SELF-AWARE SOFTWARE ARCHITECTURE STYLE AND PATTERNS FOR CLOUD-BASED APPLICATIONS

by

FUNMILADE OLUGBENGA FANIYI

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

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
June 2015

Abstract

Modern cloud-reliant software systems are faced with the problem of cloud service providers violating their Service Level Agreement (SLA) claims. Given the large pool of cloud providers and their instability, cloud applications are expected to cope with these dynamics autonomously. This thesis investigates an approach for designing self-adaptive cloud architectures using a systematic methodology that guides the architect while designing cloud applications. The approach termed *Self-aware Architecture Pattern* promotes fine-grained representation of architectural concerns to aid design-time analysis of risks and trade-offs. To support the coordination and control of architectural components in decentralised self-aware cloud applications, we propose a *Reputation-aware posted offer market coordination mechanism*. The mechanism builds on the classic posted offer market mechanism and extends it to track behaviour of unreliable cloud services.

The self-aware cloud architecture and its reputation-aware coordination mechanism are quantitatively evaluated within the context of an Online Shopping application using synthetic and realistic workload datasets under various configurations (failure, scale, resilience levels etc.). Additionally, we qualitatively evaluated our self-aware approach against two classic self-adaptive architecture styles using independent experts' judgement, to unveil its strengths and weaknesses relative to these styles.

Acknowledgements

On the journey that gave birth to this thesis, I have had the privilege of enjoying the support of many wonderful people. First, I'll like to specially thank my supervisor, Dr. Rami Bahsoon, for his immeasurable support and patience throughout the course of my study. Your dedication to your students and research is an example worth emulating.

I'm sincerely grateful to members of my thesis group: Prof. Xin Yao, Dr. Nick Hawes, and Dr. Marco Cova. Your useful insights in the bi-annual meetings have shaped my understanding of how to do research and made my experience an enjoyable one.

I'm grateful to the School of Computer Science, University of Birmingham, for sponsoring my studies and many academic conferences over the course of the PhD programme.

I thank members of the EU *Engineering Proprioception in Computing Systems (EPiCS)* project for many fruitful collaborations during the course of the project. Noteworthy are Peter R. Lewis, Tao Chen, Leandro Minku, and EPiCers in the Birmingham team.

Thanks to members of SERG for making life in Computer Science an interesting one.

Lots of thanks to Bendra, Ronke, and Ogechi for proofreading drafts of this thesis.

I sincerely thank Dr Nelly Bencomo and Dr Shan He for the time and effort in examining this thesis and providing very useful comments. The stimulating discussions during the viva and useful feedback has helped to significantly improve the thesis.

Special thanks to my brother from another mother, Ayobami Adediji. Words cannot describe how grateful I am for your support. I also thank members of RCCG Winners Place Aldershot for their prayers and encouragement, especially the Fagbayimu's.

I'm especially grateful to my family. Amos, Bola, Fola, Fade, Adura, and Bayo, words cannot describe how much I appreciate your support and patience through many emotional roller coasters typical of graduate studies. You are the most wonderful family I could wish for. Special thanks to my wife, Olumayowa, for being a strong pillar of support and a friend I can always count on. Thank you May.

Finally, I thank my Lord Jesus Christ for being my Helper.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

> "Man tries to make for himself in the fashion that suits him best a simplified and intelligible picture of the world; he then tries to some extent to substitute this cosmos of his for the world of experience, and thus to overcome it. This is what the painter, the poet, the speculative philosopher, and the natural scientist do, each in his own fashion."
>
> *Albert Einstein*

1.1 Motivation

Today's software systems, influenced by new trends such as social networking, are orders of magnitude larger than traditional systems that were in use about a decade ago. The cloud is the preferred platform for provisioning these modern large-scale systems due to its scalability and cost savings. Cloud computing provides on-demand access to an infinite pool of computational resources at a reduced costs than those incurred when provisioned in-house [28] [8].

As with conventional computing systems, cloud infrastructures are error-prone. Unpredictable software crashes and service outages are unavoidable realities of cloud systems. To assure cloud users (also referred to as application owners in this thesis), cloud providers agree to service terms that encode expected quality of service levels along with penalties

to be enforced in the event of failure to comply. These service terms are contracted as Service Level Agreements (SLAs).

SLA management (SLM)[112] is the field concerned with end-to-end management of a service throughout its life. At the heart of SLM research is the problem of how to allocate limited resources to users. It has been shown that resource allocation to service-based applications, such as cloud applications, to meet SLAs is an NP (non-polynomial) hard problem [7] [30]. Since demand for cloud resources may rapidly exceed planned forecast [164], cloud providers are unlikely to always achieve 100% SLA compliance.

As cloud applications become more mission-critical, users are switching from using a single cloud to a federated provisioning model. In the federated model, cloud services can be purchased from multiple cloud providers and combined to fulfil an application's requirements. By spreading the risks of SLA violation among multiple clouds, federated cloud applications can utilise a subset of these clouds that are reliable per time. It is projected that novel software architectures will be required for the vision of SLA compliant federated cloud applications to be realised [141].

Self-adaptation [42] offers a cost-effective approach for modelling dynamics of federated clouds and is promising at managing complex trade-offs among conflicting requirements. This is because self-adaptation endows a system with the ability to manage changes in itself and its operating environment with minimal human intervention [100]. However, given that entities in a cloud federation are decentralised, the underlying self-adaptive architecture of a federated cloud application requires an efficient mechanism for coordinating clouds, owned by different entities.

1.2 Problem Statement

Cloud computing is an active research area, yet a systematic approach for architecting federated cloud applications is an open problem. Typically, application owners continually change their service level expectations (e.g. availability and performance) depending on

emergent business requirements. Cloud providers, on the other hand, face a number of decision problems, for example: which SLAs to selectively violate in the event that all users' SLAs cannot be fulfilled? The focus of this thesis is on the application owner's perspective of the cloud SLM problem.

To cope with changing cloud providers' situation (e.g. transient reliability), the application's software architect needs to design an application that is able to:

- coordinate interaction with a large pool of cloud providers,
- select candidate cloud providers per time without compromising service quality, and
- revise cloud provider selection if quality expectations are not satisfied.

These architectural concerns call for a self-adaptive approach that takes into account dynamics of cloud providers, changing workload demands, and variations in end users' service level objectives (SLOs).

1.2.1 Areas Requiring Improvement

The following areas of existing cloud SLM solutions require further investigation.

- *Lack of a systematic approach to design and instantiate self-adaptive cloud architecture.* Federated cloud applications require decentralised software architectures to account for distributed ownership of architectural components. However, the literature on self-adaptive cloud software architectures mostly addresses the problem of self-adaptation from a centralised perspective (e.g. [33] [127]). While there are limited attempts to proffer decentralised self-adaptive architectural solution, e.g. [92] [15], such attempts do not follow a systematic approach to design and instantiate candidate architectures.
- *Coarse-grained architectural knowledge models and limited support for trade-off analysis.* In cloud research, it is usually assumed that users' SLOs do not conflict. Consequently, trade-off among SLOs are not explicitly modelled and analysed to inform

decisions about cloud provider selection. Crucially, self-adaptive cloud architecture (e.g. [121] [18]) do not explicitly model architectural knowledge concerns (e.g. goal, time, and interaction) at a fine-grain. State of the art knowledge models offer limited support for detailed trade-off analysis and do not help unveil design decisions that may pose risks to the cloud application.

- *Poor scalability of existing cloud self-adaptation mechanisms.* Federated cloud applications are deployed to large user populations, thus scalability is a key requirement. Fundamentally the adaptation mechanism implemented in the application's underlying architecture must be scalable to support various workload scenarios. State of the art self-adaptive cloud applications utilise heuristics (e.g [32]), policies (e.g [89]), and optimisation (e.g [175]) techniques, amongst others, as adaptation mechanisms. Whilst these techniques are effective for small scale workload scenarios, they do not scale well to large user populations.

1.3 Thesis Overall Aim and Objectives

This thesis proposes an architecture-centric self-adaptive approach for managing federated cloud applications that aim to comply with customer SLAs. Inspired by the concept of *Computational Self-awareness* [115] [13] we aim to contribute architectural patterns for designing cloud applications. A self-aware system is one that "...possesses information about its internal state (private self-awareness), and sufficient knowledge of its environment to determine how it is perceived by other parts of the system (public self-awareness)." [115] Computational self-awareness therefore enables sophisticated self-adaptive behaviour by endowing a system with an explicit knowledge model about itself and its operating environment. By decomposing knowledge into fine-grained levels of abstraction, as advocated by the self-aware approach, analysis of architectural design decisions is simplified [68].

Concretely, we aim to *propose architectural patterns, for designing self-aware federated cloud applications, that promotes a systematic analysis of knowledge concerns at*

fine-grained levels. The architectural patterns shall provide guidance to software architects for instantiation of candidate architectures and support the process of trade-off and risk analysis. We additionally propose an efficient and scalable market mechanism for coordinating decentralised components of self-aware federated cloud applications.

The following objectives are formulated to help us realise this aim:

- To innovate self-aware architectural patterns that adheres to well-founded architectural principles.
- To utilise the self-aware architectural patterns for systematically reasoning about the design and analysis of cloud architectures.
- To design a scalable mechanism for coordinating interaction of decentralised components in federated cloud applications designed based on the proposed patterns.

This thesis argues that self-awareness and its underlying primitives can be exploited to design SLA-compliant federated cloud applications. The levels of self-awareness presented in this thesis vary from simple stimulus-awareness to increasingly advanced meta-self-aware level [115]. This thesis studies the problem of self-adaptation from an *architecture style* perspective [134]. The primary motivation for this approach is to abstract the architecting process to a high level of abstraction that affords studying the underpinning principles that makes one software architecture different from another.

To ease the coordination of cloud applications, we turn to economics-inspired market mechanisms as a solution concept. Suppose we view cloud providers as *sellers* and application owners as *buyers* in a cloud market. The market analogy provides a good abstraction for modelling interaction of these cloud players. Markets are decentralised, scalable, and offer robust mechanisms for managing distributed systems [47]. Therefore, we adopt a suitable market mechanism namely the posted offer market mechanism [101].

The canonical posted offer market mechanism informs control decisions solely based on asking price of sellers, hence the ability of sellers to fulfil service levels is not considered. We propose a novel refinement to the posted offer mechanism to address this limitation

by capturing historic interaction with sellers using concepts from reputation management. Combining price and reputation ratings as decision variables therefore reduces the likelihood of selecting unreliable sellers (cloud providers).

1.4 Research Philosophy and Method

Scientific research calls for a disciplined approach to arrive at new findings or combine known results in novel ways. Research approaches vary depending on the investigated questions and research area involved. Shaw [148] argued that Software Engineering (SE) as a young discipline lacks consensus on accepted strategies for answering research questions. Shaw advocates that SE should mature towards a set of well-accepted research strategies to match practices in more established sciences.

Broadly, SE researchers often make use of qualitative approaches (e.g. argumentation and case studies) or quantitative approaches (e.g. proofs and statistically deduced evidences) to validate results. The quantitative approach aims for repeatable experiments, to help identify errors and promote independent verification of results [162]. Whilst sufficient for some areas (e.g. formal methods), quantitative approaches do not cater for the entirety of research areas in SE.

Software architecture research, in particular, inherently produce results in the form of abstractions and processes that help to design high quality software systems. The nature of software architecture research promotes at its core results that are process-oriented rather than concrete artefacts in the sense of software products. Two common techniques for validating software architecture results [148] are (i) to collect and analyse evidences from experts (qualitative) and (ii) to utilise case studies that demonstrate the proposed concepts in controlled experiments (quantitative).

The approach taken in this thesis combines elements of qualitative and quantitative approaches. The investigated research questions are initially validated via systematic reviews to identify gaps in the literature. The thesis thereafter investigates the hypothesis that

by designing self-adaptive systems using our proposed self-aware architectural patterns, software architects are able to more explicitly capture and analyse risky architectural design decisions and trade-offs among quality concerns. By systematically adhering to self-aware architectural patterns, software architects are forced to: (i) thoroughly reason about architectural design decisions, and (ii) account for interaction between architectural design decisions and how they trade-off against one another. Consideration of these two dimensions are crucial to improving coverage of the system's design space [57].

To test this hypothesis, a representative online shopping application case study underpinning key attributes of federated clouds was motivated. Based on this case study, firstly we validate our self-aware architecture patterns using findings from expert assessments of candidate architectures. Secondly we empirically study and validate our reputation-aware posted offer market mechanism within the context of the case study.

1.5 Running Example: Online Shopping Application in Federated Cloud

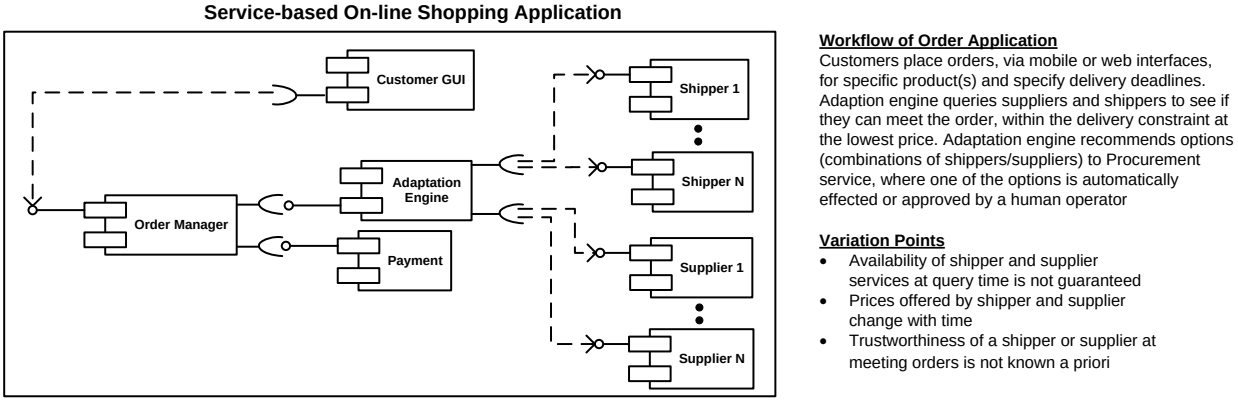


Figure 1.1: Online Shopping Application faces a selection problem when considering shipping and supplier services per order.

We consider an online shopping application as a representative example of service-based applications (SBAs) deployed in a cloud federation (see figure 1.1). Typically, every SBA is composed of abstract services, which are instantiated by functionally equivalent

concrete services at run-time (see examples in figure 1.2). The architecture of each SBA differs in topology and its constituent abstract services. The goal of each SBA is to ensure it complies with customer SLAs. Each SBA has a local perspective of its SLA compliance goal and is unaware of goals of other SBAs.

SBAs share a large pool of cloud services that offer various service levels at prices which changes with time. Externally, SBAs interact via the shared cloud services and possibly interfere with one another when competing for services. Consequently, there is a tension between each application's local goal satisfaction and the (global) objective of fair resource distribution among SBAs.

In order to manage application dynamics, each SBA is endowed with an adaptation engine (component). The objective of the component is to manage the pool of cloud services on behalf of the SBA and ensure its SLA goals are met. Thus the component makes service selection decisions, reacts to time-varying workload demands, and releases selected services in an elastic manner as demand dwindles. Moreover, the components owned by different SBAs autonomously coordinate and resolve conflicts among themselves to ensure their private goals are achieved while respecting the global objective.

To realise the online shopping federated cloud application it must address two issues.

1. Its software architecture must make it easy to represent and analyse architectural concerns i.e. goals, timing variations, and interactions with external cloud services. Therefore an architectural approach that caters for these concerns and provides support for analysis is needed.

2. A coordination mechanism for efficiently managing interaction with the large pool of cloud services without compromising SLA goals is essential.

As earlier motivated, this thesis adopts computational self-awareness as an approach for capturing architectural concerns and market-based control for coordination of architectural components. Within the context of this case study, we investigate two main research questions:

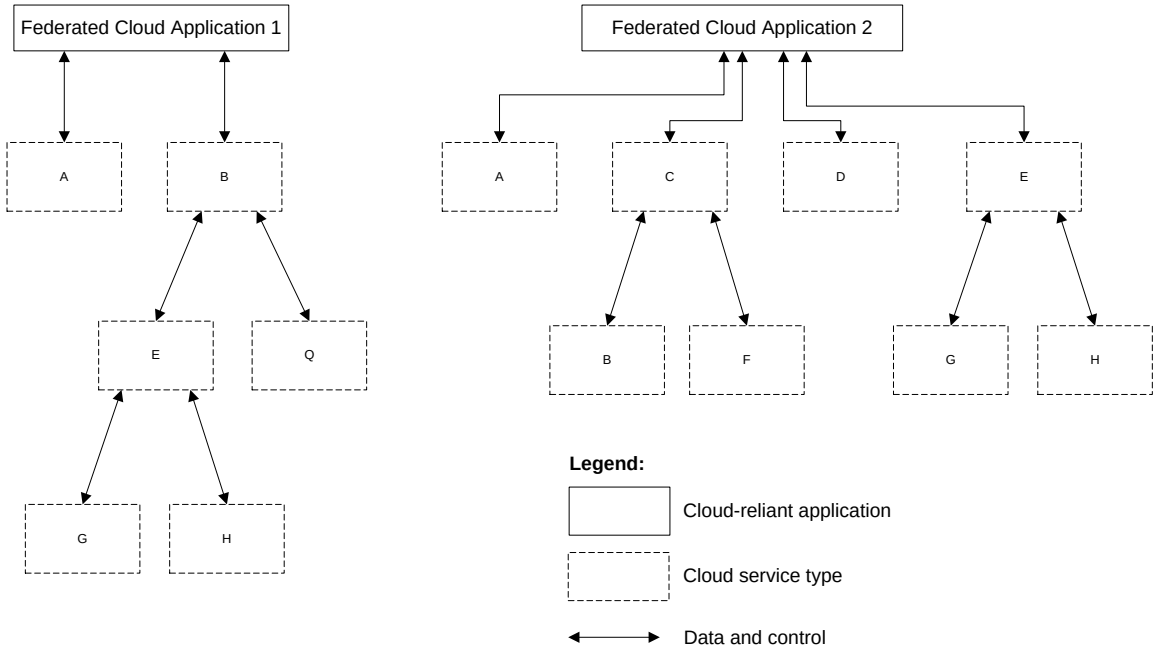


Figure 1.2: High-level overview of federated cloud applications showing reliance on cloud services

Q1: What are the architectural patterns that can be used by software architects to design SLA compliant self-aware federated cloud applications?

Q2: How can market-based control be utilised to coordinate decentralised components in the designed self-aware architectures whilst respecting SLA compliance goals?

To answer Q1, we propose five self-aware architectural patterns that provide primitives for representing architectural concerns at a fine-grained level and therefore simplifies risk and trade-off analyses. To answer Q2, we incorporate reputation measurement capabilities into the classic posted-offer market mechanism. We demonstrate that our refined mechanism is capable of coordinating interaction with multiple clouds and provide satisfactory SLA compliance.

1.6 Main Contributions

The thesis demonstrates that by taking an architecture-centric self-adaptive approach grounded on the principles of self-awareness and market coordination, federated cloud applications can improve their SLA compliance as a result of an holistic treatment of risks, trade-offs, scale, and cloud dynamics. The implication of this result is that owners of mission critical applications, e.g. enterprise software systems, will be confident to entrust their applications to the cloud, with an expectation of satisfactory SLA compliance.

The main contributions of this thesis are:

- An approach for designing self-adaptive federated cloud architecture, namely self-aware architectural patterns, based on principles of computational self-awareness. The proposed architectural patterns adheres to well-founded principles such as separation of concerns by allowing architects to reason about the representation of architectural concerns and selection of behavioural strategies independently, focusing on the feedback loops between these processes.

- A market-inspired mechanism for coordinating component interaction in self-ware federated cloud applications. In particular, we extend the classic posted offer market mechanism [101], incorporating reputation measurement capabilities to facilitate selection of cloud services based on consideration of price and historic performance.

- A systematic literature survey of SLA-based cloud research. This survey revealed advances and gaps in state-of-art SLA-based cloud research and motivates the need for an architecture-centric self-adaptive approach.

- A classification framework for structuring the literature in self-adaptive architecture style and qualitatively comparing properties of these styles on dimensions such as level of separation of concern and in-built support for learning. The comparison is aimed at understanding the underlying principles underpinning self-adaptive architecture styles.

The list of publications resulting from this research can be found in appendix A.

1.6.1 Summary of Contributions By Chapter

Table 1.1 lists the contribution of each chapter (detailed in section 1.7). The presented work is the author's original contribution, with the exception of *Self-aware Architecture Style*, which was jointly incepted by the author and members of EPiCS project.

Ch.	**Contribution**	**Credit**
2	Systematic Review of Service Management in Cloud	Author
3	Architecture-based Self-adaptation Styles	Author
4	Self-aware Architecture Style	Author and EPiCS team
4	Self-aware Architectural Patterns	Author
5	Market-inspired Mechanism for Decentralised Coordination	Author
6	Trade-off and Risk Analysis of Self-aware Cloud Software Architecture	Author
7	Conclusion and Future Work	Author

Table 1.1: Summary of Contributions By Chapters

1.7 Roadmap of the Thesis

This section presents the structure of the rest of the thesis. The roadmap to the thesis is shown in figure 1.3.

Chapter 2, Systematic Review of Service Management in Cloud, studies the state of the art in SLA-based cloud research using a systematic review methodology. Findings from the survey are used to motivate the need for a novel self-adaptive architectural design approach.

Chapter 3, Architecture-based Self-adaptation Styles, presents our classification framework for structuring and characterising self-adaptive architecture styles. A comparative study of eight representative self-adaptive architecture styles reveals gaps that motivate the self-aware architectural patterns work presented in chapter 4.

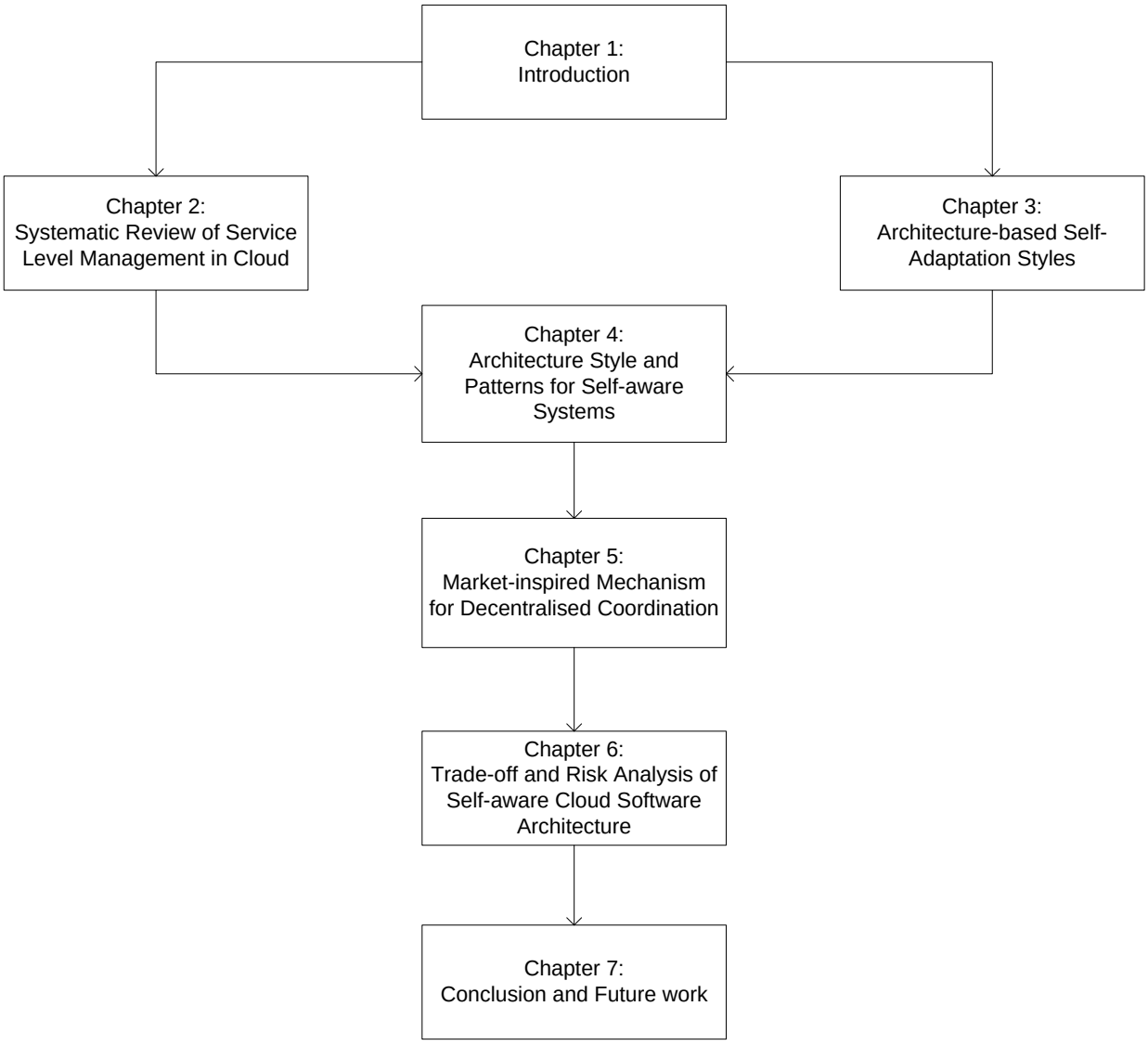


Figure 1.3: Roadmap of the Thesis

In *Chapter 4, Architecture Styles and Patterns for Self-aware Systems*, we present our five self-aware architectural patterns. The primitives of the self-awareness concept from psychology are introduced. Self-aware architectural patterns are presented to promote a systematic and disciplined way of architecting self-aware cloud applications.

