduce reliably in a live deployment.

Implementation-dependent results. The reported stability profiles reflect the specific algorithmic choices made for each capability (e.g., the Q-learning parameters and state discretisation, the goal model instantiation and the game formulation). Alternative implementations may yield different profiles; the findings should therefore be interpreted as characterising these instantiations rather than the capabilities in general.

7.4.2. External Validity

Restriction to three stability attributes. The restriction to three stability attributes is a further limitation. As discussed in Section 7.3, the framework is customisable to incorporate additional attributes; however, the scalability of the underlying reasoning techniques—particularly with respect to the management of inter-attribute conflicts—has not been evaluated beyond the current set. The choice of three attributes was additionally constrained by the readiness of the simulation toolkit to accommodate these concerns, and is consistent with their established importance in cloud-related studies.

Journal Pre-proof

Single evaluation case. The evaluation was conducted on a single case using one workload trace (*World Cup 1998*), which limits the generalisability of the findings. Whilst the dynamics of the case are sufficiently demanding to exceed conditions typically encountered in real environments, and both the simulation environment and the benchmark enabled precise control over workload parameters, allowing non-classical and extreme scenarios to be exercised systematically, the results should be regarded as exploratory rather than conclusive evidence of the approach's effectiveness. In particular, the *World Cup 1998* trace represents a specific temporal workload pattern characterised by bursty, concentrated demand surges; architectural behaviour under sustained high load, periodic or seasonal patterns, or adversarial workload conditions may differ materially from what is reported here. The findings therefore establish feasibility and provide promising directional evidence, but do not constitute sufficient grounds for broad claims of efficiency or generalisability. Future work will seek to address this limitation through evaluation against diverse workload traces and large-scale validation conducted in industrial settings.

8. Related Work

In this section, we discuss work addressing runtime goals modelling (section 8.1), learning for self-adaptation (section 8.2) and trade-offs management in self-adaptive systems (section 8.3).

8.1. Runtime Goals Modelling

Runtime requirements models—denoting requirements models that are used at runtime—play a key role in supporting the monitoring of requirements satisfaction and consequent adaptation during runtime. Runtime goal models, which extend design-time goals, have been proposed to analyse the runtime behaviour of a system with respect to requirements satisfaction and to refine the goal specification model, its assumptions and operationalisation decisions [78] [79] [80]. Runtime goals have also been employed in self-adaptive software to address uncertainty [81].

Journal Pre-proof

Seminal work related to run-time requirements modelling includes *models@run.time* and *self-explanation*. *Models@run.time* rethinks adaptation mechanisms in self-adaptive systems by leveraging model-driven engineering approaches at run time [80]. It supports requirements monitoring and control by dynamically observing run-time behaviour and maintaining a model of requirements alongside their realisation space, with the aim of monitoring requirements satisfaction and supporting unanticipated run-time changes. A related goal-oriented approach for systematically deriving architectural design from system goals was proposed by van Lamsweerde [98]. In the context of self-adaptive systems, *self-explanation* was introduced to provide an interpretation of how a system meets its requirements using goal-based models [99], focusing on explaining observed adaptation behaviour in terms of requirements satisfaction. Goal models have further been introduced as run-time entities in adaptive systems [100], context-aware systems [101] and goal-oriented models for self-aware systems-of-systems have been theoretically revisited in [102].

Whilst there has been growing research in run-time requirements engineering for self-adaptive software systems, existing approaches have limitations in enabling the self-properties—specifically self-awareness and self-expression—that are central to the class of systems considered in this paper. In particular, prior work has not addressed goals modelling explicitly for self-aware and self-expressive software systems, nor has it formalised a symbiotic relationship between run-time goal models and self-awareness capabilities. The *SAwGoals@run.time* component proposed in this paper addresses these gaps by extending the GORE model with fine-grained dynamic knowledge representation and algorithms for maintaining a live, up-to-date goal model that actively informs adaptation decisions across the levels of self-awareness.

8.2. Learning for Self-Adaptation

Learning for self-adaptation has been explored using a variety of techniques and for different purposes. Proposed approaches include learning for engineering feature-oriented self-adaptive systems [103], learning revised models for plan-

Journal Pre-proof

ning self-adaptive systems [104], modelling self-adaptive systems with Learning Petri Nets [105], and handling uncertainty using self-learning fuzzy neural networks [106]. Yerramalla et al. [107] proposed a stability monitoring approach based on Lyapunov functions for detecting unstable learning behaviour, mathematically analysing stability to guarantee that run-time learning converges to a stable state within bounded time. A reinforcement learning-based approach for planning architecture-based self-management has also been proposed [108].

Whilst these contributions have advanced learning-based adaptation in self-adaptive systems, two limitations are relevant to the present work. First, quality of adaptation — including adaptation frequency, overhead and resource overshoot — has not been incorporated as part of the stability behaviour being learnt. Second, behavioural stability as defined in this paper — the ability of the architecture to maintain a fixed level of QoS provision within specified tolerances under varying workload conditions — has not been treated as an online learning objective in its own right. The time-awareness component proposed here addresses both gaps by employing Q-learning as an online learning technique that converges towards stable QoS provision in the long run, whilst the evaluation explicitly accounts for the quality of adaptation as a first-class concern.

8.3. Trade-offs Management

Managing architectural trade-offs has attracted sustained research attention across both design-time and run-time settings. Seminal design-time contributions include the Architecture Tradeoff Analysis Method (ATAM) [109], the Cost Benefit Analysis Method (CBAM) [110], PerOpteryx [111] and the Quality-attribute-based Economic Valuation of Architectural Patterns [112]. Despite this maturity, self-adaptive architectures call for dedicated treatment: self-adaptation is primarily driven by the need to maintain multiple quality attributes simultaneously under continuously changing requirements and uncertain demand at run time, a setting that design-time analysis methods do not address.