Chapter 5, Market-inspired Mechanism for Decentralised Coordination, motivates the use of economics-inspired approaches for decentralised coordination of federated cloud application components and presents our refinement to the classic posted-offer market mechanism namely *reputation-aware posted offer mechanism*. Results from empirical studies

of the market-based self-aware cloud architecture under synthetic and realistic workload are presented to demonstrate its ability to achieve satisfactory SLA compliance.

Chapter 6, Trade-off and Risk Analysis of Self-aware Cloud Software Architecture, presents a qualitative evaluation of the self-aware architecture style within the context of our online shopping cloud application. The evaluation is carried out in comparison to two classic self-adaptive architecture styles (3-Layered [105] and DDDAS [53]) using the Architecture Trade-off Analysis Method (ATAM) [97]. The results demonstrate the potential of the self-aware architectural pattern to support trade-off analysis for federated cloud applications.

Finally, *Chapter 7, Conclusion and Future Work*, concludes the thesis by summarising the main contributions, reflecting on the research, and discussing avenues for future work.

CHAPTER 2

SYSTEMATIC REVIEW OF SERVICE LEVEL MANAGEMENT IN CLOUD

> "Society is indeed a contract. It is a partnership in all science; a partnership in all art; a partnership in every virtue, and in all perfection. As the ends of such a partnership cannot be obtained in many generations, it becomes a partnership not only between those who are living, but between those who are living, those who are dead, and those who are to be born."
>
> *Edmund Burke*

2.1 Overview of the Chapter

This chapter surveys the landscape of SLA-based cloud architecture to understand state of the art and identify open problems. We adhere to the Systematic Literature Review (SLR) guideline proposed by Kitchenham [102] [103]. A SLR documents the end-to-end process of a review. Kitchenham's guideline aims for a repeatable review process, where the search protocol can be reproduced by an independent assessor and the findings interpreted within the context of the research questions that triggered the review process.

The key findings of the systematic review indicate that MAPE-K[1] and its variants are the prominent self-adaptive architecture style in use in SLA-based cloud research. The

[1]MAPE-K is an acronym for Monitor, Analyse, Plan, Execute, and Knowledge phases of the famous IBM self-adaptive architecture style [100]. Chapter 3 studies MAPE-K and other representative styles.

result also indicates that, in general, knowledge representation at the architecture level and decentralised self-adaptive software architecture have received little attention.

One underlying theme of previous work [22][131][18] is that given the moderate dynamism and scale of systems preceding clouds, a centralised architecture often suffice as a solution. On the contrary, centralised architectures are not feasible for managing SLAs of federated cloud applications due to their large scale and dynamic topology [69].

We posit that an approach for architecting federated cloud application should provide primitives for modelling knowledge concerns at a fine-grain, to ease risk and trade-off analysis. Broadly, this thesis contributes novel self-aware architectural patterns that provide primitives to support decentralised properties of federated cloud applications.

The rest of the chapter is structured as follows. Section 2.2 introduces cloud computing, service level agreement, and autonomic computing within the context of cloud. The research questions that steered the review process are presented in section 2.3.1 (see appendix B for the systematic review protocol). Findings from the review are discussed in section 2.3.2. The chapter concludes in section 2.4.

2.2 Preambles

There is no consensus on the definition of Cloud computing [8], however, three widely-adopted definitions are those proposed by NIST [122], Buyya et al. [28], and Vaquero et al. [164].

According to NIST [122]

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released *with minimal management effort* or service provider interaction."

Buyya et al. [26] proposed that

> "A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resource(s) *based on service-level agreements* established through negotiation between the service provider and consumers."

Vaquero et al. [164] analysed more than 20 definitions of Cloud computing, and proposed an integrated definition

> "Clouds are a large pool of easily usable and accessible virtualised resources (such as hardware, development platforms and/or services). These *resources can be dynamically reconfigured* to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider *by means of customized SLAs*."

A common theme underlying cloud computing is that the cloud is fundamentally *dynamic* [75] [164] [25]. This phenomenon can be viewed from three distinct perspectives namely: (i) dynamic workings of the cloud system itself, (ii) dynamics due to changing user behaviour and requirements, and (iii) dynamics of the cloud deployment environment (e.g. network topology and runtime composition of services).

From the above perspectives, two important requirements come to light:

1. Cloud computing should be relatively autonomic to support dynamic provisioning and reduce management effort.

2. Cloud providers and application owners should implement dynamic management schemes to ensure SLAs are honoured.

2.2.1 Service Level Agreement in Computing

Service level agreement (SLA) was traditionally a business concept, as it specifies contractual financial agreements between parties who engage in business activities. Business SLAs were typically encoded in manual paper documents. Consequently, it was difficult to monitor them. Detecting whether the terms of an SLA were honoured and enforcing

penalties heavily involved humans, to interpret the SLA and mediate between signatory parties. In the late 1990s the concept of SLA gained the attention of academics and practitioners in the the computing world [17]. Telecommunications and Enterprise Network [113] communities were some of the early adopters.

Up to this point, SLAs were mostly defined in an ad-hoc manner or at best standardised only for use within specific family of organisations or application domain. Another drawback was the rigid specification of the terms of SLAs, as it was not possible to adapt the values of SLA terms once they were deployed. The emergence of grid computing and service-oriented architecture (SOA) triggered a number of important advancements in the specification of automated SLAs. This is because the openness and autonomy of grids and web services required specification formats that were not restricted to any organisation or application domain's syntax or semantics.

Two notable standardisation efforts addressed the SLA specification problem: IBM pioneered work on Web Service Level Agreement (WSLA) [52] and the Open Grid Forum (OGF) proposed Web Service-Agreement (WS-Agreement) [4]. Both frameworks promoted a notion of service-agnostic definition of service terms, measurement of service metrics, aggregation of metrics within the context of SLA parameters, and monitoring of service level objectives (SLOs). In both cases, XML schema was used as the underlying language for expressing WSLA and WS-Agreement. Therefore, the standards were sufficiently open for adoption in many application domains.

2.2.2 Life cycle of a Service Level Agreement

Over the years, researchers have contributed to the vision of automated SLA management (SLM). These contributions can be categorised along the lines of a typical SLA life cycle as shown in figure 2.1. SLM is a broad topic involving negotiation, deployment, monitoring, reporting and termination phases. A brief overview of each phase of the life cycle follows.

- *SLA Negotiation:* The requirement in this phase is for involved parties to define

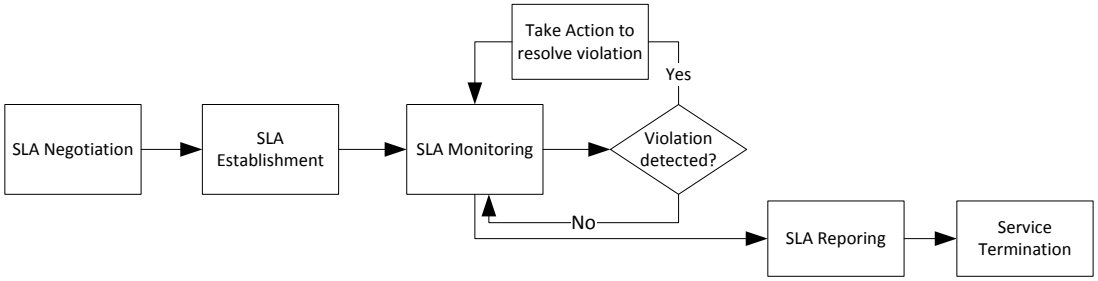


Figure 2.1: Classic SLA Management Life Cycle

terms of service and agree to levels at which service will be provided including monetary aspects. The negotiation process may provide mechanisms to support dynamic negotiation of service levels that reflects changing QoS demand of cloud users as their business operations evolves [29]. An example of such changes could be a request for more VM instances due to spikes in workload resulting from flash crowd effects. The agreed service levels are encoded using either a standard application-agnostic SLA template (e.g. WSLA [52] and WS-Agreement [4]) or ad-hoc templates that involved parties are able to interpret.

Another interesting point is that parties to the negotiation (i.e. cloud users and providers) often adopt incompatible SLA templates [19]. Therefore, negotiation among these parties often requires translation to a base SLA template before any negotiation can take place. The challenge here is to ensure these translations techniques are extensible (to accommodate new templates) and capable of producing accurate representation of the original SLAs. It is hopeful that standardisation efforts towards a unified SLA template for cloud computing would completely resolve this problem in the future.

- *Service Deployment:* Typically, service/job requests from clients are assigned to cloud resource nodes in this phase. The goal is to allocate resources having the required capacity to jobs based on their specification and valuation. SLAs often come in different classes such as Gold, Silver, and Bronze, where each class represents a different valuation of the users in that class. Cloud providers may optionally

discriminate between jobs depending on the SLA class of users who own the jobs.

- *SLA Monitoring:* Once services are deployed, it is important to periodically monitor resource nodes and the status of jobs under their execution. The monitoring activity could span many dimensions such as monitoring the functional and non-functional requirements of the job, monitoring the status of resource nodes and monitoring network availability. Monitored data can quickly grow out of proportion, hence, the time taken to analyse the data may become a bottleneck. Consequently, it is important to monitor only *relevant data* and ensure that lightweight analysis techniques are employed.

- *Violation Management:* Violation alerts which represent the likelihood of a job failing or not meeting its defined service levels are sometimes reported as part of monitored data. The goal is to take appropriate risk mitigation decisions in response to these violation alerts. The decision should be timely and suitable for the context of the violation alert. A worst-case resort may be to re-deploy the job if the risk of violating the SLA cannot be averted.

- *SLA Reporting and Termination:* the emphasis at the reporting phase is to provide SLA reports of high integrity containing detailed audit of activities that took place during service provisioning [38]. The termination phase provides a mechanism for parties to the agreement to terminate the SLA after completion of the service or in response to violations caused by any of the parties as specified in the SLA.

Next, we zoom into the rationale for an autonomic approach to cloud SLM.

2.2.3 Why Autonomic Solution to Cloud Resource Management?

Autonomic computing systems are a class of software systems endowed with abilities to manage themselves, similar to the autonomic nervous system's role in managing human body, by adapting to changes in their operating environment, user requirements, or inter-

nal changes in the system itself at run-time. Kephart and Chess [100], formally defined autonomic systems as "computing systems that manage themselves in accordance with high-level objectives from humans."

Researchers often refer to autonomic systems as "Self-adaptive", "Self-managed", "Self-organising", or "Self-*" computing system. It is not uncommon to find researchers using these terms interchangeably. As an example consider the definition below:

> "*Self-managed systems* are those that are capable of adapting as required through self-configuration, self-healing, self-monitoring, self-tuning, and so on, which are also referred to as *self-** or *autonomic systems*." [106]

We adopt the generic terms self-adaptive and autonomic computing, which will be used interchangeably in the rest of the thesis. From the conceptual underpinning of autonomic computing, we identify four key motivations for adopting a self-adaptive solution for the problem of cloud SLA management.

1. Large size: The large scale of cloud federations has exacerbated the administrative overhead of SLA management. For these large-scale systems, the time lag and high cost overhead of human-based solution, makes autonomic control solution the more appealing option. Patikirikorala et al. [132] surveyed self-adaptive systems that were designed using control approaches. Of particular interest, is the finding, by the authors, that the recent increase in research effort in the use of control theory for managing software systems is due to large systems such as cloud.

2. Heterogeneity: The openness of cloud systems coupled with its realisation using service-oriented architecture has stretched the limits of conventional and naively autonomic systems. In today's open cloud systems, cloud service providers are unable to fully anticipate the various contexts in which their services will be composed with services provided by other cloud providers. Therefore, there is the problem of self-configuring these cloud services in the most seamless way and self-optimising them at run-time to maintain acceptable quality of service. In addition, services

that are found to be faulty must be repaired using self-healing mechanisms and vulnerable services, susceptible to security attacks, require self-protection to prevent exploitation by malicious users.

3. Dynamism: The presence of many heterogeneous cloud services and components means architects of cloud-based applications have to cope with a large configuration space. This is exacerbated by the varying demands from cloud users that cause workload fluctuations, hence no single cloud service is the best for all usage scenarios.

4. Uncertainty: In federated clouds both internal triggers (e.g. software bugs) and external triggers (e.g. workload spikes) of adaptation can occur haphazardly, disrupting system stability over time. Cloud systems need to account for these changes using autonomic mechanisms to remain useful to users.

Clearly a conventional static resource control approach, relying solely on human operators, is not feasible to meet the requirements of today's cloud. We argue that autonomic computing principles holds the promise to solving the challenges motivated above.

2.3 Systematic Review Methodology and Results

2.3.1 Research Methodology

The thesis has followed a systematic review methodology to investigate the state of the art in SLA-based cloud research. This thesis studies papers that have implications for the design of self-adaptive cloud architectures. Specifically, the following pertinent research questions steered the review:

> What are the dominant architectural styles for designing federated cloud applications? To what extent do these styles provide support for trade-off analysis?

The details of the review protocol can be found in appendix B. The next section presents findings from the survey and outlines gaps in state of the art.

2.3.2 Results of Systematic Review

At the software architecture level, first we observe that 40% of papers claimed to offer solutions grounded in autonomic computing, while the other 60% do not. Of the papers that claimed to be autonomic, many of the solutions were of algorithmic nature rather than architectural in their approach. Table 2.1 shows the architecture styles in use in those papers that make explicit reference to autonomic architectures. Note, we use the term 'autonomic architecture' to refer to self-* architectures in general as defined in section 2.2.3.

XaaS	**Arch. Style**	**Self-* Arch. Artefact**	**Goal of adaptation**	**Representative Examples**
IaaS	MAPE-K	VM controller	To devise an improve knowledge management technique in the MAPE-K control loop	[18] [121]
	Collect-Analyse-Anticipate-Decide control loop	Cloud resource controllers	To adaptively control cloud resources and manage energy	[166]
	MAPE	Cloud resource manager	To allocate resources while optionally minimising operating cost	[127][65] [63][88]

	Decentralised agent-based architecture	Cloud resource controllers	To devise scalable resource managers	[176]
	Hierarchical control loop	Cloud resource controllers	To achieve a cost-aware and workload-sensitive resource allocation	[175]
SaaS	Decentralised agent-based sense-plan-act architecture	Application resource manager	To make composite cloud web services available under heavy load and failures	[15] [92]
	Centralised MAPE architecture	Load balancing service	To balance application workload across VMs	[31]
Cloud Federation	MAPE-K	Centralised federated cloud resource manager	To optimally distribute workload at a minimal cost	[33]

	Two-level Hierarchical architecture	Cloud resource managers	To realise optimised allocation performance across independent clouds or grids	[87]
PaaS	Centralised monitor-manager-allocator	Resource provisioning service	To dynamically allocate cloud resource to jobs	[24]
	Hierarchical MAPE loops	Middleware component	To improve resource utilization and performance of cloud applications	[179]
DaaS	Monitor-Analyser-Predictor-Allocator	Resource allocation framework	To maximise resource utilisation	[180]

	Hierarchical adaptation loop	Cloud resource allocator	To manage shared database resources in a cost-aware manner	[174]

Table 2.1: Software Architecture Styles By XaaS

As it can be observed from table 2.1, MAPE and its variants are the most dominant architecture styles as they are applied across several cloud layers of abstraction namely IaaS, SaaS, PaaS, DaaS, and cloud federation. Hierarchical architectural styles are used at all the aforementioned cloud layers, except SaaS, for the purposes of reducing complexity of adaptation across different levels of concerns. Whilst the approach simplifies the architecting process, both centralised and hierarchical architectures are prone to problems such as brittleness, limited scalability, and single-point of failure. Decentralised agent-based architectures are in limited use at the IaaS and SaaS cloud layers. In instances where they are used (e.g. [15] [176]), they address the problem of scalability and single-point of failure present in centralised and hierarchical architectures.

Trade-off analysis

20% of the reviewed papers provided trade-off analysis between conflicting SLA parameters in their service level objectives. Table 2.2 shows the studied trade-off spaces.

Trade-off Parameters
Application performance Vs Power or Energy consumption/efficiency
Application performance Vs Resource operation cost
Cost Vs SLA fulfilment
Cost Vs Energy

Cost Vs Latency
Cost Vs Performance
Energy consumption Vs SLA violation or fulfilment
Energy cost Vs Client QoS
SLA violations Vs Resource utilization Vs Energy consumption
Throughput Vs Read size
System utilization Vs SLA optimization goals

Table 2.2: Trade-off Analyses Space

It can be observed that at most three SLA parameters were analysed in the trade-off space whilst analysing self-adaptive cloud architecture for SLA-based resource allocation. Given the limited number of SLA parameters studied in state of the art, it may be argued that the trade-off space is representative of these SLA parameters. It is expected that as more SLA parameters are studied, the number of analysed non-functional quality attributes in the trade-off space would increase.

2.3.3 Threat to Validity

Some threats to the results of the SLR reported in this chapter are discussed below.

- Data sources: We have primarily collected studies from academic indexing services. This has limited our understanding of the research topic to academic contributions, except in cases where publications are co-authored with industrial practitioners. A broader search, including data sources such as websites of companies that provide and use cloud services may provide an interesting perspective to the review.

- Recency of findings: The study has considered papers up to the last quarter of 2013. As a mitigation step to confirm if the results of the review are still valid, we conducted a supplementary review following the same protocol for only papers

published in 2014. This revealed new studies that adhered to an architectural approach to self-adaptation (e.g. [39] [91] [85]), however, the MAPE-K architecture style was used in majority of the cases. Also, majority of the papers addressed resource management problems at IaaS cloud layer, with the exception of [85] which focused on reducing network delays at NaaS layer. Therefore, we assert that the effort to pursue the self-aware architectural patterns which promotes fine-grained architectural knowledge representation remains a viable and timely endeavour.

- Analysis of collected studies: We have analysed the primary studies with respect to our research questions, which is primarily about self-adaptive software architecture styles. It may be worthwhile to investigate other phases of SLA life cycle, such as negotiation, to understand the interrelation between problems in these phases and the SLA-based resource management problem.

2.3.4 Gap Analysis

From the foregoing, we can deduce the following gap in the literature:

1. Software Architecture Knowledge representation in the studied self-adaptive software architectures are given little considerations (e.g. goal of adaptation, temporal aspects of architectural elements, links between interacting components, etc.). Exceptions are [18] and [121] where a case-based reasoning knowledge representation technique is used to more SLA objectives. In both [18] and [121], knowledge is modelled as a coarse-grain entity, hence making it hard to reason about trade-offs among knowledge concerns as they relate to timeliness of adaptation, goals of adaptation, and interaction amongst cloud components. We argue that modelling knowledge elements of the adaptation process as fine-grained entities is crucial for improving the quality of adaptation. Therefore, we motivate the need for novel self-adaptive architecture styles that treat the knowledge elements as a first-class concern and drive the adaptation process based on fine-grained knowledge representation.

2. Decentralised self-adaptive architectures have not been explored in SLA-based cloud research. Existing decentralised architecture rely on distributed agent-based thinking [144] rather than prominent self-adaptive architecture as promoted by the architectural community. We argue that work in decentralised self-adaptive architecture should be built upon in SLA-based cloud research as it offers a simplified and principled way of reasoning about the complex interaction between components at various levels of abstractions in the cloud.

2.4 Summary

This chapter systematically reviewed the landscape of SLA-based cloud research with the view of answering research questions which are pertinent to this thesis. From the systematic review, it was found that MAPE-K [100] and its variants are the prominent self-adaptive architecture style in use in SLA-based cloud research. The result also indicated that knowledge representation at the architecture level and decentralised self-adaptive software architecture have received little attention.

Findings from the review provide evidence to support the claim that existing architecture styles offer limited primitives for granular knowledge representation and design-time trade-off analysis of self-adaptive architectures. Ergo, we motivate the need for a novel architectural approach that addresses these limitations.

From the foregoing limitations, our research pursues the goal of exploiting principles in self-awareness to arrive at new architectural patterns that caters for fine-grained knowledge representation and decentralised control (chapter 3 and 4). Our reputation-aware market-based mechanism complements the novel architecture patterns by providing scalable and robust coordination of components in decentralised architectures (chapter 5). The novel architectural patterns are used as foundation for architecting an exemplar federated online shopping cloud application (chapter 5) and qualitatively compared to two classic architecture styles to unveil its strengths and weaknesses (chapter 6).

CHAPTER 3

ARCHITECTURE-BASED SELF-ADAPTATION STYLES

> "All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change."
>
> *Grady Booch*

3.1 Overview of the Chapter

In chapter 2, using a systematic review methodology we have identified limitations of existing self-adaptive architectures for service level management in cloud computing. In this chapter, we conduct a deeper study of architecture styles for designing self-adaptive systems. More formally, "an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined" [79]. A style defines a collection of architectural design decisions that are applicable within a given context (e.g. problem domain), the constraints on a particular system within that context, and elicits the beneficial qualities to be realised in the resulting system [160]. When architecture styles are specialised for a particular problem domain, they are sometimes referred to as *reference architectures* [79]. Since the focus of this thesis is on self-adaptive architectures, we therefore use the terms

'architecture style' and 'reference architecture' for self-adaptive systems interchangeably.

Architecture styles are a useful way of specifying, designing, building, analysing, and evolving a software system relative to some constraints or trade-offs. By adopting an architecture style, a software architect can reason about the functional and non-functional requirements of the system-to-be designed and the associated trade-offs. Software architects are not mandated to faithfully implement every aspect of the architecture style upon which their system is built. Rather, architecture styles provide a set of guiding principles and rationale about *what* is achievable, *how* trade-off in design decisions can be analysed, and their impact on stakeholders' quality concerns. In practice, software architects are pragmatic in the selection and instantiation of architecture styles. Key considerations are the constraints imposed by the software system-to-be designed, the characteristics of the users and the deployment environment in which the system will be deployed.

Researchers have conducted surveys of models, methods, and architectures for self-adaptive systems, e.g., [60][89][132][143][42], however, none of these studies reflect on architecture styles from the perspective of their applicability to the cloud computing domain. As an example, the software engineering roadmap on self-adaptive systems [111] motivates the importance of control loops in the engineering of self-adaptive software systems and presents architecture patterns within the context of the MAPE architecture style. This effort complements the work presented in this chapter, since we study, in-depth, the general principles underlying a broader set of architecture styles and compare them in order to assess their adaptive capabilities.

As it was observed in chapter 2, many of the existing work on self-adaptive architecture for service level management in cloud are instances of the MAPE architecture style. It is worthwhile to consider the potential benefits or drawbacks of realising alternative architecture styles. Consequently, this chapter takes a broader perspective than previous surveys by studying representative self-adaptive architecture styles, followed by a comparative analysis based on metrics that define their adaptive capabilities. Following the comparative analysis, we deduced gaps in the state of the art.

Specifically, the contributions and structure of this chapter is as follows.

1. We define a framework, in section 3.2, for classifying the literature on self-adaptive architecture styles.

2. We study prominent architecture styles in self-adaptive software system domain (section 3.3). In each case, we identify the objectives of the style, discuss its pros and cons, and review examples of its application to various problem domains.

3. We compare the studied architecture styles and qualitatively measure their strengths and weaknesses within the context of our classification framework (section 3.4).

4. Gaps in state of the art self-adaptive architecture styles are identified, and their implication for service level management in cloud elicited (section 3.5).

3.2 Scope and Classification Framework

Architectural approaches for designing self-adaptive systems make use of the fundamental ingredients of a software architecture [134], i.e. components and connectors, to reason about the adaptation mechanism (managing system) and the system being adapted (managed system) and the interconnection between them. Components are computational entities endowed with functional and non-functional properties, hence it is possible to assess their suitability for different context of use. Connectors, on the other hand, are conduits that facilitate flow of control, objects, and messages between components. This distinction between the roles of components and connectors in the classic sense makes it possible to reason separately about the computational and communication requirements of a software system.

By viewing a software from an architectural level of abstraction, it is easier to reason about the various compositions of components and connections that are able to realise prescribed specifications. Therefore, the architectural view permits a broad scope of

reasoning about a software system without bothering about the low-level details, e.g. algorithmic or programming language, of how the software system is implemented. The presence of tools for specifying architectures in the form of Architecture Description Languages (ADL) and verifying the correctness of their specification and properties makes the architectural approach the preferred one [42].

Research has been conducted in communities that specialise in designing algorithmic techniques for implementing self-adaptive capabilities based on inspiration from non-computing domains. Notable examples of such venues are International Conference on Autonomous Agents & Multiagent Systems, International Conference on Self-Adaptive and Self-Organizing Systems, and IEEE Transactions on Evolutionary Computation. Inspired by nature (e.g.[119] amongst others), socio-economics (e.g. [27][108] amongst others), and biology (e.g. [51][151] amongst others), researchers in these communities contribute intelligent algorithms which are adaptive, scalable, and robust in myriad of scenarios. The interested reader is referred to the survey work of [60] to learn more about research in these areas.

It is worth noting that while problems studied in the aforementioned communities have the same overarching objectives and underlying characteristics (e.g. scalability, robustness etc.) as those studied in this thesis, the purpose of this chapter is to study the principles underlying the building blocks of an autonomic system (i.e. its architecture style) regardless of the computational technique in use. Taking an architectural view to self-adaptation has several benefits. According to [45]

> "As an abstract model, an architecture can provide a global perspective of the system and expose important system-level behaviours and properties. As a locus of high-level system design decisions, an architectural model can make a system's topological and behavioural constraints explicit, establishing an envelope of allowed changes and helping to ensure the validity of a change."

In order to classify contribution in the space of architectural styles for self-adaptation we characterise their adaptability properties. Characterising adaptability is important in order to: i) understand the impact of adaptability on system's goals, ii) analyse the trade-

off between system's adaptiveness and other QoS (e.g. availability, performance), and iii) engineer appropriate level of adaptability for system to self-manage itself at run-time. Architecture metrics for characterising adaptability can be divided into two categories: quantitative and qualitative metrics.

It is infeasible to quantitatively assess an autonomic system at the architecture style level of abstraction, since there is no implemented system in place. Consequently we aim to understand the underlying principles underpinning self-adaptive architecture styles using qualitative measures. Even in implemented self-adaptive systems, it may be hard and expensive to compare quantitative metrics, for example those proposed by [133] [58] [94] and [136], when such systems differ in their adaptation goals. Whereas, qualitative metrics such as level of separation of concern and in-built support for learning are easier to infer and characterise.

We propose a classification framework as shown in figure 3.1 for characterising adaptability properties of self-adaptive architecture styles qualitatively.

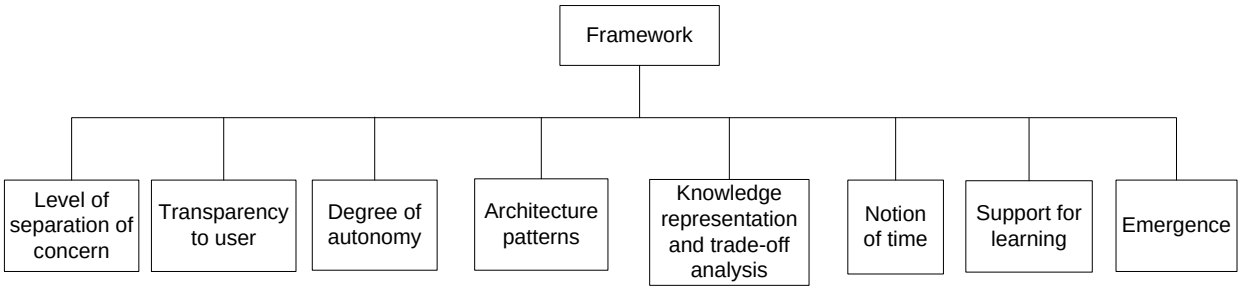


Figure 3.1: Classification Framework for Self-adaptive Architecture Styles

The qualitative metrics shown in figure 3.1 are defined as follows.

1. **Level of separation of concern:** The self-adaptive system typically consists of an adaptation engine and a managed element [100]. However, in some cases, the adaptation logic could be dispersed in the functional logic of the system, thereby blurring the distinction between the adaptive and non-adaptive parts of the system [168]. This criterion specifies the level of separation between the adaptive and non-adaptive part of the architecture style. Permissible value for this criterion are

n-level(s) of separation, where n is 1,2,3, etc.

2. **Transparency to users:** This measures the extent to which the architecture permits human interference in the adaptation loop, either by allowing them specify the goal of the system or adaptation logic. Also, the ability of the autonomic system to self-report the rationale for its actions to users is also a dimension of transparency. Transparency is measured using values: fixed goals, changeable goals, and self-reporting.

3. **Degree of autonomy:** This criterion determines to what extent the architecture style supports varying level of autonomous behaviour, ranging from zero autonomy to fully autonomous behaviour. Organic computing distinguishes itself by clearly discouraging the idea of fully autonomous behaviour [146]. In contrast controlled self-organisation is promoted in order to provide the external agent (human or automated) specifying the goals of the organic system to switch the autonomic behaviour on or off as desired.

4. **Architecture Patterns:** The self-adaptive architecture instantiated from an architecture style can often be organised in a variety of patterns [23] such as centralised, decentralised, hierarchical, and master/slave [168]. This criterion specifies the ability of the style to realise different architecture patterns. We rely on empirical evidence of implementation of the styles, no speculative claims, to determine the value for this criterion.

5. **Knowledge representation and trade-off analysis:** This criterion determines how the architecture style stores knowledge about the managed system and the adaptation process, and how this knowledge is used to perform run-time trade-off analysis between conflicting adaptation concerns. Permissible values for this criterion are implicit and explicit knowledge representation.

6. **Notion of time:** Each of the sub-components of a self-adaptive system has the

responsibility to manage the timing of its action. For example, monitoring may have to take place at fixed intervals or intervals that have to be learnt by the monitoring mechanism. In the case of hierarchical architecture patterns, adaptation actions at lower levels usually proceed quicker than adaptation actions at upper levels. This criterion specifies the provision of the architecture style to manage timing elements of self-adaptive system's subcomponents.

7. **Support for learning:** To adapt correctly in an ever changing, unpredictable environment, learning is imperative. This criterion specifies the support the architecture style provides for learning i.e. if there is in-built support for learning or none.

8. **Emergent self-adaptation:** Determines to what extent the architecture style support bottom-up architecture of self-organising systems that exhibit emergent autonomic behaviour. That is, adaptability is realised by the aggregation of simple, local actions of decentralised subcomponents, rather than through a centralised orchestral. Emergent self-adaptation can be seen in nature [99], a notable example is ant colony's foraging activity. Where present, we indicate the existence of emergence as a form of self-adaptation.

We reiterate that while other approaches to self-adaptation are useful, this thesis adheres to an architectural perspective to self-adaptation given the transparency of reasoning about the adaptation process.

3.3 Representative Self-adaptive Architecture Styles

In the subsections that follow we zoom into the specific properties of self-adaptive architecture styles and assess these properties with respect to the requirements of modern SLA-based cloud architectures. These styles describe the principles behind the systems being adapted using component and connector representations. Adaptation changes are typically in the form of architectural reconfiguration i.e. replacement of component/connector,

or modification of component/connector properties (e.g. their parameters), and rearrangement of architectural topology.

3.3.1 IBM's Autonomic Computing Reference Architecture

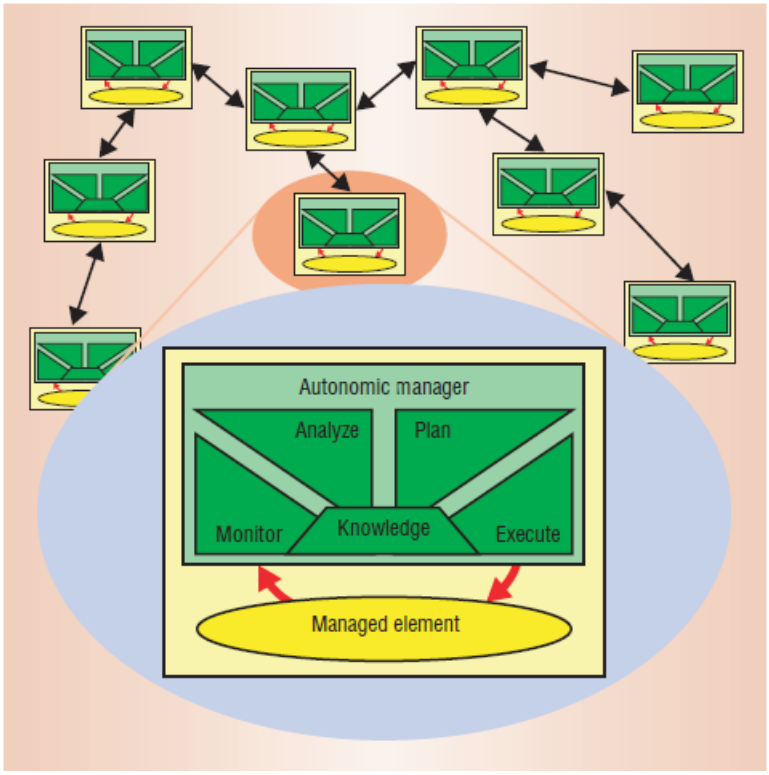


Figure 3.2: MAPE-K Architecture Style. Source: [100]

The conceptual architecture of this style has at its core the notion of an *autonomic element* that is responsible for managing its internal state and is able to interact with other autonomic elements and the external world in which it operates. As shown in figure 3.2, the autonomic element consists of two subparts: a *managed element* and an *autonomic manager*. The managed element is typically a non-autonomic computing system that requires coordination and control from an external source to meet the objective(s) specified by its owner. The autonomic manager is responsible for monitoring (M) the operations of the managed element via sensors, analysing (A) the sensed data, planning (P) alternate course of action(s), if necessary, and effecting or executing (E) the planned action(s) via actuators on the managed element. Additionally, there is a shared knowledge repository

(K) which can be used to encode shared rules, constraints, or the knowledge acquired by the other four phases. The popular MAPE-K acronym used to describe this architecture style is due to the five phases of the autonomic manager.

Autonomic elements have a structural organisation that resonates that of the human body. Similar to the anatomy of the human body that progressively builds up from molecules, cells, tissues, organs, and limbs to whole human bodies, autonomic elements are expected to operate at different levels of abstractions. Depending on the level of abstraction at which an autonomic element is operating, the type of behaviours it can exhibit and the scope of its relationship with other autonomic elements will vary. Kephart and Chess [100] posit that at lower levels, the range of behaviour and relationship to other autonomic elements are expected to be relatively static and hard-coded. At the higher levels, goals and relationships will be more dynamic and flexible, with autonomic elements exhibiting greater freedom in their behaviours and compositions or relationships.

Pros

- Adheres to the separation of concern principle by separating the five phases of the adaptation process into distinct components.
- More sophisticated patterns can be derived from the fundamental MAPE-K components [168].
- There are several implementation examples to learn from e.g. robotics [140], databases [169], software product line [1], gaming [120], and transport systems [165], just to name a few.

Cons

- Largely limited to applications requiring centralised autonomic control as interaction between autonomic managers is not explicitly modelled.
- Autonomic manager is limited to the knowledge encoded by experts such that the system cannot evolve beyond this knowledge.

- The level of granularity of knowledge (K) component in the adaptation loop is coarse. The style does not advocate modelling knowledge into fine grains to allow reasoning about adaptation along multiple knowledge dimensions.

3.3.2 MAPE based Architectures

Two styles which are sub-classes of MAPE but generic enough to be classified as representative styles with distinct qualities are presented in this section. While bearing resemblance to MAPE, these styles have been the focus of many publications, as they represent first efforts of realising self-adaptive systems based on architectural principles.

Oreizy's Architecture

The work of Oreizy et al. [129] proposed a methodology for architecting self-adaptive software systems. As depicted in figure 3.3, the architecture framework consists of an adaptation management layer and an evolution management layer.

The adaptation layer consists of monitors for evaluating the extent to which the running system is meeting some specified system goals. For example, monitoring to ensure the performance of the running system is within some acceptable range. In the event that the evaluation indicates that the system is behaving below par, relevant plans are generated to return the system to normal operation. These plans are then passed on to deployment change phase which oversees how the plan is effected on the running system by sequencing and coordinating changes.

The evolution layer caters to ensuring changes in the running system are performed in such a way that the operation of the system is not disrupted. For example, a component that is currently serving application needs should not be replaced while in a 'busy' mode. Additionally, this layer ensures that system integrity and consistency are not violated during the change process.

Pros

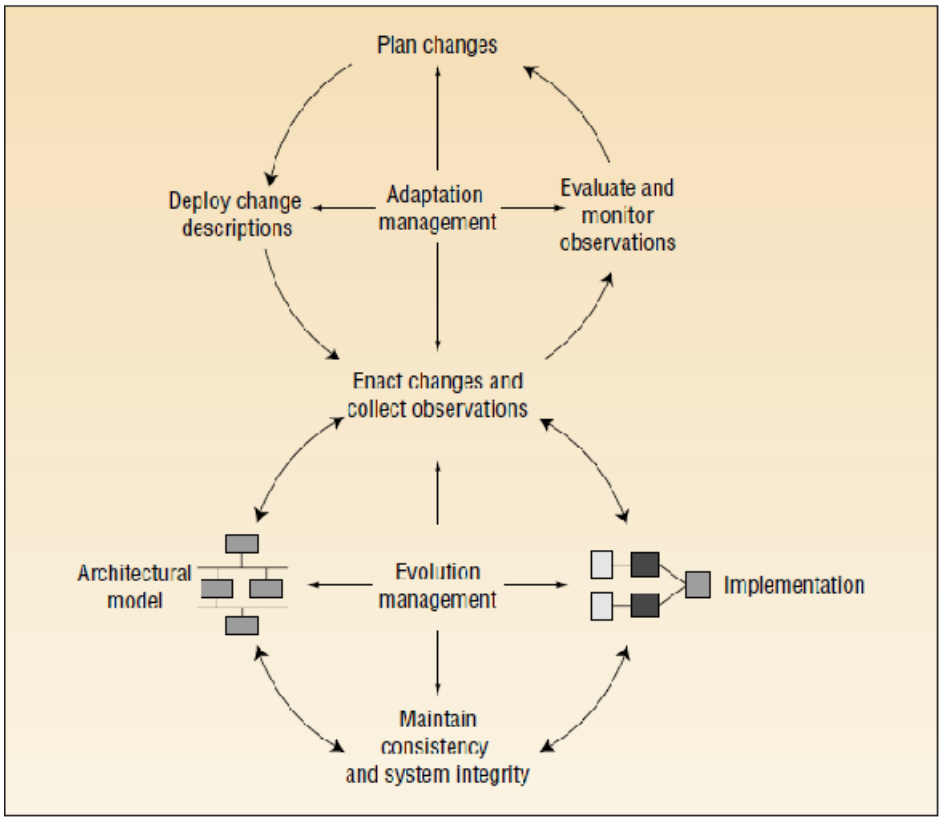


Figure 3.3: Oreizy et al. Architecture Style. Source: [129]

- Separation of concerns: adaptation layer can be changed independent of application and vice versa.
- External feedback loop between the adaptation and change management layer makes the interaction between them explicit.

Cons

- Scalability of the architecture was not articulated.
- There is no consideration of timeliness of monitoring or response in the architecture.

Rainbow

In line with IBM's autonomic vision, Cheng et al. [45] [78] adopted an architectural based approach to address the problem of automating the role of human expert in the management of software-intensive systems. Rainbow architecture framework addresses the problems of having a framework that:

1. is sufficiently generic to address the heterogeneous requirements of a variety of domain specific software systems
2. incorporates the self-adaptive (managing system) in a cost-effective way, promoting reuse across a variety of applications

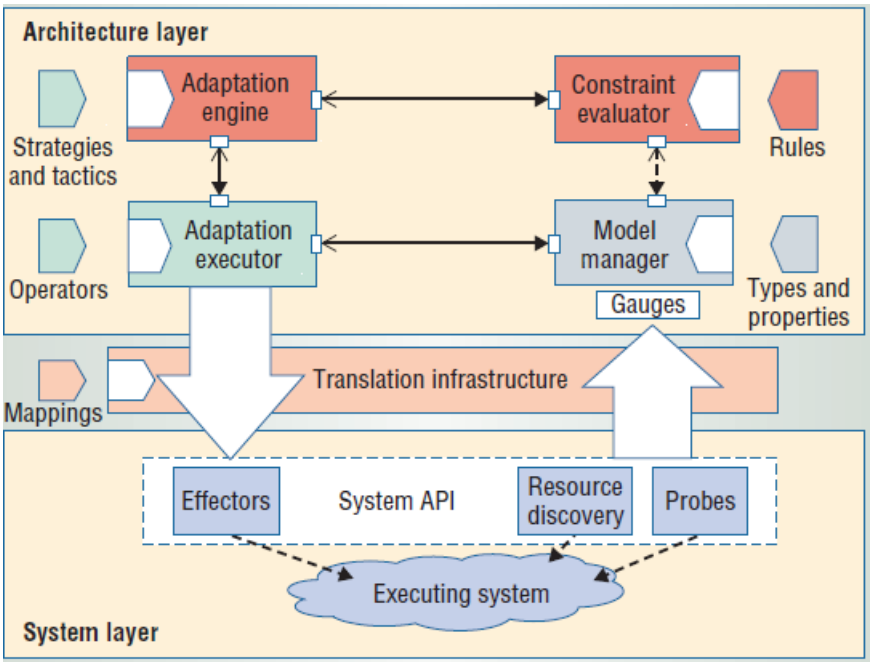


Figure 3.4: Rainbow Reference Self-adaptive Software Architecture. Source: [45]

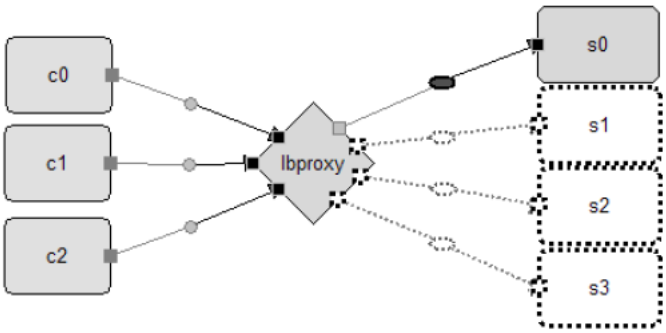


Figure 3.5: Architecture of Znn.com news application. Source: [44]

An abstract architecture model, as depicted in figure 3.4, is used to model the behaviour of the managed system. The software architecture is representative of the components and connectors in the managed system. The idea is to monitor properties of the managed system, evaluate the properties according to some rules to detect violation of desirable qualities, and effect adaptation actions in the event that a violation occurs. Rainbow relies on hooks to monitor properties of the managed system and effect adaptation changes.

A notable implementation of Rainbow is the Znn.com news website system presented in [43]. Znn.com's architecture, as shown in figure 3.5, consists of a N-tier (in this case 3-tier) web-based application, i.e. a presentation layer, application server layer, and database backend layer. Clients, depicted as "c" in figure 3.5, make stateless request for news content (e.g. pages containing videos, images, and/or data fetched from databases). Servers, depicted as "s" in figure 3.5, render pages within specified response time, and can be load balanced to decrease/increase the number of servers to ensure resources are maximally utilised. The problem of self-adaptation is to ensure that response time constraint for rendering user requests are not violated while minimising the number (cost) of servers, "s", providing the service. The adaptation strategies is such that in the event of spikes in workload, users are served content at a degraded quality (e.g. textual pages instead of images/videos). Evaluation of Rainbow architecture within the context of the znn.com application revealed that it is able to adapt cost-effectively to varying conditions [44], albeit only within the limits of human expertise encoded in the system [5].

Pros

- Separation of concerns: adaptation layer can be changed independent of application and vice versa.
- Knowledge is explicitly represented and consists of the architecture model, adaptation strategies, and utility preferences [44].

Cons

- Centralised: scalability and single point of failure is a problem.
- Non-extensible, fixed self-contained adaptation strategies.
- Little support for injection of new strategies by human.
- Reactive, since adaptation is only triggered after detected constraint violation, and not proactively anticipated.
- Deployment can be cumbersome if hooks (monitors and effectors) are *not* present in managed system or if number of required hooks are many.
- Timeliness of monitoring or response in the architecture was not considered.

3.3.3 3-layered Self-managed Architecture

Inspired by Gat's three layered architecture in the AI robotics domain [80], Kramer and Magee presented a conceptual three layered architecture [105] for self-adaptive software system (see figure 3.6). In accordance with the separation of concern principle, each layer is responsible for different aspects of the self-management control loop.

- **Component Control**: layer consists of components whose composition realises the functional and non-functional properties of the managed system. Components are able to monitor characteristics of the system being managed, for example by

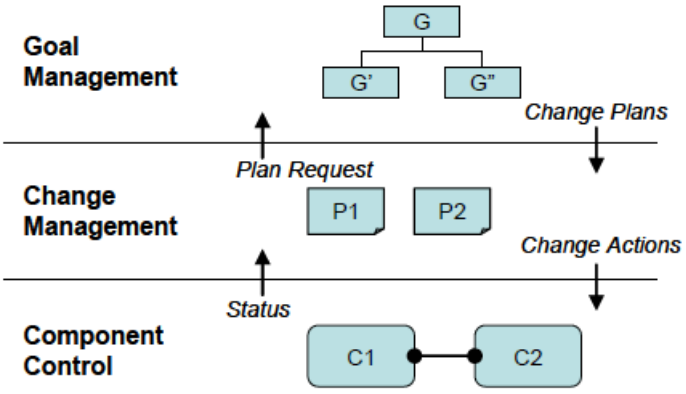


Figure 3.6: Self-Managed Architecture. Source: [105]

performing health checks for security breaches at predefined intervals. Similarly, components are able to effect changes to the managed systems in accordance with the overall specification of the managed system.

When an undesirable scenario arises, such as detection of a threat, components effect actions to ensure the system's overall specification is not violated. When components are unable to handle the detected scenario, it is cascaded to the Change Management layer.

- **Change Management**: acts as a bridge between the upper layer that encompasses the system's goals and the lower layer that reports scenarios that are beyond its control. In response to new scenarios from the lower layer, the change management layer inspects a set of pre-computed plans that correspond to different mappings of components in the lowest layer. By selecting a plan, the change management layer effects an adaptation action(s). Adaptation actions could involve adding, deleting, or updating components and/or their interconnections as required. The process of inspecting, selecting, and enforcing pre-planned adaptation strategies is a *reactive* mechanism. In the event that no plan exists to cater to a detected scenario, then the goal management layer is invoked to generate new plan for the novel scenario. Moreover, when new goals or objectives are specified at the goal management layer, the requisite plans for realising the goals are added to the change management layer.

- **Goal management**: is dedicated to generating new plans for scenarios that are not currently addressed by pre-computed plans at the change management layer. Additionally, administrators specify new goals or objectives for the managed system at this layer. For example, new goals may be injected to reflect agreed service level objectives with customers.

Skyes et al. [155][156] implemented the 3-layered style using an automatic reactive planning approach [82]. In their approach, users are able to specify high-level goals, which may impose functional and/or non-functional restrictions on eventual component selection to meet the goals. From these goals, reactive plans are generated, where a reactive plan consists of condition-action pair that determines the behaviour reified by the goals. The plans are provided as input to an architectural controller which derives the component configurations that satisfy the high-level goals specified by users. Their approach suffers from two shortcomings, namely, (1) component configuration only caters to functional properties, i.e., non-functional and structural constraints are not considered in their architectural adaptation, (2) they do not account for the state of components (e.g., idle, busy) during reconfiguration, therefore, it is possible to render components dependent on the adapted component orphan while adaptation is taking place.

Furthermore, Sykes et al. [155] posited that it is infeasible to completely specify and analyse adaptation configurations for every scenario a self-adaptive system is likely to encounter in its lifetime. That is, the specification of a pre-computed set of plan for all architectural changes the system will require is non-trivial. Crucially, in large-scale cloud applications, the large number of components and their execution modes makes any static predefined configuration ineffective at managing cloud dynamics.

These limitations were improved upon in [157], where non-functional attributes of components were monitored and utilised as a basis for composing components. Further, to account for component control in a decentralised deployment scenarios, Sykes et al. [158] proposed the use of gossip protocol [59] within the 3-layered architecture as the basis for coordinating component assembly in a scalable and efficient manner.

Pros

- Separation of concerns (goals → change management → component control): the relationship between actions carried out by components at lower levels are logically separated from the higher level specification that encompass the system's goals.
- Knowledge about each concern is captured at the layer where the concern is addressed in the adaptation loop.
- The style accounts for timeliness of decision-making (time-awareness) via two interacting external feedback loops. Component control (lowest layer) responds quickly to changes from the managed system, while successive higher levels require more time to deliberate on adaptation decisions.

Cons

- Scalability: although in [158] scalability limitation of the style is addressed using a gossip protocol, this mainly scales data sharing between components when they are cooperating towards achieving a common overarching system goal. In applications where each self-adaptive subsystem is concerned with its self-interested goal, which may conflict with other subsystem's goals, there is no provision for resolving such conflict in this approach.
- Knowledge representation is implicit, hence run-time trade-off analysis is limited.
- No in-built support for learning.
- Largely applicable in centralised deployments [167].

3.3.4 Decentralised Reference Architecture

Motivated by the limitation of existing self-adaptive architecture styles, e.g. [45], that assume a central oracle will be responsible for managing the actions of other nodes or

subsystems in a self-managed system, Weyns et al. [167] proposed a reference architecture for decentralised self-adaptation. They proposed an approach to engineer distributed self-adaptive software systems in which centralised control is not possible.

In their approach, a self-adaptive system is conceived as a collection of *local* self-adaptive systems. By visualising the system as a sum of subsystems rather than a single unit, autonomous control exists at the subsystem level. Each subsystem consists of a self-adaptive unit and a local system that it manages to achieve some specified goal. The self-adaptive subsystem is equipped with elements of the MAPE to enable it autonomously control the part of the broader system under its control. In contrast to predominantly centralised architectures, this architecture style makes explicit provision for decentralised coordination (control) among independent subsystems. Figure 3.7 shows the style.