Journal Pre-proof

Work addressing run-time trade-offs management in self-adaptive systems includes [113] [114] [115] [116] [117] [118], as surveyed in our prior work [119]. However, a prevalent limitation is the adoption of general self-adaptivity without discrete specialisation on self-* properties, and a tendency to limit consideration to two or three quality attributes rather than treating trade-offs management as a systematic, integrated concern. Cheng et al. [120] proposed a language for expressing adaptation strategies over multiple QoS objectives under uncertainty, though the approach targets human operators rather than autonomous run-time use. The most directly related prior work is that of Camara et al. [9] [10], which employs stochastic games for proactive adaptation, balancing the cost and benefit of proactive actions — the closest precedent in terms of technique to the meta-self-awareness component proposed here.

This work contributes three elements that distinguish it from the existing literature. Trade-offs management is framed explicitly as a *stability* concern, linking QoS provision objectives with quality of adaptation metrics rather than treated in isolation. Stochastic multi-player games with multi-objective queries and long-run rewards are employed to support reasoning about stability across multiple attributes simultaneously under run-time uncertainty. In addition, the trade-offs management component is situated within a broader self-awareness framework alongside goal-oriented and learning-based components, providing an integrated treatment that is largely absent from prior work and is required to address the multi-dimensional nature of architectural stability at run time.

9. Conclusion and Future Work

This paper addressed the problem of reasoning about architectural behavioural stability in self-adaptive software systems operating under dynamic and uncertain runtime conditions. Motivated by the case of cloud architectures, we proposed a framework that leverages self-awareness principles to enable architectures to maintain stable quality requirements provision in the long run. The framework contributes three stability reasoning capabilities: *goal-awareness*

Journal Pre-proof

through the *SAwGoals@run.time* component, which employs fine-grained runtime goal modelling and a symbiotic relation between runtime goals and self-awareness capabilities to drive stability-oriented adaptation; *time-awareness* employing Q-learning-based online learning that enables adaptation decisions to converge towards stability by accumulating and exploiting knowledge from historical execution; and *meta-self-awareness* using probabilistic model checking of stochastic multi-player games, which enables principled management of trade-offs between stability attributes under runtime uncertainty.

Experimental evaluation using the *RUBiS* benchmark and *World Cup 1998* workload trend demonstrated that self-awareness capabilities improve upon the reactive stimulus-awareness baseline, but that the nature and degree of improvement are highly sensitive to the chosen adaptation mechanism. These results establish the feasibility of the proposed approach and provide promising evidence of improvement across the selected stability attributes; however, as the evaluation is based on a single simulation study with one workload trace, the findings should be regarded as exploratory, and their generalisation to other workload patterns and deployment contexts requires further investigation.

Future work will investigate dynamic switching between self-awareness capabilities at runtime, with the aim of combining the complementary strengths identified here. Specifically, the investigation of combined self-awareness capability configurations—in which two or more capabilities operate simultaneously or are switched dynamically at runtime in response to evolving workload conditions—represents the primary future direction. The complementary stability profiles identified in this study—the bounded failures of goal-awareness and the energy efficiency of time-awareness—suggest that combined configurations may yield stability profiles superior to any single capability in isolation, though the knowledge coordination challenges across capability boundaries will require careful treatment. Within this direction, the trigger mechanism of time-awareness warrants particular attention: condition-triggered learning, or pairing the goal-awareness trigger with learning-based configuration selection, may address the responsiveness limitations of fixed-interval adaptation identified here.

Journal Pre-proof

A systematic sensitivity analysis of the Q-learning parameters—specifically, the learning rate α and discount factor γ—and their implications for architectural stability represents a further valuable direction for future investigation. Investigation into the computational cost and scalability of the PRISM-games-based meta-self-awareness component—specifically, the characterisation of verification time and state-space size across varying numbers of stability attributes, tactics and architectural configurations—represents a further important direction, alongside exploration of compositional and approximate verification strategies as scalable alternatives. Further investigation into the dynamics of knowledge accumulation, decay and cross-system knowledge transfer and their implications for architectural stability also represents a promising direction for future research. We also plan to extend the evaluation to additional application domains, including scientific workflows, and to conduct large-scale validation in industrial settings to assess the framework's practical viability beyond the controlled simulation environment.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by the School of Computer Science, University of Birmingham.

We would like to thank Abdessalam Elhabbash for useful discussions on self-awareness, Rafik Salama for inspiration on online learning, and David Parker for the support on PRISM-games.

Journal Pre-proof

Journal Pre-proof

References

[1] B. Chen, X. Peng, Y. Liu, S. Song, J. Zheng, W. Zhao, Architecture-based behavioral adaptation with generated alternatives and relaxed constraints, IEEE Transactions on Services Computing PP (99) (2016) 1–1.

[2] M. Salama, R. Bahsoon, P. Lago, Stability in software engineering: Survey of the state-of-the-art and research directions, IEEE Transactions on Software Engineering 47 (7) (2021) 1468–1510.

[3] R. Bahsoon, 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.

[4] R. Bahsoon, W. Emmerich, Architectural Stability, Vol. 5872 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, book section 43, pp. 304–315.

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

[6] D. Garlan, Software architecture: A roadmap, in: Conference on The Future of Software Engineering, 2000, pp. 91–101.

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

[8] M. Salama, R. Bahsoon, Analysing and modelling runtime architectural stability for self-adaptive software, Journal of Systems and Software.

[9] J. Cámara, G. A. Moreno, 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.

Journal Pre-proof

[10] J. Camara, D. Garlan, B. Schmerl, 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.

[11] A. Elhabbash, M. Salama, R. Bahsoon, P. Tino, Self-awareness in software engineering: A systematic literature review, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 14 (2).

[12] R. Kazman, L. Bass, M. Webb, 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.

[13] I. Sommerville, Software Engineering, Boston, Mass. London : Pearson Education, Boston, Mass. London, 2011.

[14] K. S. Barber, T. Graser, J. Holt, G. Baker, Arcade: Early dynamic property evaluation of requirements using partitioned software architecture models, Requirements Engineering 8 (4) (2003) 222–235.