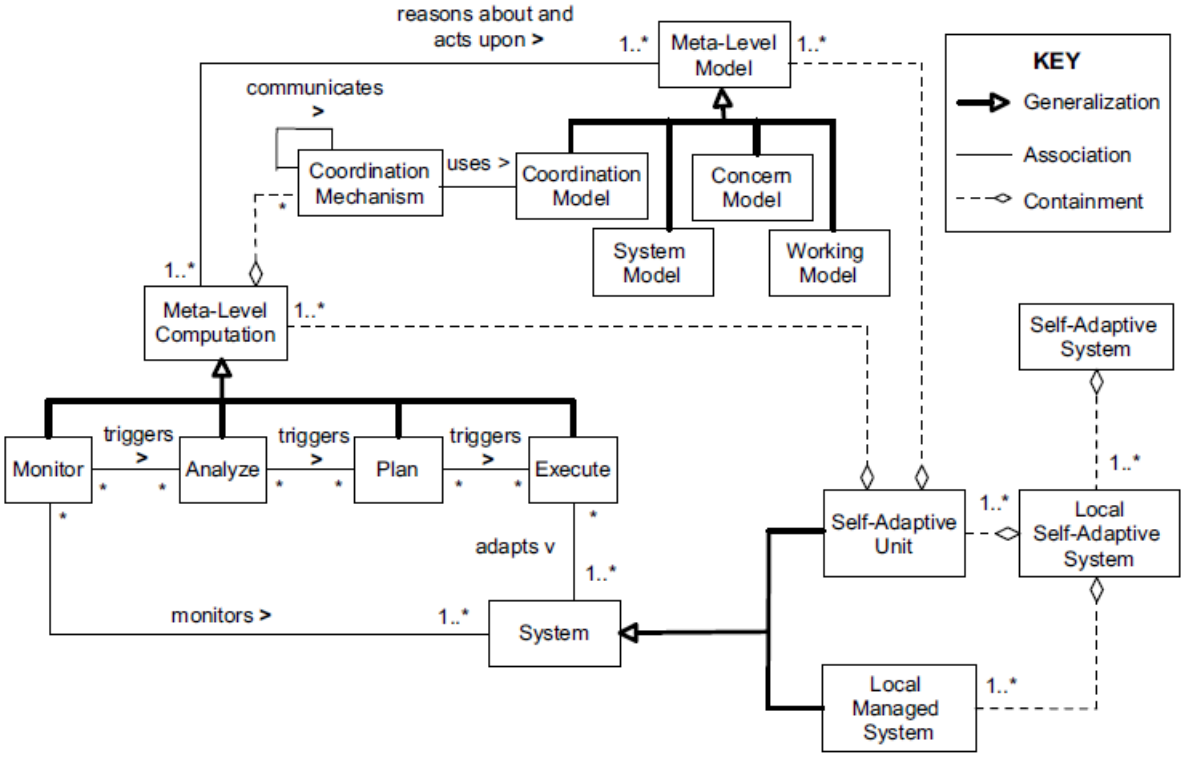


Figure 3.7: Decentralised Reference Architecture. Source: [167]

Similar to MAPE, the monitor, analyse, plan, and execute operations are represented in the architecture. These operations are referred to as meta-computations in the language of the style, which make use of meta-models to reason about concerns of the managed

system, its goals, and data structures used for information sharing [167]. Crucially, the *coordination mechanism* included in the adaptation loop provides communication and control facilities to allow one local self-adaptive unit coordinate with other self-adaptive units. The coordination mechanism and its associated model provide the underlying decentralisation capabilities for this architecture style.

Using the case study of a traffic monitoring system [167], the architecture style is instantiated as shown in figure 3.8. The objective of the traffic monitoring system is to detect traffic jam on the highway. To realise this objective, cameras (subsystems) are deployed in distributed manner, with as little overlap between their visibility region (field of view) as possible, to ensure maximum coverage of the highway. Each camera monitors the part of the traffic within its field of view and collaborates with other cameras in a de-centralised fashion to detect traffic patterns. The problem they addressed is the detection of failure of individual camera and recovery, i.e. self-healing, in the system. By exploiting a rule-based (event, condition, and effect) coordination mechanism to effect detection and correction, a de-centrally coordinated architecture is realised. Other domains that can benefit from decentralised architecture are mobile e-Learning systems [83] to coordinate mobile devices used by students in educational activities and federated cloud systems to coordinate clouds distributed across multiple sites.

Pros

- Separation of concerns: monitor, analyse, plan, execute, and coordinate operations can be easily reasoned about in terms of the meta-level computation and meta-model used.
- Scalability: removal of central bottleneck in the architecture makes it possible to scale the system.
- Robust: the decentralised style ensure no single point in the architecture is weaker than others.

Cons

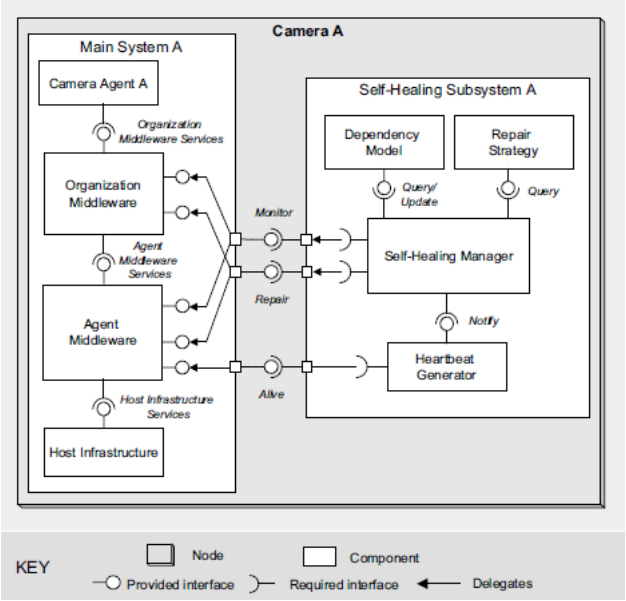


Figure 3.8: Decentralised Reference Architecture of Traffic Monitoring System. Source: [167]

- Knowledge is modelled only in terms of interaction and goal thereby reducing run-time trade-off analysis for other concerns such as timeliness of response.
- No explicit support for learning.
- There is no systematic guidance to instantiate the style. For example, architectural design decisions about how to resolve conflict between two concerns, such as overhead of communication (among decentralised subsystems) and system's performance, cannot be reached in a methodical manner.

3.3.5 FUSION

Elkhodary et al. [64] proposed FUSION, a framework for tuning self-adaptive software system at run-time as shown in figure 3.9. FUSION uses feature-based approach and online learning for analysis and adaptation. The approach is capable of coping with uncertainty, unanticipated changes and QoS trade-offs through learning and feature adaptation. The unit of adaptation is a feature, i.e. a functional or non-functional abstraction of a capability provided by the system.

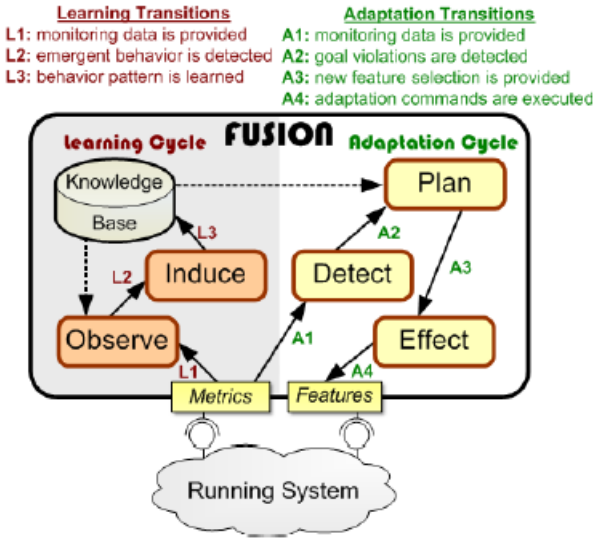


Figure 3.9: FUSION Framework. Source: [64]

Pros

- Adheres to separation of concerns principle.
- Has in-built support for learning, hence it is able to manage uncertainties.
- Using feature models reduces engineering effort compared to using analytical models such as neural networks.

Cons

- Scalability: the approach is mainly suited to centrally controlled systems. The FUSION (managing) subsystem could become a bottleneck in large scale systems.
- Brittleness: in a distributed deployment, where the FUSION self-adaptive unit manages many systems, it will become a single point of failure as the distributed system grows in size.
- Closely tied to feature modelling and reinforcement learning.
- System goals are relatively fixed once specified.

3.3.6 Observer-Controller (OC)

The goal of organic computing is to design computing systems that exhibit autonomic behaviour through *controlled self-organisation* [146]. By controlled self-organisation, this means the degree of autonomy exhibited by the system is managed by an external controller, typically the user of the system, such that the system does not manifest unanticipated emergent behaviours. In essence, the system is always accountable to the human operator who specifies the goals of the system.

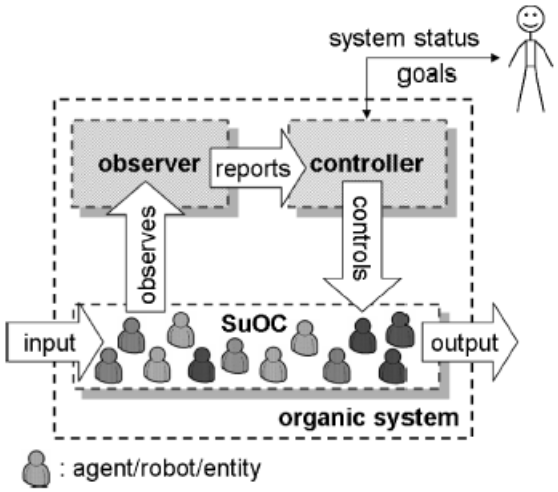


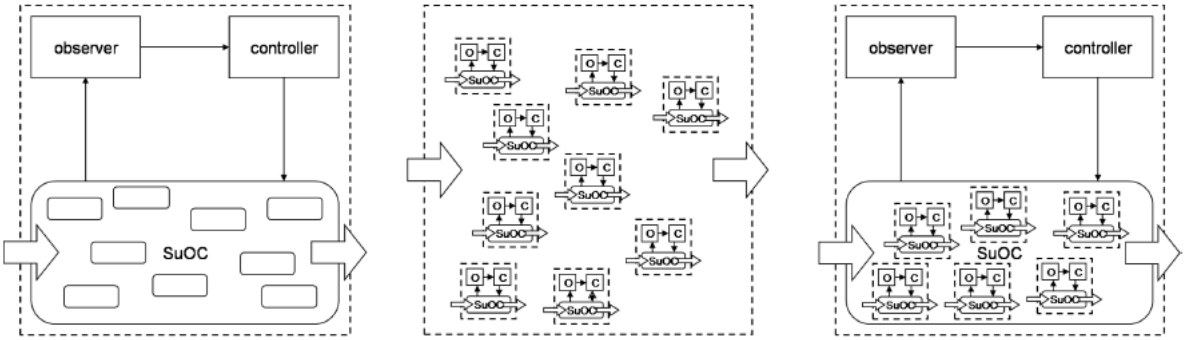
Figure 3.10: Observer-Controller Architecture Style. Source: [146]

Figure3.10 shows organic computing's Observer-Controller (OC) architecture style. The managed system in this case is described as *the system under observation and control* (SuOC), which takes input from the environment and produces output according to its functional objective. The *observer* component acts as a monitor, recording observations from the SuOC and its components, the recorded observations are aggregated, and reported to the *controller* component. The decision whether to adapt the SuOC or not is taken by the controller after evaluating the reported observation. In the event that the SuOC needs to be adapted, the controller effects the adaptation action as required.

As earlier mentioned, the distinguishing characteristics of the OC paradigm is the notion of discouraging full autonomy. The adaptation loop is always reporting its status to the external controller (e.g. a human operator), who is able to steer the system if an

emergent behaviour is discovered. This has the advantage of making users feel inclusive in the feedback loop, and making them trust the organic system. On the downside, consistently involving users contradicts the rationale for self-adaptation, therefore a balance of user involvement and autonomy is required to realise the benefit of transparency.

The OC architecture style can be organised to derive patterns for centralised, decentralised, and hierarchical control as shown in figure 3.11. In the centralised pattern (figure 3.11(a)), one pair of observer and controller component is responsible for the adaptive behaviour of the SuOC, which may itself not be a centralised system. On the other hand, the decentralised pattern (figure 3.11(b)) consists of organic systems, each responsible for different SuOCs, and may not be connected to one another. The hierarchical/multi-level pattern (figure 3.11(c)) consists of lower level organic systems that are controlled by a higher level observer-controller pair. That is, the lower level organic systems are the SuOC of the higher level organic system in the multi-level pattern. Principles of the observer-controller style have been used to architect applications in domains such as traffic light controller [20], robotics [140], and off-highway machines [172].



(a) Central (b) Decentral (c) Multi-level

Figure 3.11: Observer Controller Architecture Patterns. Source: [146]

Pros

- User involvement in the adaptation loop is explicitly captured by making it possible for user to steer the adaptation process, thereby tuning the degree of autonomy and interrupt the system if it gives rise to undesirable emergent behaviour.

- Adheres to separation of concern principle by separating the adaptation layer (observer / controller) from the managed system, i.e. SuOC.

Cons

- User involvement could become a burden, if the system frequently requires user's approval before effecting adaptation plans generated by the controller.
- Knowledge about the adaptation process is implicit in the controller and hence trade-off analysis largely depends on computational model used to implement the controller.
- The concern about difference in time at different levels of abstraction (e.g. the system and the adaptation loop) is not addressed by OC style.
- In the hierarchical and decentralised patterns of the OC style, the interaction between components of the interacting control loops is not made explicit [165]. That is, it is unclear how the observer and controller subcomponents of the interacting organic computing systems interact to realise the objective of the distributed system.
- There is no in-built support for learning, although some instances of the style make use of specialised learning models. For example [140] used a multi-layer learning approach for coordination of a society of robots, while [172] made use of machine learning for the management subsystem of an off-highway machine.

3.3.7 Dynamic Data-Driven Application System

Dynamic Data-Driven Application System (DDDAS) [53][54][130][61] is a paradigm for engineering self-adaptive systems that relies heavily on the measurement of data about the application (managed system), making decisions (e.g. predictions) based on the measured data, and injecting/impacting the managed system to ensure it operates within some bounds of desired behaviour. The managing system in this paradigm does the online or

archival measurement and prediction, typically a simulation of the application behaviour is used for this purpose. On the other hand, the managed system or application is also able to steer the measurement process, in a way that meets its application goals [53][54].

Simulating the adaptation actions before effecting them on the managed system has the advantage of mimicking likely scenarios, which may be too expensive or critical to test on the real system, in a safe and controlled environment. In essence, by simulating the adaptation actions beforehand, there is a higher guarantee of correct adaptation. The outcome of the simulation process can then be used to steer the application being managed appropriately. The feedback control loop of the DDDAS architecture style is shown in figure 3.12.

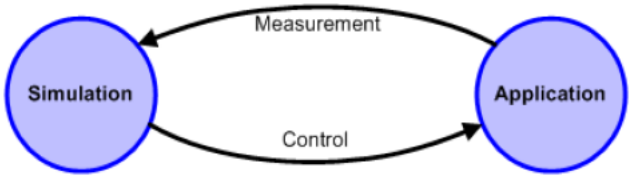


Figure 3.12: DDDAS feedback loop. Source: [128]

In [11], the idea of using DDDAS paradigm for QoS optimisation whilst minimising power consumption was proposed. Although no implementation was presented, the work demonstrates at a qualitative level how a DDDAS-based software architecture could be realised to meet QoS and power objectives.

Liu et al. [118] proposed a cloud architecture (see figure 3.13) for managing virtualised resources in a cloud. At the low level of the architecture stack, virtualised compute, storage, network resources, load balancer, scheduler, and admission controllers components are present. Each low level component is associated with an autonomous worker agent that controls the execution of the component. At the high level management layer, management processes perform scenario-based what-if simulation evaluation of management actions before updating rules to control action of low level resources. The DDDAS paradigm is realised by the simulation approach and bidirectional coordination between managed resource (low level) and the managing system (management layer). Knowledge

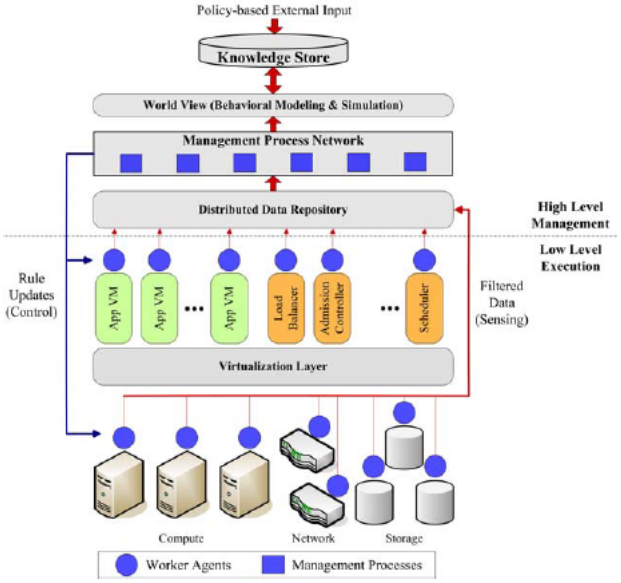


Figure 3.13: DDDAS-based Autonomic Cloud Architecture. Source: [118]

sharing is done via a distributed data repository which stores state of low level resources and informs adaptive actions at the high level management layer. In chapter 6, we present our instantiation of the DDDAS architecture style to autonomously manage SLA compliance in a federated cloud [73].

Pros

- Simulating adaptation actions before impacting them on the managed system has the benefit of improving the guarantee of adapting correctly.
- Real-time update of the simulation via measurement is useful to keeping the managing system consistent with the view of the environment and emergent behaviours in the managed system.
- Adaptation policy or strategies can be added or modified by human administrator at run-time. For example, the policy-based external input in figure 3.13 provides a means of updating the rules used to adapt low level execution components.

Cons

- Evaluating adaptation actions using simulation has the downside of being infeasible when the computing system is resource-constrained. Also, for highly perturbed

systems, there is an additional overhead of keeping the state of the simulation and managing system consistent.

- Application goals are relatively static.
- Knowledge representation is not explicitly modelled in the architecture style, although instances can make knowledge representation and its interaction with components of the feedback loop explicit (e.g. [118]).
- Has no in-built support for learning.
- Mostly suited to centralised systems where a central oracle (at the managing layer) having a global view of the system is responsible for managing the managed subsystem.

3.4 Comparative Analysis

We document the key findings deduced after studying the styles in the aforementioned sections. We classify these styles using well-defined qualitative adaptation metrics presented in section 3.2. While some metrics can easily be assessed based on inherent properties of a style, we rely on empirical evidence from concrete implementations of the styles to measure others. Table 3.1 presents the outcome of the comparative analysis.

Table 3.1: Comparison of Architecture Style for Self-adaptive System

Arch. Style → Metric ↓	MAPE-K	Oreizy	Rainbow	3 Layered	Decentral-ised Arch	FUSION	Organic Computing	DDDAS
Level of separation	1-level: Adaptation layer is separated from managed element	1-level: Adaptation layer is separated from application	1-level: Architecture adaptation layer is separated from managed system layer	4-levels: Separation of 3-tier adaptation layer from managed system	3-levels: Separation into meta-adaptive layer, adaptive layer, and managed system	1-level: Separation of adaptation layer from managed system	1-level: Separation of adaptation layer (observer / controller) from managed system (SuOC)	1-level: Adaptation layer (simulation) is separated from managed system
Transparency to users	Fixed goals	Fixed goals	Fixed goals	Changeable goals	Fixed goals	Fixed goals	Changeable goals, Self-reporting	Fixed goals
Degree of autonomy	Fully autonomous	Fully autonomous	Fully autonomous	Fully autonomous	Fully autonomous	Fully autonomous	User-driven adaptation	Fully autonomous
Architecture patterns	Centralised	Centralised	Centralised	Hierarchical	Decentralised	Centralised	Centralised, Decentralised, and Hierarchical	Centralised
Knowledge representation and trade-off analysis	Explicit knowledge representation in the K component	Implicit	Explicit representation of knowledge about adaptation strategies and utility preferences [44]	Implicit knowledge representation at each layer of abstraction: plans (upper), adaptation strategies (middle) etc.	Implicit	Explicit knowledge-base to support learning process	-	Implicit (in simulation). Some instances make use of explicit knowledge repositories (e.g. [118])

Emergence is peculiar to Organic computing as its proponents advocate the importance of emergence as a form of autonomic behaviour [124]. The Organic Computing paradigm

Table 3.2: Comparison of Architecture Style for Self-adaptive System (continued)

Arch. Style → Metric ↓	MAPE-K	Oreizy	Rainbow	3 Layered	Decentralised Arch	FUSION	Organic Computing	DDDAS
Notion of time	-	-	-	Adaptation feedback loop at lower layers are faster than higher layers	-	-	-	Simulation runs ahead of the managed application
Support for learning	No in-built support	No in-built support	No in-built support for learning	-	No in-built support	In-built support	No in-built support but implementations may realise learning using other mechanisms e.g. [140] [172]	No in-built support, although the simulation may implement learning algorithms
Emergent self-adaptation	-	-	-	-	-	-	Identified as potentially useful [124] but no principled approach was provided [146]	-

is often described as one that affords *controlled self-organisation* since the user is the principal in the adaptation process. However, there was no architecting principle to guide the reasoning of emergence, and ultimately this view of autonomic behaviour was critiqued as one that does not "suffice to characterize the essential attributes and mechanisms of (controlled) self-organization and adaptivity" [146].

3.5 Gap Analysis

This section highlights the gaps in state of the art self-adaptive architecture styles.

3.5.1 Goal-awareness

Most architecture styles assume run-time architecture goals are relatively *static*. This should not be confused with adaptation strategies which are dynamic in most cases. For example, Rainbow [45] provides mechanisms for encoding expert knowledge in an application domain as adaptation strategies, and selects one of them using decision theory at run-time [42]. The prevalence of static goal representation can be traced back to early efforts to implement autonomic systems that perform relatively fixed human administrative tasks in a cost-effective fashion. In those early systems, the goals were relatively fixed, with adaptation predominantly focused on selection of appropriate strategies (at run-time) to meet these fixed goals.

[146] and [105] are exceptional styles in this respect because they make run-time goals explicit, thereby making it possible to change user preferences and constraints. In particular, [146] makes the role of the human explicit in the adaptation loop and provides a feedback loop from the system to the user, i.e. status reporting (see figure 3.10).

Modern software systems such as cloud-based applications are deviating from the assumption of fixed goals. For example, users of a public cloud are free to express their application goals (e.g. combination of SLA parameters and constraints on those parameters) in any order, depending on their business goals. In these emerging application domains, goals are considered as *dynamic* entities that can change at run-time. Changing run-time goals should have the consequent effect of changing the adaptation strategies used to achieve the updated/new goal(s), without violating system constraints. As rightly identified by Sawyer et al. [145], involving humans in the feedback loop opens opportunities to design *self-explaining* autonomic systems that can both adapt themselves and explain their adaptation decisions , e.g. using scenarios, to end users.

3.5.2 Time-awareness

We argue that time should be treated as an explicit adaptation concern in self-adaptive architecture styles, and should not be left to the application designer as part of their implementation concerns. We explicate this position by considering the simplest autonomic system, where a central autonomic manager adapts a single computing node. Here, the notion of time-awareness can be traced to the following questions: (1) how often should the managed system be sensed (monitoring)? (2) how often should changes be impacted on the monitored system (execute)? (3) should the adaptation loop rely on historic data or anticipate future events while making adaptation decisions? It is clear that depending on the notion of time, different architectural configurations would suffice. Architecture-level primitives that aid reasoning about how *time* affects the architectural design decisions are crucial.

In distributed systems, the problem of timing is exacerbated, as coordinating autonomic managers, working in a cooperative setting, to adapt the managed subsystems require a notion of time that is consistent. This concern is often left to the application designer, for example a heartbeat timing mechanism is used to coordinate decentralised cameras in [167]. With the exception of [105] and [53], where time is explicitly considered as an adaptation concern, most architecture styles treat time as an implicit aspect of the adaptation loop.

In [105], the speed of adaptation at each layer of the hierarchical architecture progressively reduces as it commences from the component control layer (responsible for monitoring/execution), to change management layer (performs strategy selection), and goal management (plans new strategies for emergent scenarios). It is unclear how this architecture caters to the timing requirement of decentralised architectures. The bidirectional co-adaptation approach in [53] assumes the measurement subsystem (simulation) runs ahead of the application (managed system) being adapted. The rationale of the difference in time here is to allow the managing subsystem time to assess the impact of various adaptation actions before impacting the system.

3.5.3 Interaction-awareness in Decentralised Architectures

Most of the architecture styles are specialised for centralised control [129][45][64][53] and hierarchical control [105]. The work of [81], although not instantiating any of the reference architectures in this chapter, realised decentralised control via a broadcast coordination mechanism, which suffered from scalability limitations. Sykes' [158] implementation of a decentralised architecture based on the 3-layered style [105] is largely due to the use of the gossip coordination mechanism. Hence, we agree with Weyns' [167] claim that the choice of coordination mechanism is crucial in a decentralised architecture. Furthermore, the style could make the realisation of such mechanisms easier by providing primitives to implement and reason about coordination.

The exception is the decentralised reference architecture [167], which is specialised for decentralised control, and makes the coordination mechanism and its associated model explicit. Although variants of [146] can be structured to realise centralised, hierarchical, and decentralised patterns, it does not make the coordination among the interaction autonomic system (termed organic system) explicit. As earlier mentioned, a noteworthy effort in architectural pattern cataloguing is [168], where 5 architecture patterns derived from interacting MAPE loops were proposed. However, the patterns are limited in their representation of run-time goals and knowledge is represented at a coarse-grain level, hence limiting run-time trade-off analysis.

The state of the art in self-adaptive architecture would benefit from a style that supports *decentralisation by design*, making interaction among interacting autonomic subsystems explicit as well as fine-grained knowledge representation. Additionally, there is no systematic methodology to instantiating any of the self-adaptive architecture style. This contrasts to practice in more mature software architecture pattern communities [79][23], where methodical processes for understanding and instantiating reference architectures are given special attention.

3.5.4 Fine-grained Knowledge and Trade-off Analysis

It was observed that there was no unified, principled approach about representing the *knowledge* that a self-adaptive system should encode in other to realise its objective. In [45], the focus of knowledge is the representation of adaptation strategies, while knowledge in [64] is essentially a supporting mechanism to facilitate the learning capabilities of the architecture. In [105], knowledge is distributed according to the concerns on different layers of the hierarchical architecture (although it is not made explicit). Other architecture styles make the knowledge concern of adaptation an implicit concern. Application designers may violate this implicit assumption by making knowledge explicit to aid the architecture of their system. As an example, [118] used a distributed knowledge repository to manage interaction between the low level execution layer and high level management layer in a DDDAS-based cloud architecture.

We argue that such coarse grain and unstructured representation of knowledge has the effect of misguiding application designers on which principles to adopt for encoding different knowledge concerns. In particular, we believe that by making knowledge explicit and represented at a fine-grain level to address concerns for time, goal, and interaction, reasoning about time-awareness (as discussed in section 3.5.2), goal-awareness (as discussed in section 3.5.1), and decentralisation (as discussed in section 3.5.3) and the trade-off between them will be significantly improved.

3.6 Summary

In this chapter we studied representative architecture styles for designing self-adaptive software systems. In particular, the studied architecture styles are measured with respect to the degree of adaptability they afford. From our findings, we characterised the extent to which each studied architecture style realised the adaptation metrics of interest, for example: level of separation of concerns, support for learning, knowledge representation etc. Additionally, we performed a comparative analysis to position our findings from each

architecture style relative to other styles using well-defined adaptation metrics. Furthermore, we conducted a gap analysis and found that state of the art architecture styles are lacking in their modelling of goals, time, interaction in decentralised systems, and fine-grained knowledge representation to support trade-off analysis.

Given the requirements of modern cloud service level management, we argue that a novel architecture style that fills the identified gaps and provides a methodological approach to engineering self-adaptive cloud applications is needed. Chapter 4 presents the self-aware architecture style, our novel contribution to architecture-based self-adaptation that has in-built support for learning, fine-grained knowledge representation, and eases analyses of non-functional qualities and trade-offs at multiple levels of abstraction.

CHAPTER 4

ARCHITECTURE STYLE AND PATTERNS FOR SELF-AWARE SYSTEMS

> "I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."
>
> *Alan Turing*

4.1 Overview of the Chapter

A handful of architecture styles have been contributed in line with the vision of architecture-based self-adaptation e.g. [53] [45] [105] [64] [146]. These approaches often make simplified assumptions about knowledge acquisition and representation when modelling and managing trade-offs encountered in dynamic, open systems. As a result, the quality of adaptation tends to be limited as it does not fully capture complex trade-offs arising from heterogeneity of the interacting nodes, the operating scale, openness, and dynamics of the environment.

It has been observed that "fine-grained" knowledge representation provides useful primitives for more effective design of self-adaptive architectures [67][36][98]. Decomposing knowledge about a system to finer grain raises the system's awareness about itself, i.e. its *self-awareness*. According to Lewis et al. [115] a self-aware system is one that

"...possesses information about its internal state (private self-awareness), and sufficient knowledge of its environment to determine how it is perceived by other parts of the system (public self-awareness)." Prior to Lewis et al. [115], Kephart and Chess [100] posited that self-awareness is an enabler to realising advanced autonomic behaviour.

The EU-funded FP7 project Engineering Proprioception in Computing Systems[1] (EPiCS) [13] [66] has produced results related to how concepts of self-awareness and self-expression can be used to engineer autonomic computing systems. We exploit these concepts and the *self-aware architecture style* [68] as conceptual foundations for innovating self-aware architectural patterns, which are the novel contributions of this thesis. Computational self-awareness endows a self-adaptive system with capabilities for acquiring knowledge by monitoring via internal sensors (within the self-adaptive system), external sensors (in the operating environment), and representing knowledge using learning models.

This chapter's objective is to answer the first research question (Q1) in chapter 1:

> What are the architectural patterns that can be used by software architects to design SLA compliant self-aware federated cloud applications?

Building on the conceptual foundations of self-awareness, this chapter contributes five architectural patterns for designing self-aware computing systems. These patterns adhere to well-tested architectural principles, such as separation of concerns, and provide a systematic approach for instantiating a self-aware architecture. Each pattern provides a solution to a recurring design problem in a given context [93]. By inspecting the characteristics of a pattern and exemplar of its use, architects of federated cloud applications are able to make informed design decisions about their systems. Patterns are presented in increasing order of complexity and build on one another to aid comprehension.

The contributions of the chapter are as follows:

- The self-aware architecture style is presented in section 4.2. Unlike state of the art architecture styles, the self-aware style incorporates fine-grained knowledge rep-

[1]The self-aware architecture style presented in this chapter was jointly incepted by the author in collaboration with members of EPiCS project. The work on self-aware architecture patterns is the author's original contribution.

resentation to model concerns relating to stimuli, goal, interaction, and time. It offers in-built support for learning and facilitates knowledge representation, run-time strategy selection, and reasoning about adaptation actions at a meta level.

- We advance the foundation of the self-aware style by codifying design lessons learnt from existing applications of the style in the form of patterns [68] [40]. Our novel architectural patterns (section 4.3) are generic templates that serve as guidelines for organising self-aware applications [23], therefore, aiding reuse of the style.

4.2 The Self-Aware Architecture Style

This section describes an architecture-style for self-awareness by looking at a self-aware node. A node in our context refers to the boundary of the system that is to be managed. More generally, a node has autonomy over its representation of itself and operating environment, and is able to exert its behaviour on the environment and other nodes.

It is important to clarify conceptual differences about how a self-adaptive node is perceived by the agent and architecture communities. Approaches to self-adaptation presented in chapter 3 can be classified as architecture-based self-adaptation [42]. Whereas the type of system capable of self-adaptation could be described in the agent literature as architectures of agents that are capable of learning [142]. There is a subtle difference between these two types of architectures in terms of the boundary of what is considered to be the self and level of control exerted by the environment in the relationship between the self and environment. This difference is elaborated upon as follows.

A self-adaptive software system essentially consists of two parts: (i) a managed part that can be monitored and controlled, operating in an environment which may be uncontrollable, and (ii) an adapting part that can monitor the managed part, reason about it, and adapt it to realise some goal. A software agent is essentially an autonomous system that can reason about itself and its environment and can perform actions in the environment to meet its design objectives [46].

A self-aware node as conceived in this thesis interacts with an environment, which may be a *controllable* managed system or an *uncontrollable* environment. While this difference may impact choice of architectures, in the model of our style controllability of managed system and/or environment is not given special treatment - this concern is left as design decisions when instantiating the style in various application domains. Perry and Wolf's seminar work [134] provide an instructive definition to clarify these terminologies.

> "An architecture style is that which abstracts elements and formal aspects from various specific architectures. An architectural style is less constrained and less complete than a specific architecture." [134]

This definition sheds light into why constraints, such as controllability, may be treated as soft constraints when presenting an *architecture style* as opposed to when presenting *a specific architecture* of a system. Since this chapter considers an architecture from the stylistic perspective rather than a specific architecture, therefore, the two types of controllable states identified in the agent and architecture communities may be abstracted as design decisions to consider when instantiating the style. More precisely, we view the self-aware style as one that describes *how* a self-adaptive system may be designed in the context of both controllable and uncontrollable environments.

Figure 4.1 depicts the internal composition of a constituent node in the said style. It describes its structure, interaction, and relationship with the environment. The self-aware style introduces an approach to analysis, reasoning, and management of self-adaptation and their dynamic trade-offs by decomposing knowledge according to the concerns of the system. Specifically, knowledge is decomposed into five distinct levels of awareness, which are discussed in this chapter, namely: stimuli, interaction, time, goal, and meta. The idea is to simplify architectural design and analysis of self-adaptive system by decoupling knowledge concerns across these dimensions and making the interaction between them explicit. By taking this approach, trade-off points [48] that may hinder the system from meeting its requirements are easier to identify and reason about.

Self-awareness processes are able to collect information both from internal sensors

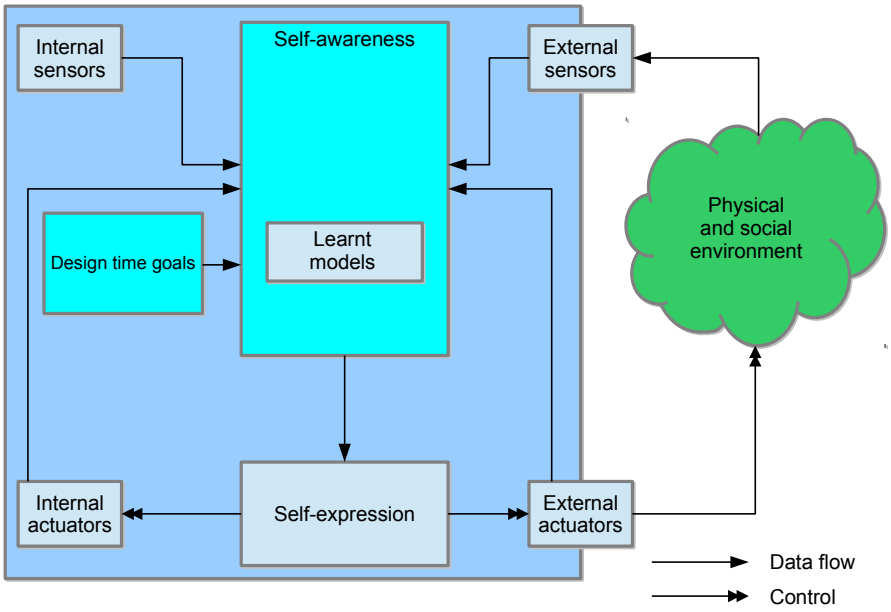


Figure 4.1: Overview of Architecture of a Self-aware Node. Source: [68]

(regarding private experiences internal to the node and typically externally unobservable) and external sensors (regarding experiences of the node's physical environment as well as of other nodes). Additionally, self-awareness processes are able to observe the actions taken by the node, and have access to goals specified for the node at design time.

Self-expression processes make use of knowledge obtained and represented by self-awareness processes and determine appropriate actions as a result. The self-expression component therefore has control over actuators. The self-expression component has no privileged direct access to the design-time goals, however in a typical instantiation, a self-awareness process will be responsible for representing goal information in a meaningful, useful and efficient manner (e.g. through a utility function), to the self-expression component. In this way, though a node may be designed with multiple complex and context dependent goals, it may possess the ability to be aware of which goals are relevant given its current context, and expose only those to the self-expression component at a given time. This separation can act to simplify the required self-expression behaviour.

4.2.1 Primitives of Self-aware Architecture Style

This section introduces two important primitives underlying the self-aware architecture style [68]: levels of self-awareness and the notion of public and private self-awareness.

Levels of Self-awareness

According to [115], there are five levels of computational self-awareness that can be used to describe a computing system's self-awareness capabilities as described below.

Stimulus-aware: A node is stimulus-aware if it has knowledge of stimuli, which enables it to respond to external entities. The node is not able to distinguish between the sources of stimuli neither can it distinguish between past or future stimuli. Stimuli-awareness is the lowest level of awareness and a prerequisite for other levels of awareness.

Interaction-aware: A node is interaction-aware if it has knowledge that stimuli and its own actions form part of interactions with other nodes and the environment. It has knowledge via feedback loops that its actions can cause specific reactions from the environment in which it is deployed.

Time-aware: A node is time-aware if it has knowledge of historical and/or likely future phenomena. Implementing time-awareness may involve the node possessing an explicit memory, capabilities of time series modelling and/or anticipation.

Goal-aware: A node is goal-aware if it has knowledge of current goals, objectives, preferences and constraints. Goal-awareness constitutes objectives of the system that are capable of changing at run-time; not hard-coded design-time goals. Utility functions and state models are examples of implementation options for goal-aware capability.

Meta-self-aware: A node is meta-self-aware if it has knowledge of its own level(s) of awareness and the degree of complexity with which the level(s) are exercised. This advanced state of awareness permits choosing between implementation options for realising lower levels of awareness. Also, it permits a self-aware system to degrade gracefully instead of failing catastrophically.

In architectural terminology, each level of self-awareness maps to a component in a

software architecture. Therefore, we are able to reason about interaction of these levels by conceptualising them as subcomponents of a self-aware node. Figure 4.2 depicts an architectural representation of a self-aware node showing how its subcomponents relate to one another.

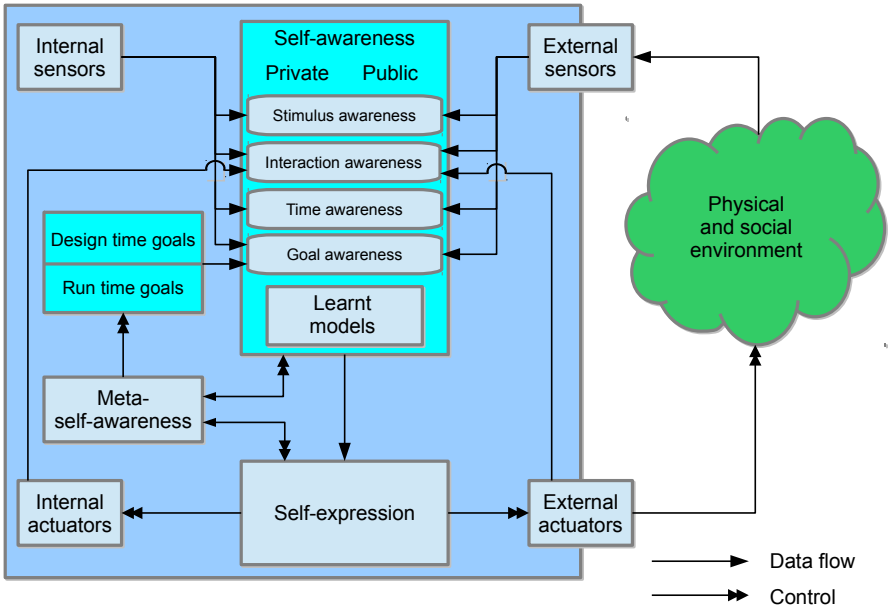


Figure 4.2: A Self-aware Node showing interaction of subcomponents realising different levels of awareness. Source: [68]

Private and Public Self-awareness

Private and public self-awareness are concerned with internal and external sources of knowledge respectively. Self-aware architecture style makes a distinction between external and internal sources of data, from which the self-awareness component constructs knowledge. Data connectors clearly establish this relationship in figure 4.2.

With the exception of interaction awareness, all self-aware subcomponents can exhibit both private and public self-awareness. Since external accessibility is a necessary prerequisite for interaction, interaction awareness subcomponent is only public.

The self-expression component of the architecture (in figure 4.2) exploits learnt knowledge to effect correct behaviour of the system using one or more strategies. These strate-

gies determine the node's actions, given the availability and state of the node's knowledge. Clearly, the types of learning carried out in the self-awareness component, and the types of models available, will have a large bearing on what strategies it is possible to learn and enact in the self-expression component.

To design a self-aware cloud application, the software architect must decide: (i) what level(s) of self-awareness to implement, (ii) the structural organisations of the level(s) of self-awareness, and (iii) the expected quality of service of each structural organisation. Next, we present our self-aware patterns that help architects make informed decisions.

4.3 Patterns for Self-aware Architecture Style

Building on the primitives of self-aware architecture style, we codify the knowledge about how to architecture self-aware applications in the form of architecture patterns in this section. We elicit five patterns, where each pattern is decentralised by design. That is, structurally our self-aware patterns resembles a peer-to-peer network of interconnecting self-aware nodes, varying only in the number of the subcomponents and the type of interconnection between them.

Until recently, architecture patterns for self-adaptive systems have received little attention [168]. Many existing patterns target specific application domains [123], limiting their reuse outside the domains where they were originally conceived. Weyns et al. [168] argued that UML notations are limited in their ability to characterise self-adaptive architecture patterns, hence they proposed a simple, generic notation for describing patterns for Monitor-Analyse-Plan-Execute (MAPE) architecture style. Our patterns are distinct in focus from Weyns' in the sense that while we model knowledge concerns in the architecture, their attention was about MAPE component interaction.

We adopt a pattern notation, similar to the one in [168] for describing our self-aware patterns. Firstly, Weyns's notation [168] is simple and easy to comprehend. Secondly, we believe describing our self-aware patterns using existing notation in the self-adaptive

community makes our work accessible to other researchers and paves the way for others to build on our work. The pattern notation is depicted in figure 4.3.

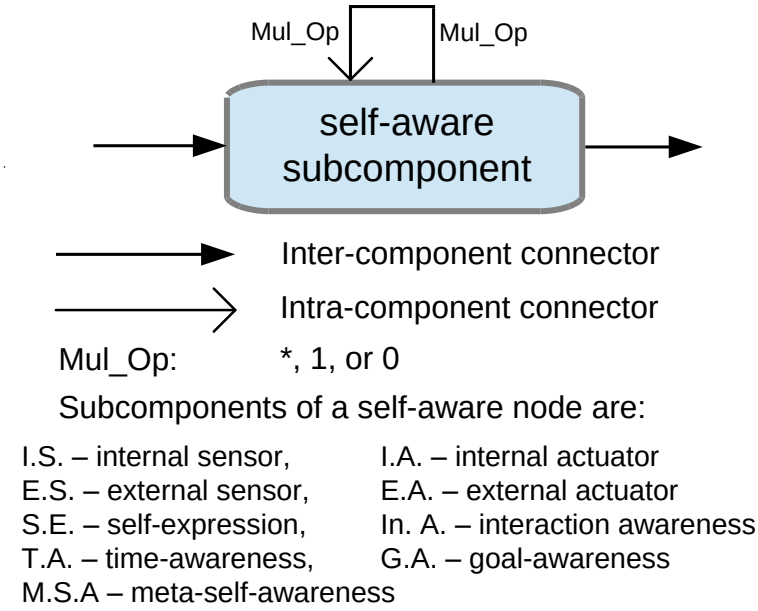


Figure 4.3: Notation for Describing Self-aware Architecture Pattern

Two types of connectors are used to express the relationship between subcomponent of a self-aware node and its relation to other self-aware node(s). The intra-component connector applies to subcomponents of the same type, while inter-component connector applies to linkage between components of different types. There are three types of multiplicity operators (mul_op). The multiplicity operator asterisk, *, signifies the many-side of a connection, while 1 or 0 indicates that one or zero connections of the type specified is permitted. E.g., a * on both sides of the intra-component arrow of a subcomponent means one or more subcomponents of that type can be linked to one another across nodes.

Stimulus-awareness subcomponent is considered an invariant in the five self-aware patterns, hence it is not shown in our pattern description, since as discussed in section 4.2.1 it is a prerequisite for any form of awareness. We document our patterns using standard pattern template [23] as follows.

- Problem/Motivation: A scenario where the pattern is applicable

- Solution: A representation of the said pattern in a graphical form
- Consequences: A narration of the outcome of applying the pattern
- Example: Instance of the pattern in real applications or systems

Next, we present the five self-aware patterns using the template described above.

4.3.1 Basic Information Sharing Pattern

Problem/Motivation. Sometimes one computing node may not be sufficient to cope with the complexity of an application or to meet the demands of users as they scale. To manage application complexity, functionalities could be divided among several self-aware nodes, where each node is specialised in a few functionalities, collaborating to provide the application's service. More self-aware nodes may also be introduced to meet the scalability requirement of the system. In each case, at the basic level, there is a need to provide a means for the nodes to interact with one another to carry out their respective roles.

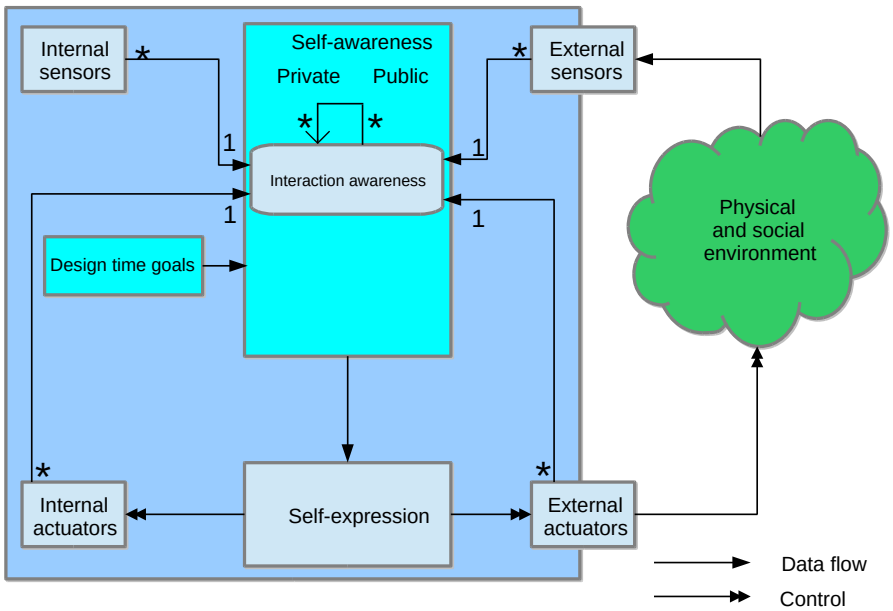


Figure 4.4: Basic Information Sharing Pattern

Solution. The simplest pattern for interacting self-aware nodes is the basic information sharing pattern. In this pattern, a self-aware node contains only the interaction-awareness subcomponent, which can be connected to one or more self-aware nodes as shown in figure 4.4. Each self-aware node may have one or more sensors (internal/external) and actuators (internal/external). The underlying characteristic of this pattern is that peers are linked only at the level of interaction-awareness.

An example of the basic pattern where two nodes are connected via their interaction-awareness subcomponents is shown in figure 4.5. Although only two nodes are shown in figure 4.5, the number of connected nodes is not limited to two. The number of nodes is limited by the scalability of the interaction mechanism. For instance, a broadcast mechanism may limit the number of interconnected nodes when compared to a gossip protocol. In practice, a node may be connected to either all or a subset of nodes in the systems depending on its role in the system.

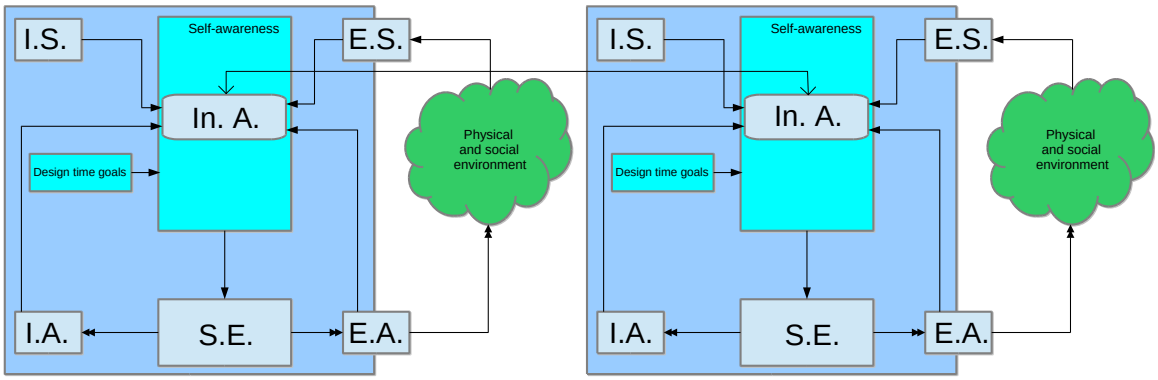


Figure 4.5: Concrete Instance of the Basic Pattern

Consequences. Self-aware nodes could use the interconnection between them to negotiate the protocol to use for communicating in a network. This pattern can also be used to facilitate sharing information among nodes about neighbourhood relation in a network.

Crucially, in this pattern each self-aware node maintains its autonomy about how to make adaptation decisions via its self-expression component. This means that each node is responsible for its interpretation and reaction to the information shared via interaction-awareness. Therefore, this pattern is not suitable for cooperative problem-solving scenar-

ios, where nodes need to reach an agreement among themselves about the best course of action for the problem. This limitation is addressed in the *coordinated decision-making pattern* (see next section). The basic information sharing pattern assumes the system's goal is preconfigured at design time, consequently, constraining the system's adaptation.