[15] D. Ameller, C. Ayala, J. Cabot, X. Franch, Non-functional requirements in architectural decision making, IEEE Software 30 (2) (2013) 61–67.

[16] K. Angelopoulos, V. E. S. Souza, 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.

[17] C. Becker, S. Betz, R. Chitchyan, L. Duboc, S. M. Easterbrook, B. Penzenstadler, N. Seyff, C. C. Venters, Requirements: The key to sustainability, IEEE Software 33 (1) (2016) 56–65.

[18] R. Chitchyan, C. Becker, S. Betz, L. Duboc, B. Penzenstadler, N. Seyff, C. C. Venters, Sustainability design in requirements engineering: State

Journal Pre-proof

Journal Pre-proof

of practice, in: IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 533–542.

[19] C. J. C. H. Watkins, P. Dayan, Q-learning, Machine Learning 8 (3) (1992) 279–292.

[20] P. Dayan, C. J. C. H. Watkins, Reinforcement Learning: A Computational Perspective, John Wiley & Sons, Ltd, 2006.

[21] D. L. Poole, A. K. Mackworth, Artificial Intelligence: Foundations of Computational Agents, Cambridge University Press, 2010.

[22] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, A. Simaitis, Automatic verification of competitive stochastic systems, Formal Methods in System Design 43 (1) (2013) 61–92.

[23] M. Kwiatkowska, D. Parker, C. Wiltsche, PRISM-Games 2.0: A Tool for Multi-objective Strategy Synthesis for Stochastic Games, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 560–566.

[24] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50.

[25] Rice University Bidding System (RUBiS).
URL `www.rubis.ow2.org`

[26] M. Arlitt, T. Jin, A workload characterization study of the 1998 world cup web site, IEEE Network 14 (3) (2000) 30–37.

[27] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), ISO/IEC/IEEE 24765:2010(E) Systems and software engineering – Vocabulary, Report (2010).

[28] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

Journal Pre-proof

[29] M. Shaw, D. Garlan, Software Architecture: Perspectives on an emerging discipline, Prentice-Hall, Inc., 1996.

[30] L. Bass, P. Clements, R. Kazman, Software architecture in practice, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., 2003.

[31] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.

[32] C. Seo, G. Edwards, S. Malek, 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.

[33] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004) 11–33.

[34] V. T. Rajlich, K. H. Bennett, A staged model for the software life cycle, Computer 33 (7) (2000) 66–71.

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

[36] B. I. Witt, Software Architecture and Design : Principles, models, and methods, New York : Van Nostrand Reinhold, New York, 1994.

[37] H. Gomaa, Software Modeling and Design : UML, use cases, architecture, and patterns, Cambridge University Press, Cambridge, 2010.

[38] C. Lianping, M. A. Babar, B. Nuseibeh, Characterizing architecturally significant requirements, IEEE Software 30 (2) (2013) 38–45.

Journal Pre-proof

[39] P. R. Anish, B. Balasubramaniam, A knowledge-assisted framework to bridge functional and architecturally significant requirements, in: 4th International Workshop on Twin Peaks of Requirements and Architecture, ACM, 2014, pp. 14–17.

[40] R. Kazman, L. Bass, Toward deriving software architectures from quality attributes, Technical Report CMU/SEI-94-TR-010, Software Engineering Institute, Carnegie Mellon University (1994).

[41] K. J. Åström, B. Wittenmark, Adaptive control, Addison-Wesley, 1989.

[42] R. Laddaga, Self-adaptive software, Technical Report 98-12, DARPA BAA (1997).

[43] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (3) (1999) 54–62.

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

[45] A. C. Meng, On evaluating self-adaptive software, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 65–74.

[46] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, 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, pp. 80–89.

[47] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 3rd Edition, Addison-Wesley Professional, 2012.

Journal Pre-proof

[48] T. Nya, S. Stilkerich, P. R. Lewis, X. Yao, Self-aware and self-expressive systems, Awareness Magazine.

[49] P. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, 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.

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

[51] S. Parsons, R. Bahsoon, P. Lewis, X. Yao, Towards a better understanding of self-awareness and self-expression within software systems, Technical Report CSR-11-03, School of Computer Science, University of Birmingham (2011).

[52] R. Sterritt, 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.

[53] A. Elhabbash, R. Bahsoon, P. Tino, Towards self-aware service composition, in: IEEE International Conference on High Performance Computing and Communications,IEEE International Conference on Cyberspace Safety and Security, IEEE International Conference on Embedded Software and Sysemst (HPCC,CSS,ICESS), 2014, pp. 1275–1279.

[54] E. Vassev, M. Hinchey, Awareness in software-intensive systems, IEEE Computer 45 (12) (2012) 84–87.

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

Journal Pre-proof

[56] T. Chen, R. Bahsoon, Toward a smarter cloud: Self-aware autoscaling of cloud configurations and resources, IEEE Computer 48 (9) (2015) 93–96.

[57] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of Cloud Computing, ACM Communications 53 (4) (2010) 50–58.

[58] T. Chen, F. Faniyi, R. Bahsoon, P. R. Lewis, X. Yao, L. Minku, L. Esterle, The handbook of engineering self-aware and self-expressive systems, Technical report, School of Computer Science, University of Birmingham (2014).

[59] N. M. Villegas, G. Tamura, H. A. Müller, Architecting software systems for runtime self-adaptation: Concepts, models, and challenges, Elsevier (Morgan Kaufmann), Boston, 2017, pp. 17–43.

[60] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4 (2) (2009) 1–42.

[61] P. Lago, S. A. Kocak, I. Crnkovic, B. Penzenstadler, Framing sustainability as a property of software quality, Communications of the ACM 58 (10) (2015) 70–78.

[62] International Organization for Standardization and International Electrotechnical Commission (ISO/IEC), ISO/IEC 9126-1 – Information technology – Software product quality – Quality model, Report ISO/IEC 9126-1:2001, ISO/IEC (2000).

[63] L. Dobrica, E. Niemela, A survey on software architecture analysis methods, IEEE Transactions on Software Engineering 28 (7) (2002) 638–653.

[64] S. S. Yau, J. S. Collofello, Some stability measures for software maintenance, IEEE Transactions on Software Engineering SE-6 (6) (1980) 545–552.

Journal Pre-proof

[65] D. Bush, A. Finkelsteiin, Requirements stability assessment using scenarios, in: 11th IEEE International Conference on Requirements Engineering (RE), IEEE Computer Society, 2003, p. 23.

[66] S. S. Yau, J. S. Collofello, Design stability measures for software maintenance, IEEE Transactions on Software Engineering SE-11 (9) (1985) 849–856.

[67] M. E. Fayad, A. Altman, An introduction to software stability, Communications of the ACM 44 (9) (2001) 95.

[68] M. O. Elish, D. Rine, Indicators of structural stability of object-oriented designs: A case study, in: 29th Annual IEEE/NASA Software Engineering Workshop, 2005, pp. 183–192.

[69] D. Kelly, A study of design characteristics in evolving software using stability as a criterion, IEEE Transactions on Software Engineering 32 (5) (2006) 315–329.

[70] M. Jazayeri, On architectural stability and evolution, in: 7th Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag, 2002, pp. 13–23.

[71] S. A. Tonu, A. Ashkan, L. Tahvildari, Evaluating architectural stability using a metric-based approach, in: 10th European Conference on Software Maintenance and Reengineering (CSMR), 2006, pp. 10 pp.–270.

[72] A. Molesini, A. F. Garcia, C. v. F. G. Chavez, T. V. Batista, Stability assessment of aspect-oriented software architectures: A quantitative study, Journal of Systems and Software 83 (5) (2010) 711–722.

[73] A. Chomchumpol, T. Senivongse, Stability measurement model for service-oriented systems, in: 9th Malaysian Software Engineering Conference (MySEC), 2015, pp. 54–59.

Journal Pre-proof

[74] A. van Lamsweerde, Requirements Engineering: From system goals to UML models to software specifications, Wiley, 2009.

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

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

[77] A. van Lamsweerde, Requirements engineering in the year 00: A research perspective, in: 22nd International Conference on Software Engineering (ICSE), ACM, 337184, 2000, pp. 5–19.

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

[79] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, 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.

[80] G. Blair, N. Bencomo, R. B. France, Models@run.time, Computer 42 (10) (2009) 22–27.

[81] M. S. Feather, S. Fickas, A. van Lamsweerde, C. Ponsard, Reconciling system requirements and runtime behavior, in: 9th International Workshop on Software Specification and Design, 1998, pp. 50–59.

[82] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, 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.

Journal Pre-proof

[83] G. Procaccianti, P. Lago, G. A. Lewis, Green architectural tactics for the cloud, in: IEEE/IFIP Conference on Software Architecture (WICSA), 2014, pp. 41–44.

[84] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, A. Simaitis, PRISM-games: A Model Checker for Stochastic Multi-Player Games, Springer Berlin Heidelberg, 2013, pp. 185–191.

[85] T. Chen, 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.

[86] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time temporal logic, Journal of the ACM (JACM) 49 (5) (2002) 672–713.

[87] A. Bianco, L. de Alfaro, Model checking of probabilistic and nondeterministic systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 499–513.

[88] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, Automated Verification Techniques for Probabilistic Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 53–113.

[89] M. Salama, A. Shawish, R. Bahsoon, Dynamic modelling of tactics impact on the stability of self-aware cloud architectures, in: IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2016, pp. 871–875.

[90] C. Calero, M. Piattini, Introduction to green in software engineering, Springer, 2015, pp. 3–27.

[91] M. Salama, R. Bahsoon, Quality-driven architectural patterns for self-aware cloud-based software, in: IEEE 8th International Conference on Cloud Computing (CLOUD), Applications Track, New York, NY, USA, 2015, pp. 844–851.

Journal Pre-proof

Journal Pre-proof

[92] M. Salama, R. Bahsoon, A taxonomy for architectural stability, in: 31st ACM/SIGAPP Symposium on Applied Computing (SAC), Software Architecture: Theory, Technology, and Applications Track (SATTA), Pisa, Italy, 2016.

[93] S. K. Garg, R. Buyya, NetworkCloudSim: Modelling parallel applications in cloud simulations, in: 4th IEEE International Conference on Utility and Cloud Computing (UCC), 2011, pp. 105–113.

[94] A. Beloglazov, 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 24 (13) (2012) 1397–1420.

[95] Amazon Web Services, Inc., Amazon EC2 Instance Types, accessed: 2017-10-01.
URL https://aws.amazon.com/ec2/instance-types/

[96] M. Maurer, I. Breskovic, V. C. Emeakaroha, I. Brandic, Revealing the mape loop for the autonomic management of cloud infrastructures, in: IEEE Symposium on Computers and Communications (ISCC), 2011, pp. 147–152.

[97] A. Elhabbash, R. Bahsoon, P. Tino, Self-awareness for dynamic knowledge management in self-adaptive volunteer services, in: IEEE International Conference on Web Services (ICWS), 2017, pp. 180–187.

[98] A. van Lamsweerde, From system goals to software architecture, Vol. 2804 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, book section 2, pp. 25–43.

[99] K. Welsh, N. Bencomo, P. Sawyer, J. Whittle, Self-explanation in adaptive systems based on runtime goal-based models, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, book section 5, pp. 122–145.

Journal Pre-proof

[100] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, 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.

[101] M. Vrbaski, G. Mussbacher, D. Petriu, D. Amyot, Goal models as run-time entities in context-aware systems, in: 7th Workshop on Models@run.time, ACM, 2012, pp. 3–8.

[102] E. Cavalcante, T. Batista, N. Bencomo, P. Sawyer, Revisiting goal-oriented models for self-aware systems-of-systems, in: IEEE International Conference on Autonomic Computing (ICAC), 2015, pp. 231–234.