Examples. Federated datacenters and clouds, owned by distinct entities, are good candidate applications of the basic information sharing pattern. The owners of such clouds or datacenters may choose only to share status information about availability of resources or current load and not cooperate beyond this level. Thus, each cloud provider maintains autonomy over its resources while collaborating with other cloud providers in a limited way to facilitate outsourcing of resources, if required. Participants in a grid computing set-up utilise similar communication model and rely on incentive-based mechanisms to facilitate resource sharing [173].

4.3.2 Coordinated Decision-making Pattern

Problem/Motivation. Decisions made by individual self-aware nodes in a group may be suboptimal due to their limited view of the system and its operating environment. As noted in the basic information sharing pattern, individual self-aware nodes do not cooperate when making decisions. In applications requiring near-optimal and consistent global decision making in a cooperative setting, a more advanced architectural pattern may be required. In particular, such a pattern should make it possible for nodes to synchronise their self-expressive actions.

Solution. The coordinated decision-making pattern provides a means of coordinating actions of multiple, interconnected self-aware nodes. Figure 4.6 shows this pattern. It differs from the basic pattern in that self-expressive nodes are linked to one another, such that they are able to agree on *what* action to take.

Consequences. Unlike the basic pattern, given the * to 0 multiplicity on the self-expression component in figure 4.6, it is not mandatory for nodes to link their self-expression components to each other. This makes it possible for nodes to form clusters,

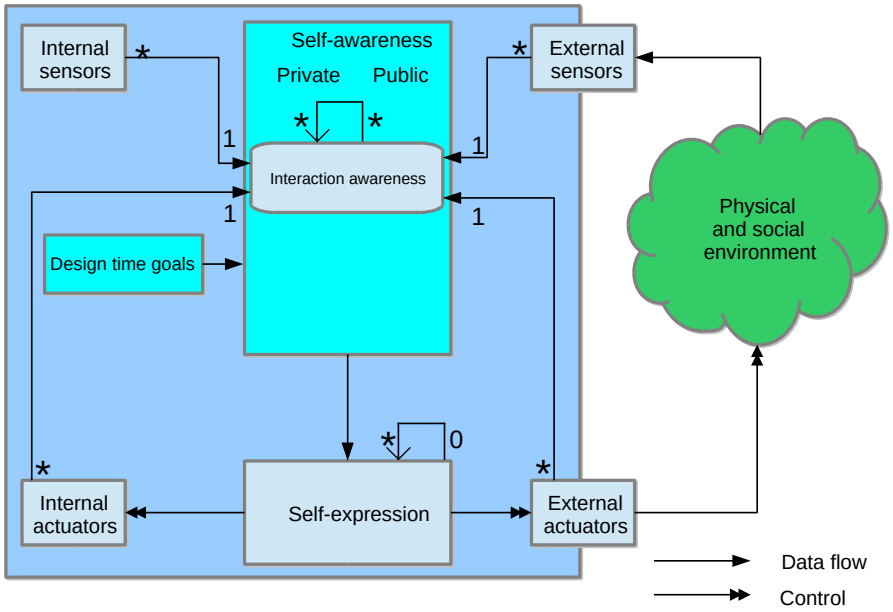


Figure 4.6: Coordinated Decision-making Pattern

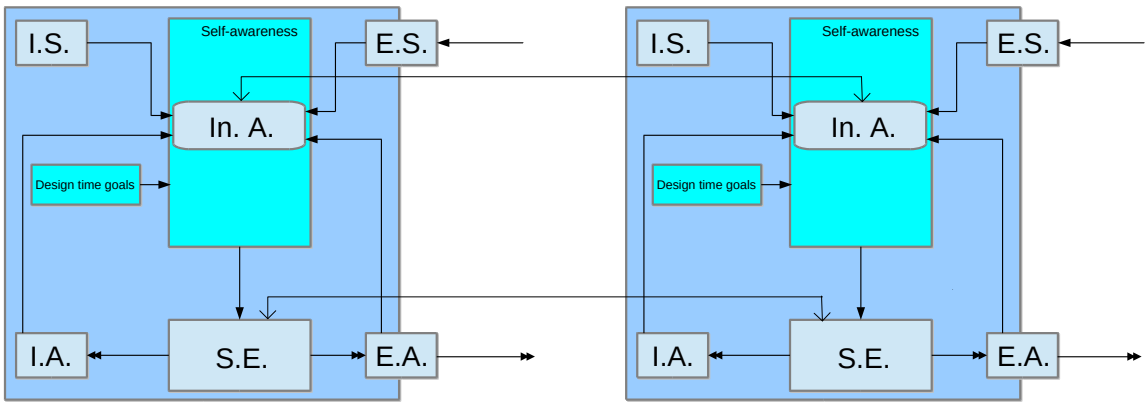


Figure 4.7: Concrete Instance of Coordinated Decision-making Pattern

where nodes in a cluster cooperate to solve problems in one part of a system, while nodes in other clusters cooperate to solve problems in other parts. Figure 4.7 shows an example where two self-aware nodes instantiate this pattern. As argued in the case of the basic pattern, using two nodes to illustrate the pattern as shown in figure 4.7 does not limit the number of nodes that can realise the pattern in a real system.

The downside of this pattern is that although nodes are able to form clusters and cooperate on *what* action to take, they are unable to decide the timing of such actions,

i.e. *when* to act. This notion of time insensitivity is addressed in the *Temporal Knowledge Sharing Pattern* (see next section). The temporal knowledge sharing pattern incorporates time-awareness capabilities into the coordinated decision making pattern.

Examples. Large-scale cloud federations where providers agree to implement unified resource allocation policies, irrespective of how such policies are enforced at individual cloud levels, are a candidate application of this pattern. In such federated clouds, policy changes are negotiated via interaction-awareness subcomponents, upon agreement the self-expression component of each cloud enforce the agreed policy within its (local) cloud.

4.3.3 Temporal Knowledge Sharing Pattern

Problem/Motivation. As stated in the previous section, coordinated decision-making pattern does not provide a means of coordinating the *timing* of actions agreed upon by cooperating nodes. This limitation may not be tolerated in applications where timing of actions has an impact on the integrity of the application. Also historic knowledge may be required to forecast future actions, in order to improve the accuracy of adaptive actions.

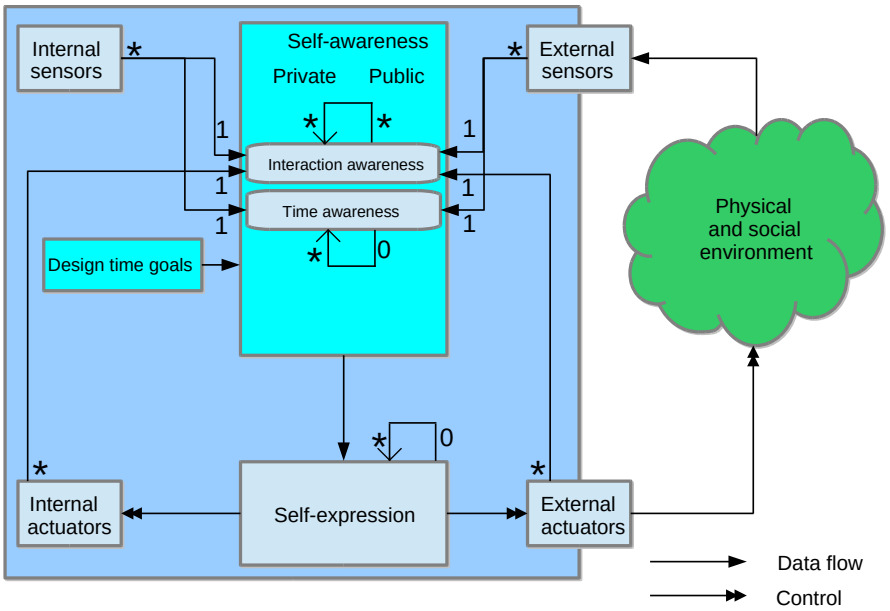


Figure 4.8: Temporal Knowledge Sharing Pattern

Solution. The temporal knowledge sharing pattern solves this problem by incorporating time-awareness capabilities into the coordinated decision-making pattern. As shown in figure 4.8, each self-aware node has a time-aware subcomponent which is, optionally (as denoted by its multiplicity), linked to other self-aware nodes to represent timing information. An example where two nodes are connected using this pattern is shown in figure 4.9. This timing information can be exploited by the self-expression component to manage the timing of adaptation actions across multiple nodes.

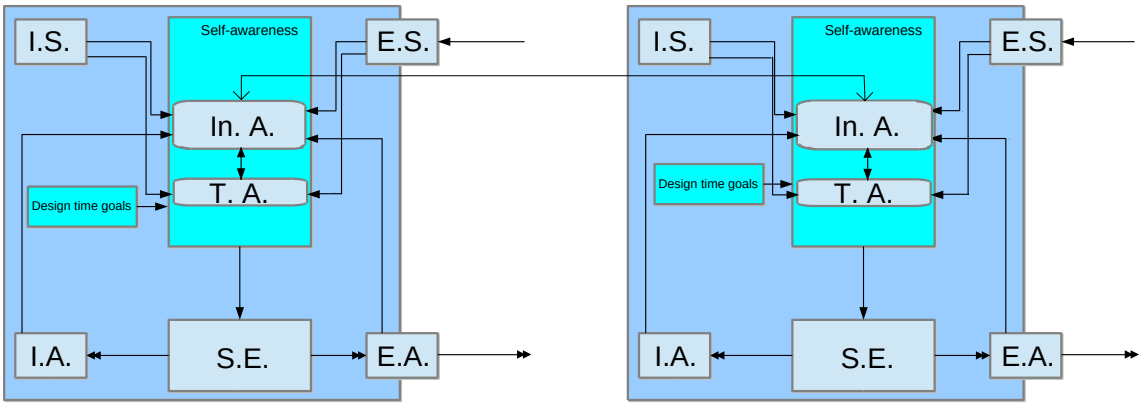


Figure 4.9: Concrete Instance of Temporal Knowledge Sharing Pattern

Consequences. The knowledge of timing information provides a rich basis to enrich the power of the adaptation action that is possible. However, there are a lot of design considerations left to the application designer who instantiates the style. For example, how often should timing information be recorded? In storage constrained systems, how long should acquired knowledge be stored for before forgetting (removing) them? Should the forgetting process be total, i.e. delete all knowledge acquired within a period at once, or selective? Depending on the concerns of the application at hand, these questions will have different answers.

It should be noted that up till now, all the patterns discussed do not cater to changing goals. That is, they assume the goal of the self-adaptive system is known at design-time and statically encoded in the system, without opportunity to modify it at run-time. The pattern discussed in the next section - *Goal Disseminating Pattern* - will address the challenge of modifying or changing goal at run-time.

Examples. Clusters in cloud datacenters, where the servers in the cluster cooperate to execute tasks assigned to the cluster head, are able to exploit this pattern. For example, a parallel scientific application assigned to the cluster, requiring coordination across different time-steps of the application could utilise the pattern to ensure actions taken in each time-step are coordinated to avoid compromising the integrity of the result.

4.3.4 Goal Disseminating Pattern

Problem/Motivation. User preferences are mostly dynamic, i.e. users want different things at different times. As an example, a user who is pleased with operating a computing system using a touch screen at one time may prefer a voice interaction mood at another time. These changes in user preferences may range from simple changes, such as mood of user-interaction, to more advanced ones. Furthermore, a computing system may itself decide to change its goal, depending on the amount of resources available to it. A federated cloud application that is unable to scale its resources may choose to satisfy SLAs of only premium users, instead of aiming to satisfy SLAs of all users. Therefore a specialised pattern that allows explicit representation of run-time goals and facilitate changes to these goals, as the system evolves, is needed.

Solution. Figure 4.10 shows the goal disseminating pattern that address the concern of representing run-time goal. A goal-awareness subcomponent represents knowledge about run-time goals, which can be changed as the system evolves. The goal-awareness subcomponent in a self-aware node can, optionally, share its state information with goal-awareness subcomponents in other self-aware nodes.

As with previous patterns, goal information sharing is not necessarily globally shared with all nodes. Hence, a subset of nodes in a system could share their goal state, while their goal information is disjoint from other nodes. It is important to note that sharing goal information is not equivalent to unifying goal state across nodes. It is possible for nodes to share goal information, while each pursues its distinct goal. The reverse scenario, where goal information are unified across nodes, is also possible.

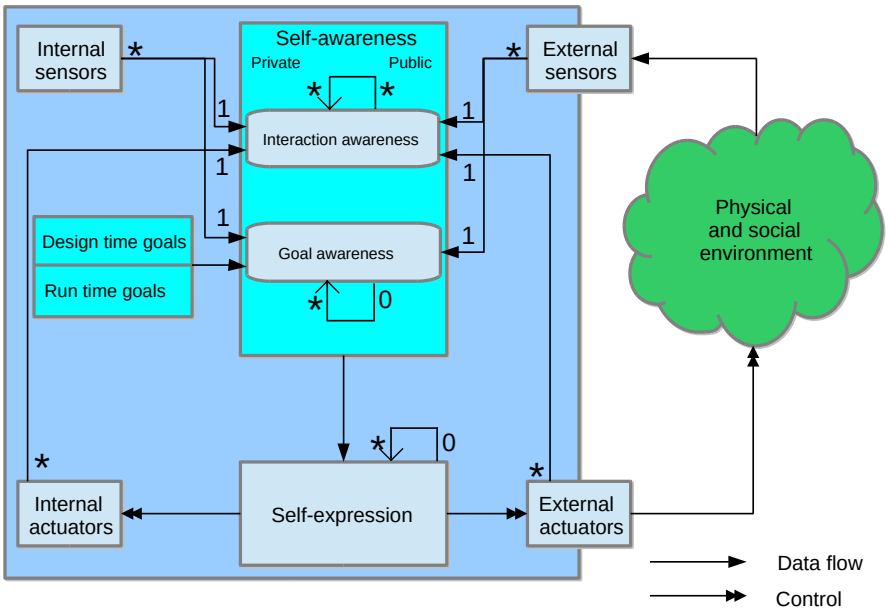


Figure 4.10: Goal Disseminating Pattern

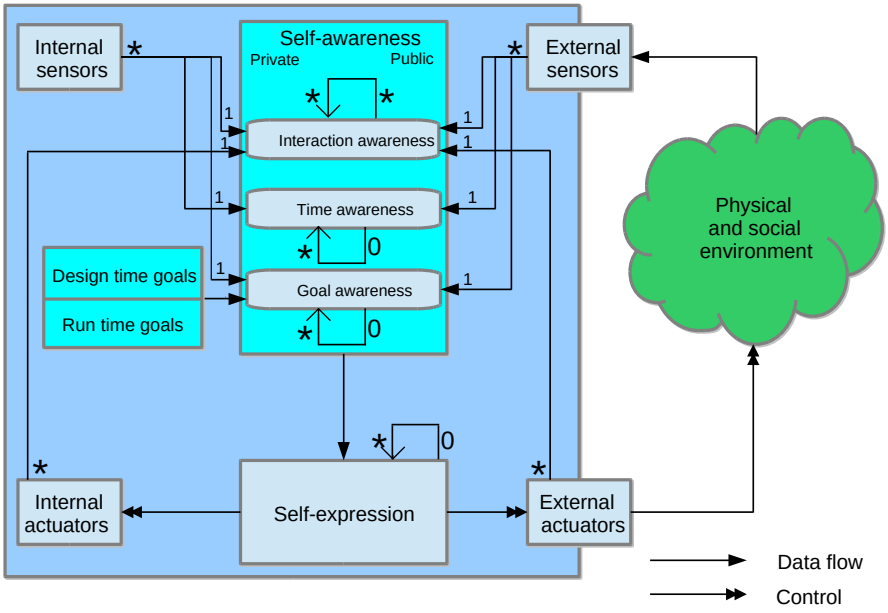


Figure 4.11: Goal Disseminating Pattern (with time-awareness capability)

Consequences. As can be observed from figure 4.10, a time-awareness subcomponent is not included in this variant of the style. This implies that time-awareness is not a necessary prerequisite for goal-awareness. While each node is able to change its goal at

run-time, it does not represent temporal information to realise the capabilities of the temporal knowledge sharing pattern. For the sake of completeness, we include a variant of the pattern that addresses this limitation (see figure 4.11). The variant in figure 4.11 makes the inclusion of temporal knowledge explicit, making it suitable for application domains where changing goals and forecasting are required.

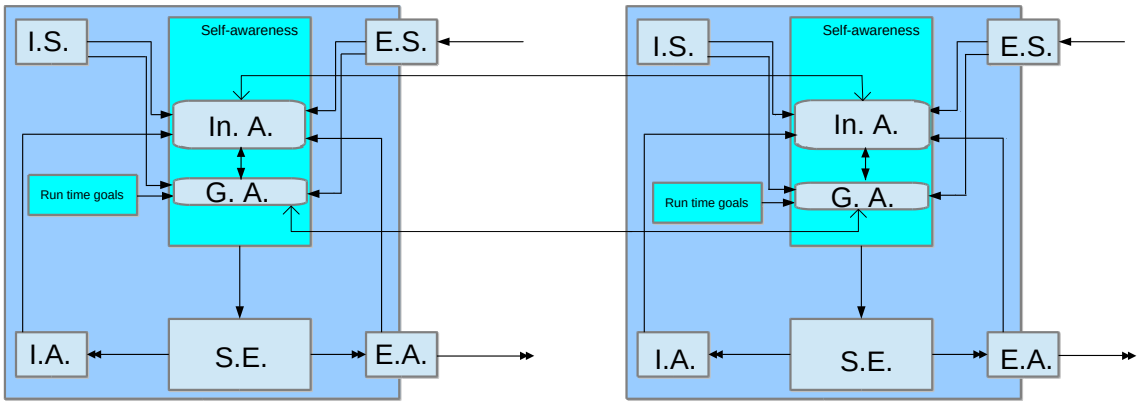


Figure 4.12: Concrete Instance of Goal Disseminating Pattern (variant 1)

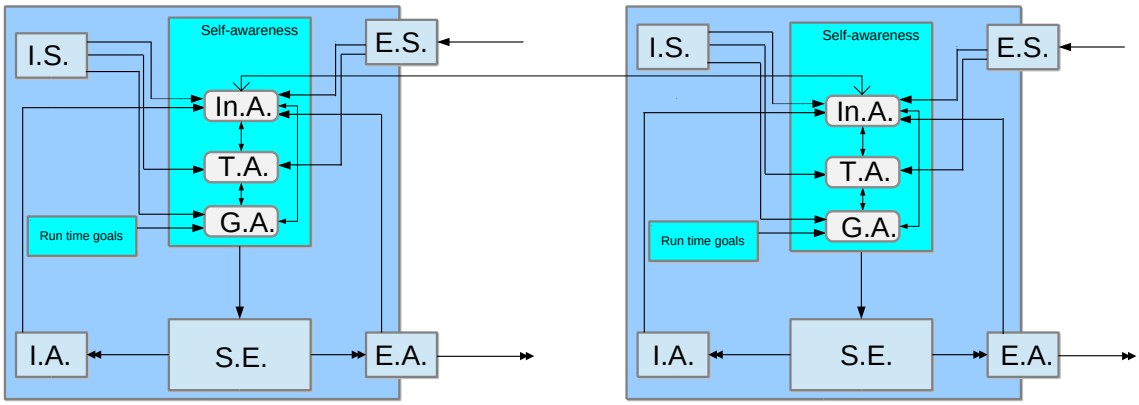


Figure 4.13: Concrete Instance of Goal Disseminating Pattern (variant 2)

In both variants of the pattern, self-expression component makes use of the goal-awareness subcomponent to make strategic decisions in line with the system's current goal. Figure 4.12 shows an instance of the pattern (without time-awareness), while figure 4.13 an instance (with time-awareness).

Examples.

Service-based applications operating in dynamic, open cloud environment are possible

candidates of this pattern. Here, applications are composed from cloud services which are selected based on QoS and cost considerations. A service that is highly performant at one time may degrade in quality at later times due to overloading of the service. Each application has service level agreement (SLAs), to which it must adhere. Application goals encoded in SLAs may themselves change as users demand different levels of service from time to time.

Using the second variant of the goal-disseminating pattern (see figure 4.11) in this scenario has the benefit of making each application capable of representing temporal knowledge about service performance and forecasting which service(s) are likely to be more dependable and long-lasting. Also, the goal-awareness subcomponent makes it possible to represent SLA terms of users and adapt such goals as they change. Lastly, by sharing temporal knowledge, applications can cascade knowledge of service performance among themselves. It should be noted that this introduces opportunities to falsely badmouth or inflate performance of services. Considerations for filtering out good knowledge are left to the computational models used to implement time- and goal-awareness.

4.3.5 Meta-self-awareness and Self-aware Patterns

As discussed in section 4.2.1, meta-self-awareness is useful for managing the trade-off between various levels of self-awareness and for modifying goals at run-time. Since reasoning at the meta level is considered an advanced form of awareness, which may be beneficial or necessary in some contexts and not beneficial in others. This section specially treats the relation between meta-self-awareness and the patterns discussed in previous sections.

We reiterate that one of the distinct benefits of the self-aware style is to reduce the complexity of modelling adaptive behaviour when compared to non-self-aware approaches. For the sake of illustration, consider the problem of modelling and tuning an online learning algorithm, e.g. neural network, for deciding actions of an application in different scenarios. It is known that this task is time-consuming and requires expertise mathematical skills, which may not be readily available [64]. Additionally, in some use cases, small

changes in the application scenarios may render the solution proffered by the algorithm invalid or incorrect - another cycle of algorithm tuning may be needed to cater to these changes. An alternative approach is to provide families of algorithms for different contexts and dynamically select the appropriate algorithm at run-time using online learning capabilities of the meta-self-aware component.

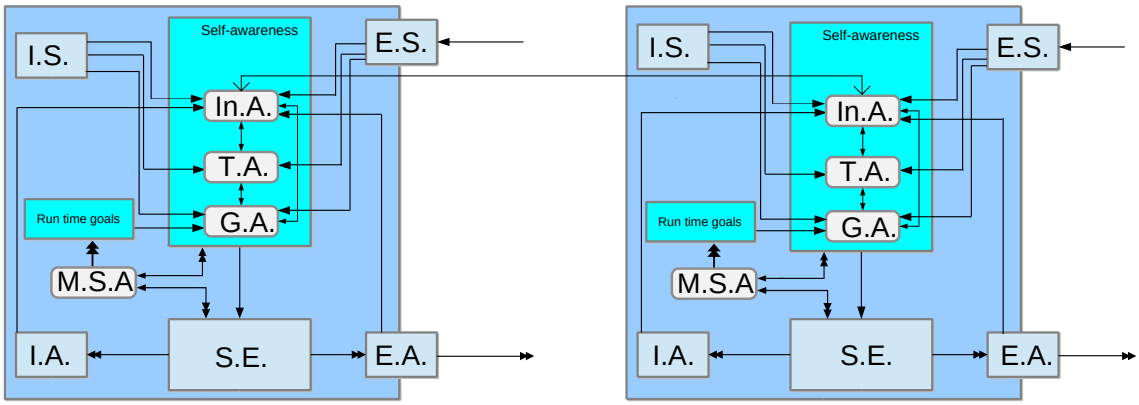


Figure 4.14: Concrete Instance of Goal Disseminating Pattern (including meta-self-awareness component)

While the first approach offers faster adaptation, if application scenarios are relatively stable, the second approach is able to better cope with complexity in highly perturbed environments, where one algorithm is insufficient to cover the scope of adaptive behaviour. Accordingly, we recommend that every pattern can optionally incorporate the meta-self-aware component depending on the complexity to be managed and expertise of the application designer. Figure 4.14 shows a variant of the goal-dissemination pattern where a meta-self-aware pattern is present to manage trade-off between goal-, interaction-, and time-awareness subcomponents.

4.4 Related Work

4.4.1 Classic Self-adaptive Architecture Styles

Similar to classic architecture styles surveyed in chapter 3 (e.g. [100] [129] [45] [105][167][64][146][53]), our self-aware style emphasises a separation between the acquisition and representation of the learnt knowledge and the decision making processes. However, knowledge in our style is explicitly modelled at a finer grain (stimuli, goal, time, interaction, and meta) than classic styles. Similar to architectural pattern approach in organic computing [146] and MAPE [168], we provide a catalogue of patterns to guide software architects. In contrast to existing work in self-adaptive architecture patterns, we explicitly model the knowledge dimension of the control loop. Since knowledge (self-awareness) is an enabler of a system's self-adaptive capabilities, our approach offers a more comprehensive coverage of the system's design space.

Crucially, with the exception of [167], classical self-adaptive architecture styles are mainly centralised or hierarchical, however our approach is decentralised by design. This offers capability to manage dynamics of large scale federated cloud applications. Aside from [105] and [53], the notion of time-awareness was not explicitly captured in existing approaches. In our approach, we provide explicit patterns to guide architects when designing time-critical applications.

Our work complements learning-inspired architecture styles such as FUSION [64]. While FUSION makes use of feature-based models and reinforcement learning for analysis and adaptation, we provide an extensible style for architects to choose models that best fit their problems. This consideration is especially important in the meta-self-aware pattern of our style as learning models implemented in self-aware and meta-self-aware subcomponents may well be different. One limitation of our work is that we have not explored the concept of emergence as a form of self-awareness as suggested by [115] and captured in organic computing [146]. This dimension is left as future work.