[103] N. Esfahani, A. Elkhodary, S. Malek, A learning-based framework for engineering feature-oriented self-adaptive software systems, IEEE Transactions on Software Engineering 39 (11) (2013) 1467–1493.

[104] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, K. Inoue, Learning revised models for planning in adaptive systems, in: International Conference on Software Engineering (ICSE), IEEE Press, 2013, pp. 63–71.

[105] Z. Ding, Y. Zhou, M. Zhou, Modeling self-adaptive software systems with learning petri nets, IEEE Transactions on Systems, Man, and Cybernetics: Systems 46 (4) (2016) 483–498.

[106] D. Han, J. Xing, Q. Yang, J. Li, 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.

[107] S. Yerramalla, B. Cukic, M. Mladenovski, E. Fuller, Stability monitoring and analysis of learning in an adaptive system, in: International Conference on Dependable Systems and Networks (DSN), 2005, pp. 70–79.

[108] K. Dongsun, P. Sooyong, Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software, in:

Journal Pre-proof

ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2009, pp. 76–85.

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

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

[111] A. Koziolek, H. Koziolek, R. Reussner, PerOpteryx: automated application of tactics in multi-objective software architecture optimization, in: I. Crnkovic, J. A. Stafford, D. Petriu, J. Happe, P. Inverardi (Eds.), Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS (QoSA-ISARCS), ACM, 2011, pp. 33–42.

[112] I. Ozkaya, R. Kazman, M. Klein, Quality-attribute-based economic valuation of architectural patterns, Technical Report CMU/SEI-2007-TR-003, Software Engineering Institute, Carnegie Mellon University (2007).

[113] J. Teich, From dynamic reconfiguration to self-reconfiguration: Invasive algorithms and architectures, in: International Conference on Field-Programmable Technology (FPT), 2009, pp. 11–12.

[114] R. Mirandola, 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.

Journal Pre-proof

[115] D. Menasce, H. Gomaa, S. Malek, J. P. Sousa, Sassy: A framework for self-architecting service-oriented systems, IEEE Software 28 (6) (2011) 78–85.

[116] D. Perez-Palacin, R. Mirandola, J. Merseguer, QoS and energy management with petri nets: A self-adaptive framework, Journal of Systems and Software 85 (12) (2012) 2796–2811.

[117] X. Peng, B. Chen, Y. Yu, W. Zhao, Self-tuning of software systems through dynamic quality tradeoff and value-based feedback control loop, Journal of Systems and Software 85 (12) (2012) 2707–2719.

[118] L. W. Shen, X. Peng, W. Y. Zhao, Quality-driven self-adaptation: Bridging the gap between requirements and runtime architecture by design decision, in: X. Bai, F. Belli, E. Bertino, C. K. Chang, A. Elci, C. Seceleanu, H. Xie, M. Zulkernine (Eds.), IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), International Computer Software & Applications Conference, 2012, pp. 185–194.

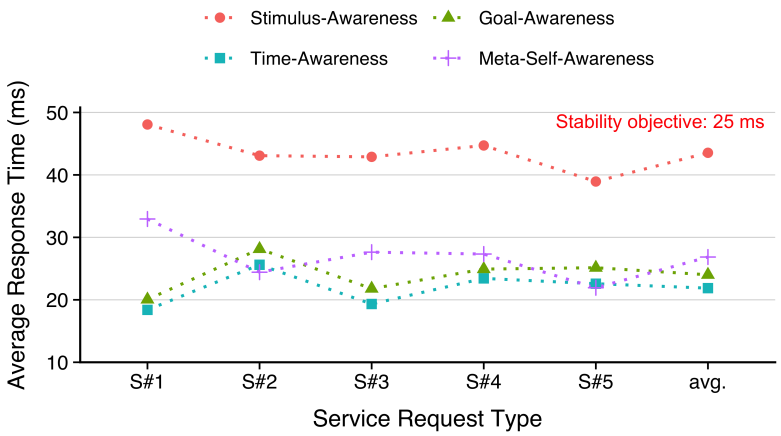
[119] M. Salama, R. Bahsoon, N. Bencomo, Managing trade-offs in self-adaptive software architectures: A systematic mapping study, Elsevier (Morgan Kaufmann), Boston, MA, 2017, pp. 249–297.

[120] S. W. Cheng, D. Garlan, Stitch: A language for architecture-based self-adaptation, Journal of Systems and Software 85 (12) (2012) 2860–2875.

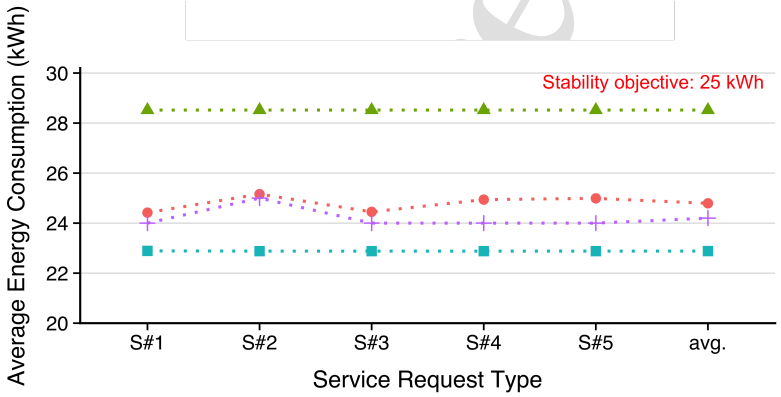
Appendix A. Online Resources

The code and data of the evaluation are available online at `https://github.com/m-salama/StabilityReasoningCloud`. The developed platform for simulating self-adaptive and self-aware cloud is also published as open-source, available at `https://github.com/m-salama/SAdSAwCloudSim`.

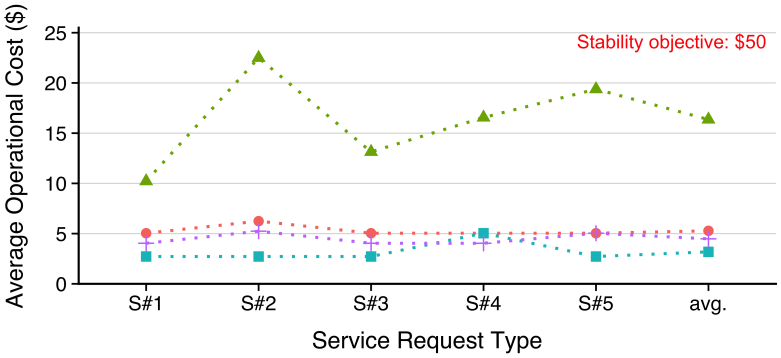
Journal Pre-proof



(a) Average response time



(b) Average energy consumption



(c) Average operational cost

Figure 5: Average stability attributes across all architectural configurations and service request types.

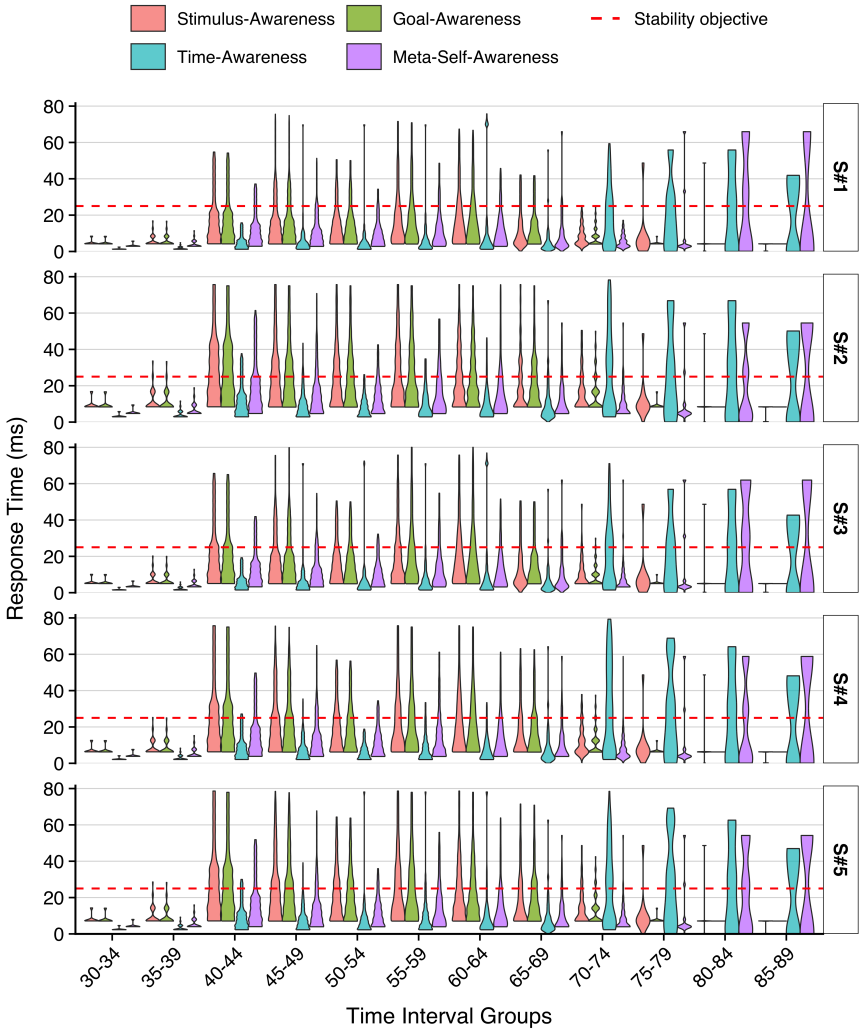


Figure 6: Temporal distribution of response time across grouped time intervals (30–89), stratified by service request type (S#1–S#5). Violin width reflects number of requests at each response time value.