4.4.2 Approaches with Explicit Claim to Self-awareness

SEEC (SElf-awarE Computing) is another framework that claims self-aware capabilities [86]. SEEC relies on the (O)bserve-(D)ecide-(A)ct (ODA) [86] architecture style. The (O)bserve and (A)ct in ODA serves as monitor and executor components respectively, while analysis and planning tasks are performed by the (D)ecide component. ODA is centralised by design and does not account for the levels of self-awareness in our approach.

Zambonelli et al. [178] positioned their research on self-awareness and self-expression from a formal modelling perspective using ensembles to realise self-adaptation at both individual node and collective levels. In particular, concepts from artificial immune system and MAPE-K architecture style were used to model knowledge and realise adaptive behaviour. Similar to our work, [62] identified five levels of self-awareness namely: event-, situation-, adaptability-, goal-, and future-awareness, however, no architecture or style was suggested for reasoning about each level and how they relate to one another.

4.5 Summary

This chapter introduced the conceptual foundation of self-aware architecture style and its design principles. We described primitives of the style such as levels of self-awareness and the notion of public and private self-awareness. We advanced the foundation of the self-aware style by contributing five novel architectural patterns to codify solutions to common design problems encountered when using the style in specific contexts. We argued for the ability of the patterns to fill gaps identified in existing architecture style and justified design decisions made when incepting the patterns.

As a first step to evaluate the architectural patterns, in chapter 5, we instantiate the goal disseminating pattern within the context of the online shopping cloud application introduced in section 1.5. One of the core requirements for a quantitative evaluation of the decentralised self-aware cloud architecture is a coordination mechanism to control interaction of architectural components. In chapter 5, we propose and evaluate such a

mechanism namely *Reputation-aware posted offer mechanism* within the context of our self-aware cloud architecture. Chapter 6 takes a qualitative dimension to the evaluation of the self-aware style by relying on independent stakeholders to compare it with two classic architecture styles and analysing feedback from independent application designers on their experience using the self-aware style.

CHAPTER 5

MARKET-INSPIRED MECHANISM FOR DECENTRALISED COORDINATION

"All models are wrong, but some are useful."

George E. P. Box

5.1 Overview of the chapter

Federated clouds are characterised by distributed ownership of architectural components by multiple cloud providers where no single provider has global knowledge of the entire federation. To engineer a self-aware architecture for federated cloud applications, it is crucial to investigate computational mechanisms for coordinating the components of such a decentralised system.

The coordination of components in a decentralised system significantly impacts the system's ability to meet its design objectives. In the federated cloud application context, the underlying software architecture and the implemented computational mechanism driving interaction of components in the architecture both determine the extent to which the application can meet customer SLAs.

In Computer Science literature, coordination and control is a well-studied topic especially in the agent-based community. Notably cloud computing researchers have used

heuristics (e.g [32]), policies (e.g [89]), and optimisation (e.g [175]) mechanisms for coordinating cloud resources. Whilst these mechanisms are useful for managing interaction of cloud architectural components at small scales, they do not scale well to larger deployments. In particular, they are limited in their ability to manage trade-offs amongst design objectives in large scale deployments such as federated cloud settings.

Suppose we view the set of cloud services which may be used to compose a federated cloud application as an economy. *Market-based Control* (MBC) [47] offers a class of mechanisms which makes use of such an economy analogy to address limitations of heuristics, policies, and optimisation approaches. MBC approaches have been used for managing trade-offs amongst multiple (conflicting) objectives in distributed systems such as communication networks [107], interactive music [35], and smart camera networks [67]. Essentially MBC translates socio-economic concepts that make market institutions scalable and resolve trade-offs autonomously into computational terms. By designating components as buyers and sellers in the market economic system and coordinating their interaction using a mechanism, a robust, scalable, and decentralised system can be realised.

Given the suitability of MBC to manage federated clouds, we adopt the *posted-offer* market mechanism [101] for coordinating architectural components (section 5.3 details the rationale for our choice). This chapter's objective is therefore to answer the second research question (Q2) raised in chapter 1:

> How can market-based control be utilised to coordinate decentralised components in a self-aware architecture whilst respecting SLA compliance goals?

To answer the above question, we extend the classic posted-offer mechanism to incorporate measurement of agents' performance using reputation ratings as a way of capturing their reliability at fulfilling SLAs. The refined mechanism is referred to as *reputation-aware posted offer market mechanism*. In this chapter, we demonstrate that our refined mechanisms performs better than classic posted offer mechanism at complying with SLAs.

Researchers are beginning to explore application of economics-inspired principles to problems in cloud [24] [126] [41][76]. These research contributions can broadly be classi-

fied as using economics approach for dynamically pricing cloud resources, using economics mechanisms to facilitate negotiation between parties in a cloud market, and for dynamically allocating cloud resources to users based on SLA requirements. This thesis takes the third perspective in our treatment of economics-inspired coordination and control.

Part of the work presented in this chapter has been published in [71] and [116]. Concretely, this chapter makes the following contributions:

- A short review of MBC approaches and their application to cloud (section 5.3).
- The design of a novel reputation-aware posted offer market mechanism that meets the requirements of SLA-based cloud resource management is the focus of section 5.4 and 5.5.
- A candidate self-aware federated cloud architecture using our reputation-aware posted offer market mechanism is instantiated and presented in section 5.6.
- Empirical study of the self-aware cloud architecture and reputation-aware versus classic posted offer mechanism under synthetic and realistic workload (section 5.7).

The work presented in this chapter complements self-aware architecture patterns introduced in chapter 4, in that the market mechanism serves as a way of coordinating decentralised components in the self-aware federated cloud architecture instantiated here.

5.2 Preliminaries

First, we introduce the terms that are used in the discussions that follow.

- Agents: "are computer systems [or architectural components] that are capable of independent, autonomous action in order to satisfy their design objectives...As agents have control over their own behaviour, they must cooperate and negotiate with others in order to satisfy their goals" [46].

- Market: an institution that facilitates interaction (trading) among agents[1] by exchanging currency and (re)distributing resources. In the computational sense, markets are mostly virtual rather than physical institutions. Also, the exchanged currency is usually artificial, instead of real cash.
- Buyer: a market agent who requires some resources and is willing to acquire it in exchange for currency.
- Seller: a market agent who provides some resources and is willing to release it in exchange for currency.
- Market Mechanism: governs *how* market agents interact, i.e. interaction between forces of demand and supply, to determine prices and units of services sold in the market.
- Non-Strategic Market Agents: simply follow the rules defined by the market mechanism without adapting their behaviour based on emergent scenarios.
- Strategic Market Agents: behave rationally, by adapting in a way that give them advantage over other agents. An example of such adaptation could be to search for the most cost-effective deal in the market.

5.3 Markets and Clouds

This section presents market based approaches, their application to cloud computing, and the role of reputation management in market design.

5.3.1 Centralised and Decentralised Markets

Designers of market-based systems adhere to a centralised or decentralised approach depending on the goal of the system. In a centralised market system, a market marker or

[1]The term agent is used here, since multi-agent systems are a good way of modelling interacting market entities.

auctioneer collects *bids* and *asks* from buyers and sellers respectively (see figure 5.1a). The auctioneer matches these bids and asks in some way (depending on the mechanism), to decide the pairing of buyers and sellers. A famous and well studied centralised market mechanism is the Continuous Double Auction (CDA) [104]. For large scale systems, the auctioneer constitutes a bottleneck, and therefore limits the scalability of the system. Also, failure of the auctioneer or delays in computing results could make the system unusable. To address these limitations, some markets instantiate multiple auctioneers, either as a way of separating the load or having specialist auctioneers for different scenarios.

On the other hand, a decentralised market has no central entity (see figure 5.1b). Instead, buyers and sellers explore the market and bargain with one another to decide who to trade with, in a distributed manner. An example of a mechanism that follows this approach is bilateral bargaining mechanism [50]. While this approach is more scalable than its centralised counterpart, the search time to explore the market limits performance in a large system. Moreover, there is no guarantee that the resulting buyer-seller pairing is optimal. That is, a combination of buyer and seller does not assure either of maximum utility.

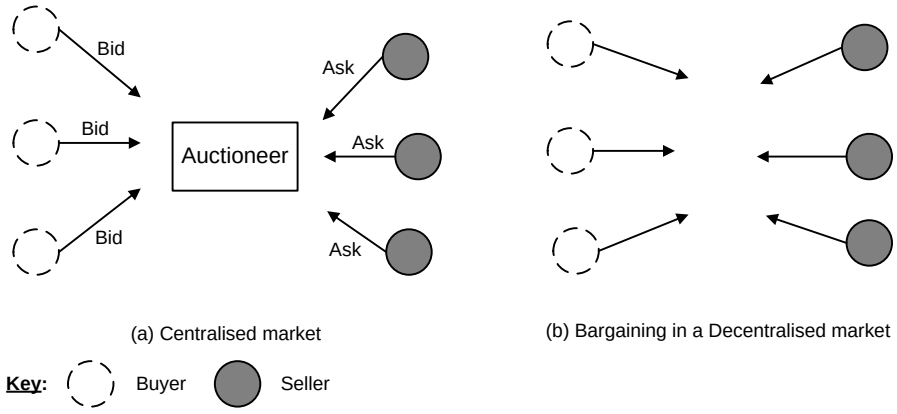


Figure 5.1: Centralised and Decentralised Market Set-up. Source: [71]

Both centralised and decentralised approaches have been used for resource allocation in distributed systems such as grid computing [171] [138]. The application of market-based approaches to cloud differs from grid due to the near instant on-demand request

for cloud services coupled with unanticipated user requirement and duration of service. The market analogy for cloud was proposed by Buyya [26]. Buyya envisioned the cloud as a global market where several customers (cloud users) having various requirements meet sellers (cloud vendors) possessing different capabilities to trade. It was suggested that such trading could be achieved via a cloud exchange in which brokers manage the selection of cloud vendors on behalf of the users.

Following this proposal, researchers have pursued in-depth study of market-oriented cloud from various dimensions, including: price modelling [110], resource sharing among service providers [76, 152, 147], and resource allocation at the hardware layer [177]. These results mostly focus on the use of market mechanisms for selection of clouds with specified QoS or price for service composition at the SaaS layer.

Other researchers have focused on cloud services at the IaaS layer [6] [150] [154]. The work of [6] provided a game theoretic formulation of the service provisioning problem in cloud systems. In [150], the problem of running independent equal-sized tasks on a cloud infrastructure with a limited budget was studied. They concluded that a constrained computing resource allocation scheme should be benefit-aware, i.e., the heuristics for task allocation should incorporate the limited resource in supply within the system. Sun et al. [154] proposed a Nash Equilibrium based Continuous Double Auction (NECDA) cloud resource allocation algorithm. They used the continuous double auction and a Nash equilibrium solution concept to allocate resources in an *M/M/1* queuing system, with the objectives of meeting performance QoS.

The relation between clouds and markets, as depicted in figure 5.2, makes it possible to apply concepts from markets to computational resource allocation problems in the cloud. The utility derived from consumption of cloud resource determines the buyer's valuation of the resource and the price they are willing to pay for it. The notion of variability in markets, in terms of seller's capacity etc., can be used to model trends, such as heterogeneity and unpredictability of resource nodes in cloud software architectures.

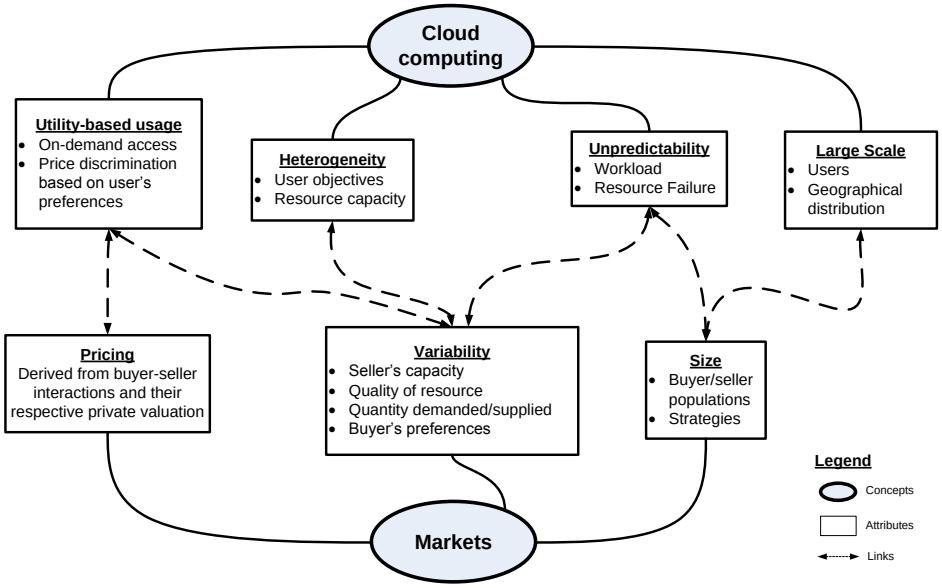


Figure 5.2: Conceptual Relationship Between Cloud Computing and Markets. Source: [71]

5.3.2 Reputation Management and Markets

Another dimension for distinguishing between resource allocation in cloud architectures is to consider whether they are state-based or stateless, when allocating resources at runtime. Stateless approaches make allocation decisions based on predefined rules without consideration of the extent to which resource nodes are able to meet the expected QoS. In contrast, state-based approaches take into account QoS concerns before allocating resources.

One state-based approach for managing the behaviour of resource nodes in a dynamic environment is the use of reputation metrics. This is especially true in multi-agent systems where the concept has been well-researched and applied to problems of trust management in ad-hoc mobile networks [21] and peer-to-peer computing [135]. In these domains, the system designer defines what constitutes *acceptable* and *unacceptable* behaviour. Consequently, an agent's reputation rating (or trust value) increases when it acts in an acceptable manner and decreases when it acts otherwise. Modern recommender systems utilise this concept to elicit shopping trends, subsequently dispatching targeted marketing ma-

terials that best reflect the user's preferences.

Market designers leverage on the notion of reputation measurements to elicit the trustworthiness or dependability of market agents to ensure efficient resource allocation [55] [137]. An agent's reputation could then be used to make decisions about whether it is more likely to meet the service level terms of jobs allocated to it.

5.4 Posted Offer Market Mechanism

This section presents the ideas and assumptions about the posted offer market mechanism.

Aside from the market mechanisms mentioned in section 5.3, another viable market mechanism is the retail-inspired posted offer or posted price market mechanism originally proposed by Fred E. William [170]. It was devised as a simple way of investigating the impact of auctions in a distributed market setting. The two scenarios considered in his experiment are (i) buyers making price offers and sellers responded only with quantity offers, and (ii) sellers posting their quantity offers and buyers responded only with price bids. Even though the notion of *price leadership* (i.e. whether buyer or seller made the first price offer or quantity offer respectively) was focal to his investigation; the distinction between the scenarios are often ignored in actual implementation.

In William's investigation, the assumptions underlying the posted offer mechanism include (i) both buyer and seller information (i.e. price offer and quantity offer respectively) are private, (ii) traded product are homogeneous across all sellers, (iii) buyers and sellers could not change their price offer and quantity offer respectively once this is decided at the beginning of the trading period, and (iv) seller(s) are free to sell to any buyer(s) and buyer(s) are free to buy from any seller(s).

Each seller is given a production cost schedule showing the quantity s/he can produce and the cost of production per quantity. Similarly, each buyer is given a revenue schedule showing the quantity s/he can purchase and revenue per quantity. A fixed cost is

charged each time a seller or buyer participates in a trading period. Section 5.5 details the algorithms that control buyer and seller agents' behaviour in the market.

5.4.1 Previous Extensions to the Mechanism

In comparison to a central auction, the bidding process of the posted offer model is known to be less cumbersome and more reflective of resource provisioning scenarios in distributed computer networks [107]. Indeed, the posted offer mechanism has been adopted and extended for managing resources in computational systems under various assumptions [107][37][173][114]. Such extensions were proposed towards solving problems in domains such as communication networks [107], large-scale service-oriented systems [114] and computational grids [37][173]. Table 5.1 summarises these contributions.

Author	**Market objective**	**Assumption(s)**	**Extension**
[107]	Investigated the impact of self-interested agents on global market objective	Agents were self-interested and utility maximising with local view of market	Proposed a technique that reduced the oscillations caused by delays in a communication network
[37]	Minimise mean flow time per job (i.e. job turnaround time)	Small-size network of active nodes (2-10 machines)	Heuristic-based learning for decision-making and detection of inaccurate job estimates
[173]	Improve market efficiency in a decentralised P2P computational grid	Network and hardware failure were not accounted for	Simplistic price adjustment and penalty schemes
[114]	Fully decentralised load balancing in large-scale service-oriented systems	Resource nodes were reliable and network connectivity was uniform	Adaptive pricing by evolutionary market agents
Proposed in this thesis	Improve SLA compliance in cloud-based applications	Resource nodes and network connectivity are unreliable	Reputation metrics are incorporated to identify dependable resource nodes

Table 5.1: Variants of Posted Offer Market Mechanism

5.5 Design of Reputation-aware Posted Offer Market

Our reputation-aware posted offer market mechanism is presented in this section.

Consider a marketplace consisting of a set of buyers and sellers trading in a cloud federation. Buyers and sellers have different utility functions representing their valuation of resources to be sold or bought respectively. Buyers are service seeking nodes, acting on behalf of application owners and willing to exchange currency to meet application SLAs. Sellers are service providing nodes, acting on behalf of cloud providers capable of offering service for payment. Moreover, there are multiple types of services traded in the market.

The workload pattern of service requests or jobs is defined by some random distribution. A service request is defined by a set of multiple attributes which specifies its SLA terms. Trading between buyers and sellers takes place according to rules specified by the market mechanism in place at trading rounds. A trading round denotes a time instance when a buyer agent can select a candidate cloud service (seller). The market objective is to facilitate rapid matching of buyers and sellers to maximise allocative efficiency.

In reality sellers (cloud providers) are not fully reliable neither are network connections linking buyers to sellers, hence, the mechanism driving the interaction of traders needs to account for these to maximise allocative efficiency. It is important to note that the reliability distribution of sellers is not known to buyers beforehand, hence, this information needs to be learnt. Next, we present our approach for selecting reliable cloud providers.

5.5.1 Refinement of Classic Posted Offer Mechanism

First we state a number of assumptions that distinguish the classic posted offer market (where humans are traders) from our variant which is deployed in computing systems. Our assumptions are consistent with practices in market-based control [47][50].

1. Seller agents are resource providing nodes in a computer network and buyer agents are service agents representing users of these resources.

2. Each buyer agent is responsible for a job in each trading round, for which it makes bids to sellers who may offer to execute the job.

3. Each seller agent has a reputation associated with it. Seller agents report their reputation rating when asking for jobs.

4. Currency in the market is not 'real currency', rather 'artificial money' are exchanged between buyers and sellers. Thus, 'artificial money' is used only as a control tool.

5. Trading happens concurrently across multiple buyer-seller pairs and continues as long as at least an active buyer and seller exists in the market.

Buyer to Seller Trading

Computational systems which are controlled by market algorithms fundamentally rely on "price" as the tool for understanding the interaction among market agents, and it aids the system design in a way that ensures some desirable global properties are realised. However, price alone tells us little about whether agents are reliable or not. To elicit the reliability of agents, we model their behaviour using price and reputation metrics which capture their trading capabilities and performance over time. Measuring both dimensions gives a holistic view to improve the efficiency of resource allocation.

We refine the generic posted offer mechanism by introducing the notion of *reputable market agents* to provide a metric for measuring the reliability of sellers at executing service requests allocated to them. Due to the dynamics of the environment, cloud providers vary in their capability and reliability measures. Therefore, the refined steps of the mechanism consists of collecting performance data about cloud providers, and making use of this knowledge to make decision about resource allocation to cloud providers.

Losing and Gaining Reputation

After each transaction sellers are rated based on their performance at completing the requested service. Buyers rate sellers by comparing the actual performance with those

specified in the SLA associated to the service request. Using this distributed reputation rating approach, sellers gain reputation ratings for service requests completed according to SLAs and lose reputation ratings for those that violate SLAs.

Figure 5.3 depicts a sequence diagram showing interaction of buyer and seller agents in our refined posted offer market mechanism.

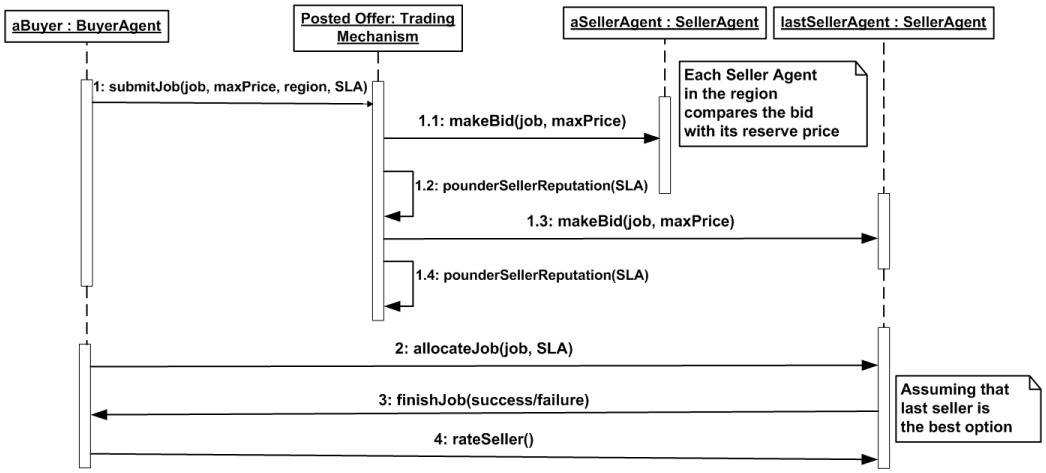


Figure 5.3: Reputation-aware Posted Offer Market Mechanism

In a cloud deployment context, buyers and sellers in our refined mechanism are mapped to cloud entities as shown in figure 5.4.

Notations

The variables used in our mechanism formulation and algorithms are defined in table 5.2.

Seller Agent Resilience Model

By resilience, we mean "The persistence of service delivery that can justifiably be trusted, when facing changes" [109]. Resilience is measured by the ability of the mechanism to filter out resource nodes that are unable to meet a job's SLA. In order to introduce faulty behaviour into the system, we define the *Failure Rate* of a resource node (seller agent) as the instantaneous probability of the node failing on demand. Therefore, a seller agent, S, has a failure rate in the bound $[0, 0.1]$, where a value of 0 signifies maximum

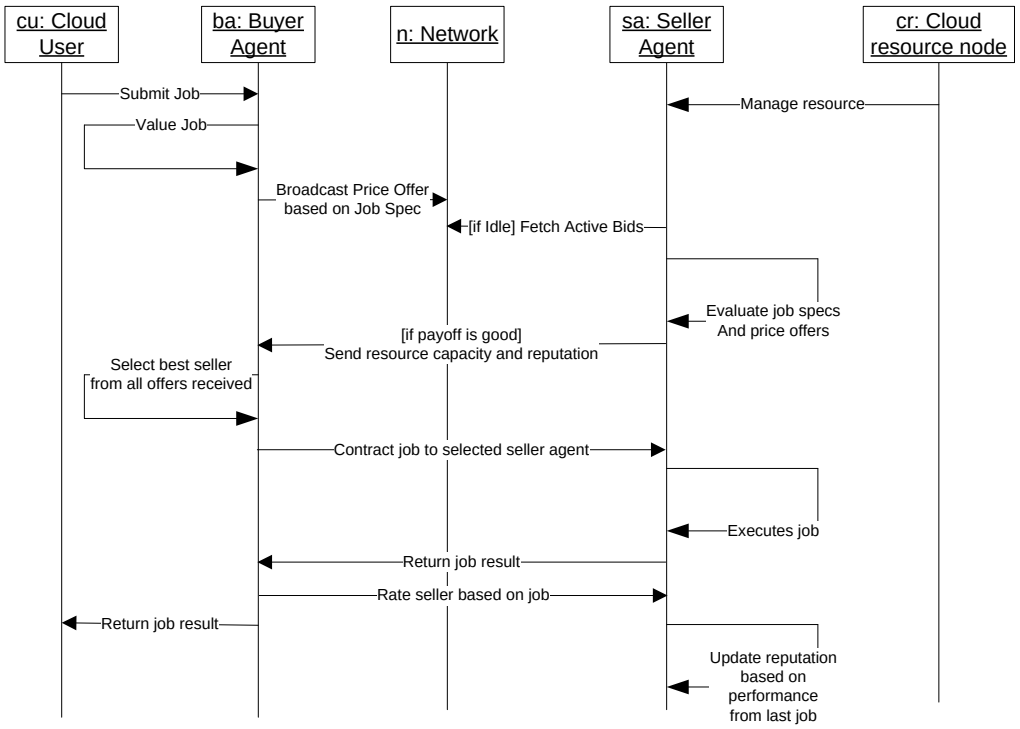


Figure 5.4: Reputation-aware Posted Offer Mechanism in Cloud System

reliable behaviour, and a value of 0.1 denotes maximum unreliability. Two sub-intervals of the bound are compared (see figure 5.5) across three scalability scenarios as shown in table 5.3. High resilience indicate trustworthy service delivery and the opposite holds for low resilience.