Journal Pre-proof

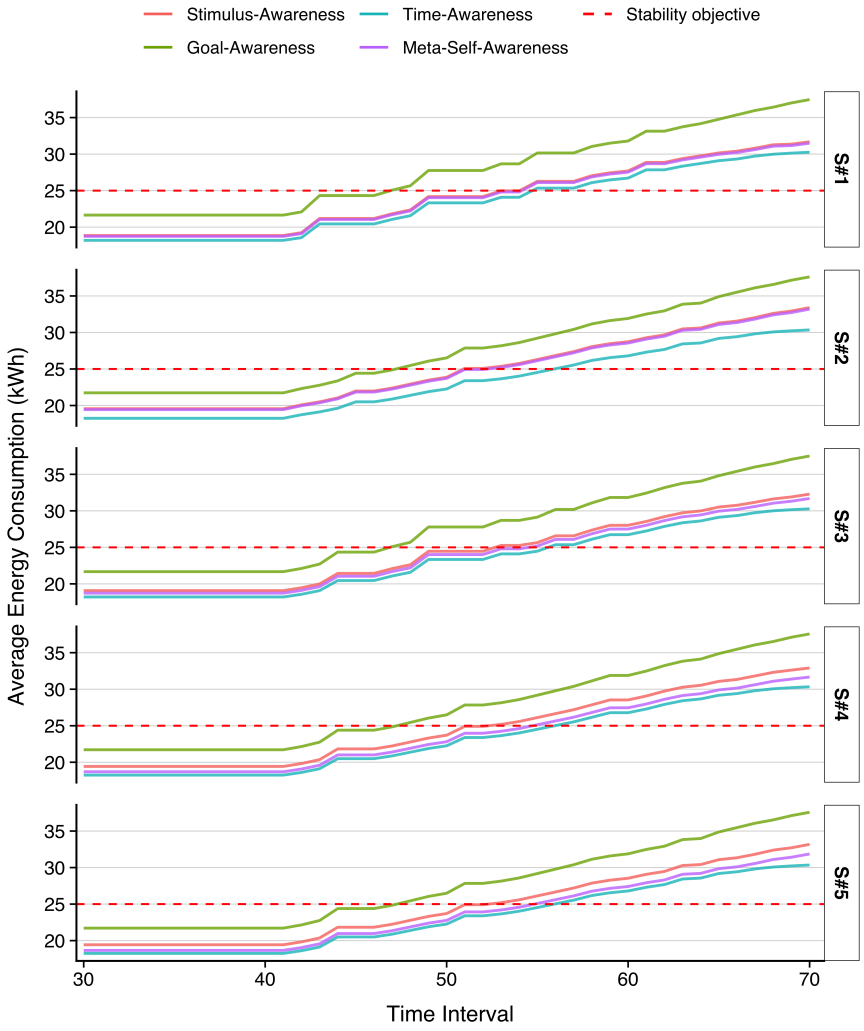


Figure 7: Cumulative average energy consumption (kWh) across time intervals (30–70), stratified by service request type.

Journal Pre-proof

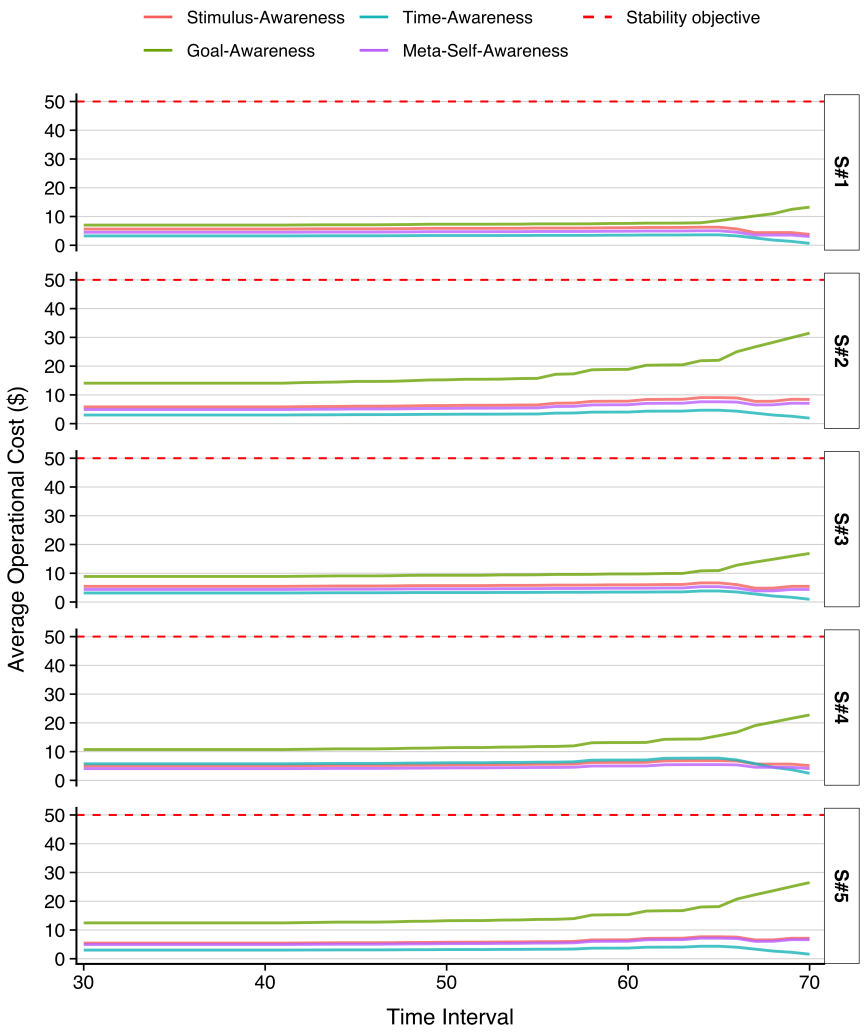


Figure 8: Average operational cost ($) across time intervals (30–70), stratified by service request type.

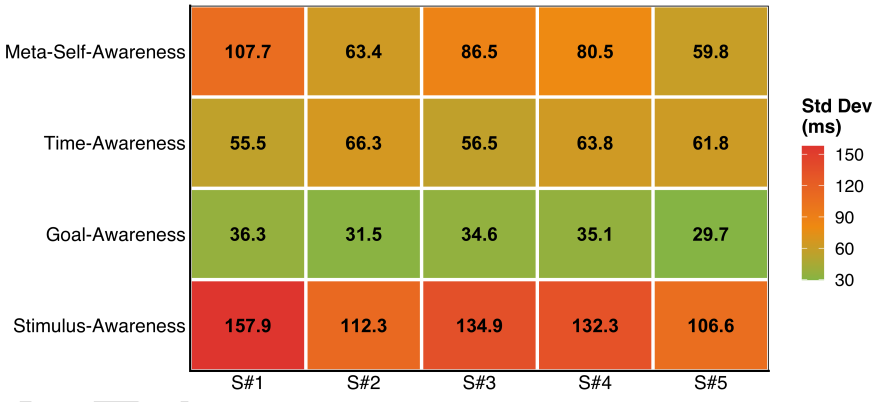
Journal Pre-proof



(a) Mean response time



(b) Median response time



(c) Response time standard deviation

Figure 9: Response time statistics across architectural configurations and service request types.

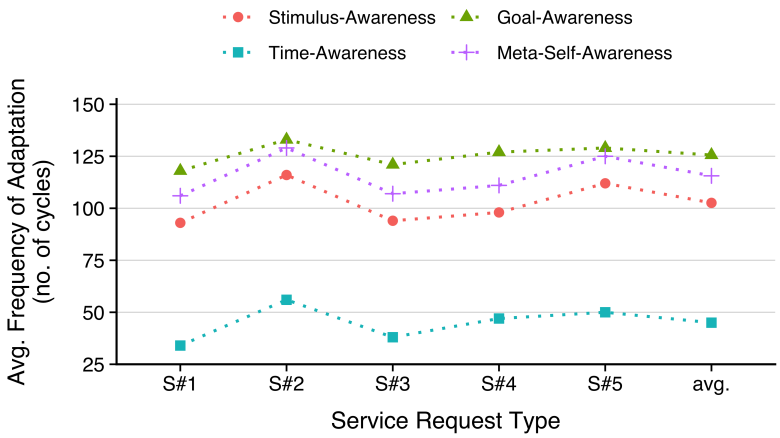
Journal Pre-proof



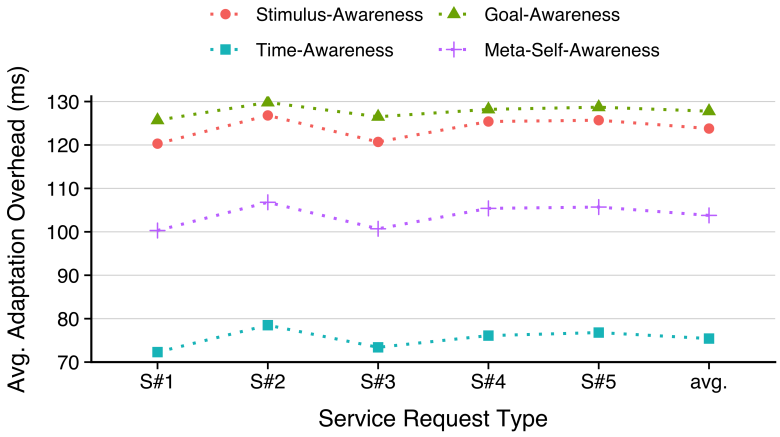
Figure 10: Average violation rate of the response time stability objective across architectural configurations and service request types.

Journal Pre-proof

Journal Pre-proof



(a) Adaptation frequency



(b) Adaptation overhead

Figure 11: Average adaptation properties across architectural configurations and service request types.

Journal Pre-proof

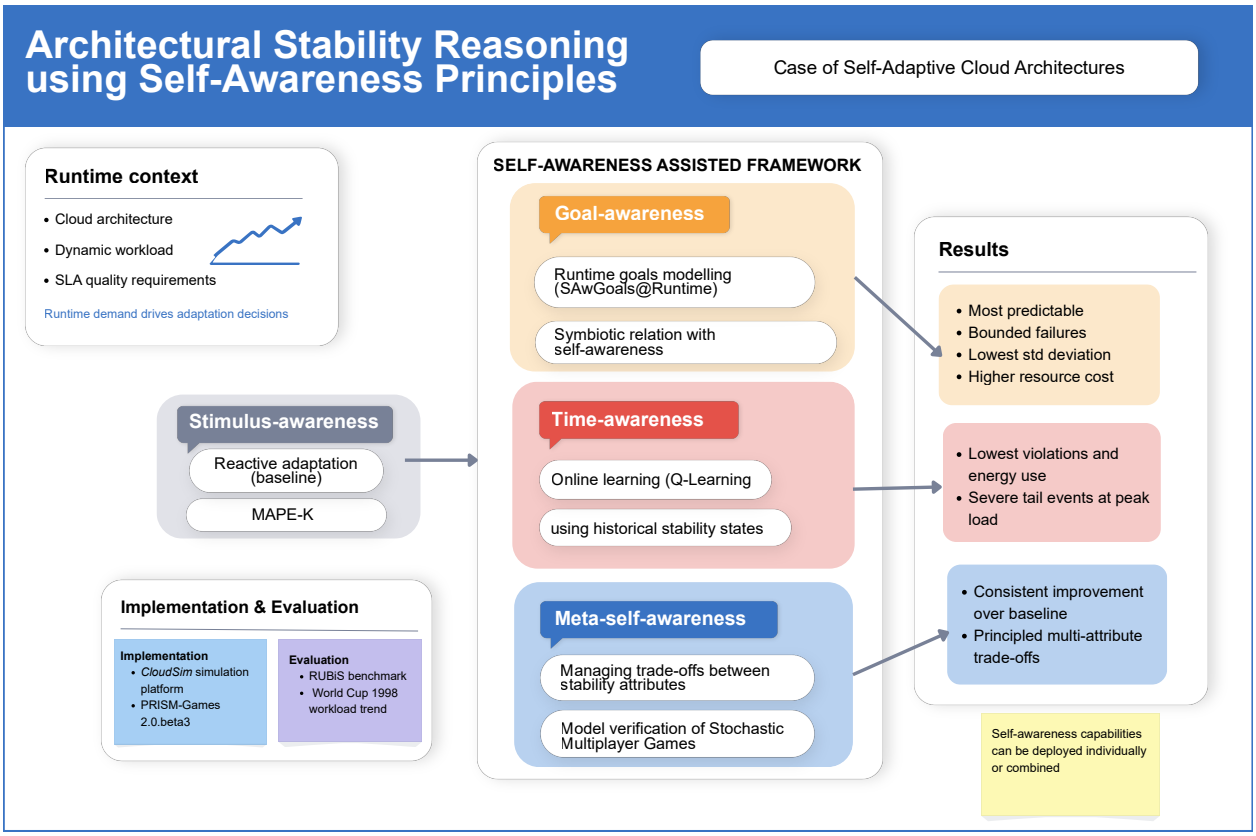
Highlights

Architectural Stability Reasoning using Self-Awareness Principles: Case of Self-Adaptive Cloud Architectures

- Goal models, online learning, and stochastic games underpin stability reasoning
- Goal-awareness achieves the most predictable stability with bounded SLA violations
- Time-awareness reduces violations, energy use, and adaptation overhead
- Meta-self-awareness moderates instability via stochastic game-theoretic trade-offs
- Self-awareness capabilities improve over reactive adaptation on stability metrics

Journal Pre-proof

Journal Pre-proof



Journal Pre-proof

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Journal Pre-proof