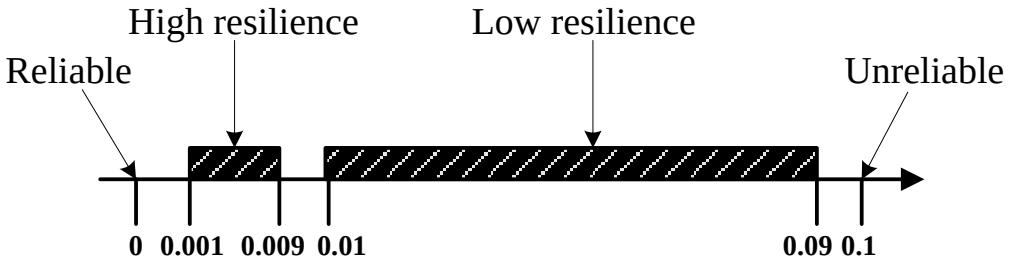


Figure 5.5: Failure Rate Interval

Service Level Agreement (SLA) and Reputation Models

To ease analysis, we define an SLA model consisting of four non-functional (NF) attributes: availability, performance, security, and SLA priority. SLA priority, in particular,

Variable	**Definition**
WorkloadDataset	A synthetic/real workload data representing the distribution of jobs over trading rounds
NumOfTradingRnd	Number of trading round in simulation run
ResilienceLevel	The resilience level (high or low) of seller agents in the market
jb	A job or a request for service
jb_{SLA}	A tuple (Availability, Security, Performance, Priority) that defines a job's SLA based on buyer agent values in table 5.4
N	Number of jobs in a trading round
M	Number of seller agents in market
ListOfJobs	A collection of jobs $\{jb_1, jb_2, .., jb_N\}$
B	A buyer agent
ListOfBuyerAgents	A collection of buyer agents $\{B_1, B_2, .., B_N\}$
S	A seller Agent
S^{Rep}	Seller agent's reputation rating
ListOfSellerAgents	A collection of seller agents $\{S_1, S_2, .., S_M\}$
ListOfInterestedSellerAgents	An arbitrary collection of tuples (S, S^{Rep})
ListOfEligibleSellerAgents	An arbitrary collection of seller agents
T^{Rep}	Market threshold reputation i.e. mean of S^{Rep} for all seller agents

Table 5.2: Definition of Variables

captures the cloud user's *valuation* of the job. The idea is to use the SLA model as a descriptor for a job's computational requirement.

Table 5.4 captures the metrics for measuring each non-functional attribute and the range of permissible values that buyer and seller agent's can assume for each attribute. These attribute-values are randomly initialised for each job submitted to the system following a normal distribution. As it can be observed from the table, we have deliberately set higher values for seller agents for all attribute-values. This is because, the objective of the study is primarily to evaluate the self-aware architecture's scalability and its resilience to failure, rather than the intricate details of buyer to seller matching process.

The two key ingredients of the posted offer mechanism (see figure 5.3) are the reputa-

Scenario	**Failure Rate**	# B	# S
Low resilience (case A)	[0.01 - 0.09]	50	50
Low resilience (case B)	[0.01 - 0.09]	200	200
Low resilience (case C)	[0.01 - 0.09]	500	50
High resilience (case A)	[0.001 - 0.009]	50	50
High resilience (case B)	[0.001 - 0.009]	200	200
High resilience (case C)	[0.001 - 0.009]	500	50

Table 5.3: Resource Node Failure Scenarios

Attribute	**Metric**	B **Values**	S **Values**
Availability	Uptime (%)	90 - 99.9999	100
Performance	# instructions per second	80 - 99.9999	100
Security	Encryption Support	Yes (1) or No (0)	Yes (1)
Priority	Low, Medium, High	Random	-

Table 5.4: SLA Model: Attribute-Metric-Value for Buyer and Seller Agents

tion model used to assess the seller agents, and buyers and sellers utility functions. The models used to capture these properties of the mechanism are presented below.

Reputation Model: the reputation rating of a seller agent, S, is defined by an historic model as follows:

$$S^{Rep} = \frac{Total_{Request} - Failed_{Request}}{Total_{Request}} * 100\% \tag{5.1}$$

where $Total_{Request}$ and $Failed_{Request}$ are the total number of requests and total number of failed requests executed by the seller respectively.

In every trading round, buyer make use of the threshold reputation rating, T^{Rep}, which is the mean of all S^{Rep} in the market, that is

$$T^{Rep} = \frac{\sum_{i=1}^{M} S_i^{Rep}}{M} \tag{5.2}$$

Buyer agents (irrespective of the strategy in use), consider only sellers with $S^{Rep} >= T^{Rep}$ in that trading round. The only exception being the first trading round, since the

value of T^{Rep} cannot be computed at this stage - nothing is known about the reputation of seller agents at first interaction.

Utility Functions: given an SLA, the buyer utility function is defined as

$$U_b(jb) = w_b + (k * \beta_{price}) \tag{5.3}$$

The value of w_b is initialised based on the job's priority. For results presented here, w_b is empirically set to:

$$w_b = \begin{cases} 2, & \text{if SLA Priority is High.} \\ 1, & \text{if SLA Priority is Medium.} \\ 0, & \text{otherwise.} \end{cases} \tag{5.4}$$

SLA priorities are randomly assigned to jobs, following a normal distribution. k is a sensitivity factor for tuning the valuation of the buyer agent. $k = 0.1$ for all scenarios considered here. The value of β_{price} is derived from summation of NF attributes of the buyer agent. That is $\beta_{price} = \sum_{i=1}^{4} B_{Attribute_i}$, where, $Attribute_i$ refer to attributes in table 5.4.

Given a job request, the seller utility is defined as

$$U_s(jb) = w_s * \theta_{price} \tag{5.5}$$

For results presented here, w_s is empirically set to:

$$w_s = \begin{cases} 0.1, & \text{if SLA Priority is High.} \\ 0.01, & \text{if SLA Priority is Medium.} \\ 0.001, & \text{otherwise.} \end{cases} \tag{5.6}$$

Similar to β_{price}, the value of θ_{price} is derived from summation of NF attributes of the seller agent. That is $\theta_{price} = \sum_{i=1}^{4} S_{Attribute_i}$, where, $Attribute_i$ refer to attributes in

table 5.4.

It is worth noting that this formulation ensures that when trading solely based on price, it is always possible to find a seller agent for a job request. However, there is no assurance that the chosen seller will successfully execute the job. It is this problem that our refinement to the posted offer mechanism addresses by preferring reputable sellers to non-reputable ones, making it adapt to fluctuating seller resilience levels.

5.5.2 Trading Strategies and Procedures

Trading Strategies

A market agent's trading strategy determines how the agent makes trading decisions. Either buyer or seller agents can be equipped with trading strategies, ranging from very simple ones to complex strategies. Two buyer strategies, originally conceived in [116], are considered here to illustrate the impact of the applied strategy on the outcome allocation.

- *Time Savers*: purchase from any seller chosen at random, provided the price is acceptable, i.e., selling price less or equal buyer's valuation.
- *Bargain Hunters*: are always in search of the seller with the best possible price. That is, the selling price must be the lowest among available sellers. If more than one seller offers the lowest price, then one is chosen at random.

Crucially, the significance of the impact of these strategies can be appreciated only when their benefits, in terms of number of failure recorded, and overhead are compared. Analysing the strategies in isolation without having this big picture in mind provides little insight for understanding the trade-off between their benefits and overhead.

Reputation-aware Mechanism Procedures

The variables used in the algorithms presented in this section are defined in table 5.2. The market coordinates interaction of buyers and sellers using Algorithm 1, while Algorithm 2

is used by buyer agents for selecting seller agents. In Algorithm 1: steps 1-3 initialises the algorithm, steps 5-10 initialise each trading round, buyers concurrently select sellers for jobs assigned to them in steps 12-22. It is important to note that T^{Rep} is dynamically updated as trading occurs, since it is an aggregate of seller reputation.

Algorithm 1 Market Coordination Procedure (Reputation-aware)

Input: WorkloadDataset, NumOfTradingRnd, ResilienceLevel

1: Set $n :=$ NumOfTradingRnd;
2: Randomly initialise SLA values for jobs in all trading rounds;
3: Randomly initialise ListOfSellerAgents based on ResilienceLevel;
4: **for** $i = 1$ to n **do**
5: Set $N :=$ Number of jobs in WorkloadDataSet for trading round i;
6: Initialise ListOfJobs for N jobs;
7: Initialise ListOfBuyerAgents for N jobs;
8: **for** each job jb in ListOfJobs **do**
9: Randomly assign jb to a buyer in ListOfBuyerAgents: $B(jb) \leftarrow jb$;
10: **end for**
11: Compute threshold reputation, T^{Rep}, for trading round i using Eqn 5.2;
12: **for** each buyer B in ListOfBuyerAgents **do**
13: B privately computes utility, $U_b(jb)$, for its job using Eqn 5.3;
14: B broadcasts tuple (B, jb) to sellers;
15: **for** each seller S in ListOfSellerAgents **do**
16: S privately computes utility, $U_s(jb)$, for all job requests using Eqn 5.5;
17: **if** $(U_s(jb) > 0)$ **then**
18: Compute reputation rating, S^{Rep}, of S using Eqn 5.1;
19: Add tuple (S, S^{Rep}) to ListOfInterestedSellerAgents;
20: **end if**
21: **end for**
22: Buyer selects a seller from ListOfInterestedSellerAgents using Algorithm 2;
23: **end for**
24: **end for**

In step 1 of Algorithm 2, only one strategy is used by all buyers per run (in simulation) of the algorithm. That is, strategies are homogeneous for buyers for each simulation run.

Algorithm 2 Buyer Decision-making Procedure (Reputation-aware)

Input: ListOfInterestedSellerAgents, jb, T^{Rep}

Output: *JobStatus*

1: Set St := Strategy *BargainHunter* or *TimeSaver* defined in section 5.5.2;
2: **for** each seller S in ListOfInterestedSellerAgents **do**
3: **if** ($S^{Rep} >= T^{Rep}$) **then**
4: Add S to ListOfEligibleSellerAgents;
5: **end if**
6: **end for**
7: **if** ListOfEligibleSellerAgents is empty **then**
8: Resubmit job, jb, in next trading round;
9: **else**
10: Pick S from ListOfEligibleSellerAgents using strategy St;
11: Dispatch job, jb, to seller S;
12: Set *JobStatus* := Success or Fail depending on performance of seller, S;
13: **if** (*JobStatus* is Success) **then**
14: Seller, S, updates reputation, S^{Rep}, using Eqn 5.1;
15: **else**
16: Seller, S, increments failed requests and updates reputation, S^{Rep}, using Eqn 5.1;
17: **end if**
18: **end if**

Classic Non-Reputation-aware Posted Offer Mechanism Procedures

The variables used in the algorithms presented in this section are defined in table 5.2. Algorithms for the classic non-reputation-aware posted offer mechanism [107] are presented in Algorithm 3 (market coordination) and Algorithm 4 (buyer decision-making). Trading is based only on the notion of price; reputation of sellers is not considered.

Algorithm 3 Market Coordination Procedure (Non-Reputation-aware)

Input: WorkloadDataset, NumOfTradingRnd, ResilienceLevel

Set $n :=$ NumOfTradingRnd;
Randomly initialise SLA values for jobs in all trading rounds;
Randomly initialise ListOfSellerAgents based on ResilienceLevel;
for $i = 1$ to n **do**
    Set $N :=$ Number of jobs in WorkloadDataSet for trading round i;
    Initialise ListOfJobs for N jobs;
    Initialise ListOfBuyerAgents for N jobs;
    for each job jb in ListOfJobs **do**
        Randomly assign jb to a buyer in ListOfBuyerAgents: $B(jb) \leftarrow jb$;
    end for
    for each buyer B in ListOfBuyerAgents **do**
        B privately computes utility, $U_b(jb)$, for its job using Eqn 5.3;
        B broadcasts tuple (B, jb) to sellers;
        for each seller S in ListOfSellerAgents **do**
            S privately computes utility, $U_s(jb)$, for all job requests using Eqn 5.5;
            if $(U_s(jb) > 0)$ **then**
                Add tuple S to ListOfEligibleSellerAgents;
            end if
        end for
        Buyer selects a seller from ListOfEligibleSellerAgents using Algorithm 4;
    end for
end for

Algorithm 4 Buyer Decision-making Procedure (Non-Reputation-aware)

Input: ListOfEligibleSellerAgents, *jb*

Output: *JobStatus*

1: Set *St* := Strategy *BargainHunter* or *TimeSaver* defined in section 5.5.2;
2: **if** ListOfEligibleSellerAgents is empty **then**
3: Resubmit job, *jb*, in next trading round;
4: **else**
5: Pick *S* from ListOfEligibleSellerAgents using strategy *St*;
6: Dispatch job, *jb*, to seller *S*;
7: Set *JobStatus* := Success or Fail depending on performance of seller, *S*;
8: **end if**

5.6 Instantiation of Self-aware Cloud Architecture

A self-aware architecture that realises the requirements of the Online shopping cloud application (c.f.Chapter 1, section 1.5) is presented in this section.

Table 5.5: Rationale for Selecting Architecture Pattern

Pattern	**Suitable?**	**Rationale**
Basic Information Sharing	No	Caters for only interacting nodes, without modelling changing goals, time concern, and sharing of self-expressive decisions
Coordinated Decision-making	No	Caters for interacting nodes, without modelling changing goals and time concerns
Temporal Knowledge Sharing	No	Caters for interacting nodes and temporal knowledge sharing, without modelling changing goals
Goal Disseminating	Yes	Caters for interacting nodes, changing goals, and temporal knowledge sharing

In the self-aware approach, the underlying software architecture of the adaptation subsystem should cater for fine-grained representation of knowledge pertaining to changing goals, workload, and service availability. The self-aware style, presented in chapter 4,

offers primitives for modelling knowledge using a multi-level representation approach that simplifies run-time trade-off analyses.

Given the choice of five self-aware architecture patterns (see chapter 4) from which an architecture instance could be derived, we revisit our pattern catalogues to assess the suitability of each pattern at meeting the requirements of the problem at hand. As shown in table 5.5, the Goal Disseminating Pattern is most representative for the Online shopping cloud application. The responsibility of each level of awareness in the goal disseminating pattern within the context of the Online shopping cloud application is described in table 5.6. The architecture artefacts that realise different subcomponents of the self-aware cloud architecture are described in table 5.7.

Table 5.6: Responsibility of levels of awareness in SLA-based Cloud Architecture

Level of awareness	**Dynamics to manage**	**Constraints**
Stimulus-awareness	Sensing changes to workload and responding to user requests	Detection of workload changes, add/remove services
Goal-awareness	Changing user goals as SLAs vary from one user to another	Addition of new SLA, modification of existing SLA
Interaction-awareness	Communicating with cloud services and adaptation subsystems belonging to other service-based applications (SBAs)	Network connectivity
Time-awareness	Sensitivity to temporal changes in workload and cloud service availability	User arrival rate, number of services

Using the reputation-aware posted offer as adaptation mechanism, the self-aware architecture of the adaptation subsystem is shown in figure 5.6. The design goal of the system is to meet the SLA compliance of cloud users. The run-time goal, which changes per user request, captures the SLA goal of the request currently being managed by the adaptation subsystem. For example, in the context of the online shopping application, each order introduces a new SLA goal, which is defined by the delivery time and cost constraints.

The adaptation subsystem is deployed in an environment where it senses cloud ser-

Table 5.7: Architecture Artefact that realise Subcomponents Self-aware cloud architecture

Self-aware subcomponent	**Architectural artefact**
Stimulus-awareness	Workload dataset captures spikes and dwindles in user requests and their SLA classes
Goal-awareness	An utility function is used to deduce candidate service's likelihood of meeting service levels
Interaction-awareness	A locally performance repository stores availability information about cloud services and provide service connection information to facilitate interaction
Time-awareness	A locally managed performance repository stores reputation rating of services, in terms of the level to which the service met promised QoS, and the duration of their use
Self-expression	Locally managed strategies are encoded in each self-aware subsystem to guide the process of service selection

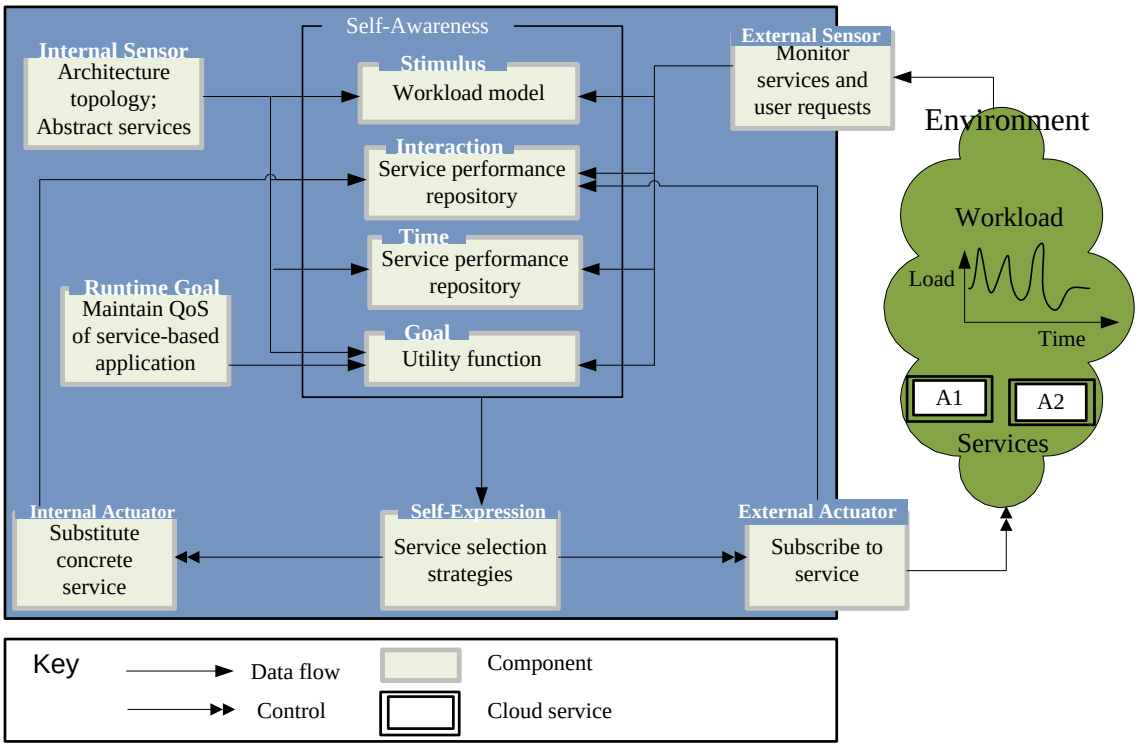


Figure 5.6: Self-aware Cloud Architecture of an SBA's Adaptation Subsystem

vices and the workload generated by user requests. The external sensor and actuator components serve as monitors and hooks for adapting cloud services. Typically, external monitors and actuators are cloud APIs which can be dynamically adapted at run-time.

The internal sensor is an architectural model of the application topology and characteristics of its components which need to be respected when instantiating cloud services. In this architecture, the internal actuator carries out service substitution at run-time.

5.7 Empirical Study of Self-aware Cloud Architecture

This section studies the self-aware cloud architecture by comparing the outcome SLA compliance under two coordination mechanism: (i) reputation-aware posted offer mechanism (ii) non-reputation-aware posted offer mechanism. The study is conducted using synthetic workload (section 5.7.3) and realistic Google Cluster workload [56] (section 5.7.4).

5.7.1 Setup and Justification for Experimental Approach

Simulation-based approach is used for our experimental study. While we acknowledge that there is much to be learnt from a real deployment of our self-aware cloud architecture (see section 5.8), we adopt a simulation approach to evaluate the architecture for the following reasons. Firstly, it is well-known in the cloud community [25] that simulations are preferred for studying properties of novel computational mechanisms as they afford the opportunity to repeat experiments in a quick and inexpensive controlled environment [12]. Secondly, extreme scenarios and edge cases which are hard to replicate in real clouds may be studied using simulations, thereby improving the robustness of the proffered solution. Thirdly, experimentation by simulation reduces the effect of the interference problem which results due to co-located cloud services starving other services in the shared infrastructure, thereby causing an experiment to result to different outcomes depending on how much interference is present at the times the experiments are carried out.

The CloudSim simulation toolkit [25] was used for evaluation purposes. CloudSim is developed using Java programming language, it abstracts the cloud infrastructure (virtual machines, hosts, network, etc.) using computational models that are extensible, and it

provides a way of experimenting with workload datasets (synthetic and real). In our work, we have extended the broker and scheduling classes to implement the functionality of our reputation-aware market mechanism. The configuration of the experimental platform is a 6GB RAM, 2.40GHz, 64-bits Windows 7 machine. For all cases, results are averaged over 10 independent simulation runs to account for stochasticity.

5.7.2 Objective of the Study

In line with the objective of the thesis, we investigate the following aspects of the self-aware cloud architecture.

1. Sensitivity of the architecture to high and low resilient cloud services when operating under two self-expressive trading strategies (bargain hunter and time savers).
2. Effect of scale on the architecture.
3. Overhead incurred by the architecture's adaptation mechanism in terms of how long it takes to find suitable cloud services in a trading round.

We differentiate buyers agents in our results using the notation below.

- RBH - Bargain Hunter strategy in reputation-aware mechanism
- RTS - Time Saver strategy in reputation-aware mechanism
- NRBH - Bargain Hunter strategy in non-reputation-aware mechanism
- NRTS - Time Saver strategy in non-reputation-aware mechanism

5.7.3 Study under Synthetic Workload

Since buyer agents manage jobs on behalf of cloud users, we model scalability of jobs by increasing the number of buyer agents in the simulation. The overhead of each strategy is measured by the number of seller agents inspected before a trading decision is made.

Experimental Setting

In this study, number of sellers, M, and number of buyers, N, are to $\#S$ and $\#B$ respectively as defined in table 5.3 for case A, B, and C under each resilience mode, ResilienceMode, (low or high). Number of trading round, NumTradingRnd, is set to 10000. WorkloadData, is a synthetic workload defined by generation of job, jb, for each buyer agent, B, at every 10th time step. SLA for each job, jb_{SLA}, is randomly initialised using a normal distribution based on values defined in table 5.4. Each seller, S, has its resource provisioning capacity defined by values defined in table 5.4.

Results

The results for low and high resilience cases under synthetic workload are shown in figure 5.7 and 5.8.

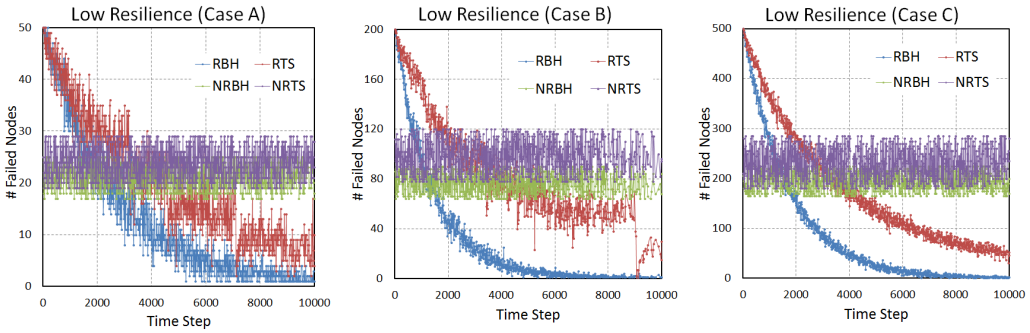


Figure 5.7: Low Resilience Cases - Sensitivity to Failure

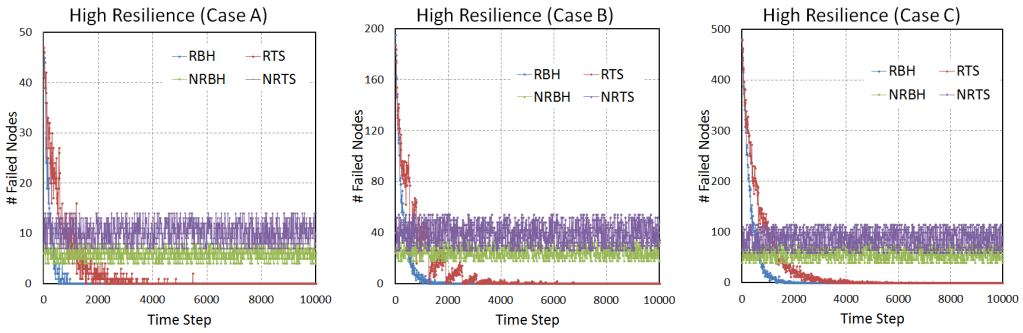


Figure 5.8: High Resilience Cases - Sensitivity to Failure

Next, we interpret the results in terms of failure rate, scalability, and overhead incurred by reputation-aware and non-reputation-aware posted offer mechanisms.

Effect of Reputation on Failure Rate

In low resilience cases (figure 5.7) NRBH and NRTS performed better than RBH and RTS initially (approximately first 1000 time steps). The percentage success of NRBH and NRTS was in the region of 70-50% and 60-40% respectively. RBH and RTS were more successful than NRBH and NRTS after the first 1000 time steps. Both RBH and RTS performed better overall and eventually converged towards zero failed nodes. RBH and RTS have percentage success in the region 90-70% and 70-60% respectively.

In high resilience cases (figure 5.8), the success rate of NRBH and NRTS are in the region of 90-75% and 85-65% respectively. NRBH and NRTS record higher success percentage than RBH and RTS in the first (approximately) 500 time steps. For the rest of the simulation, RBH and RTS outperformed both NRBH and NRTS, having percentage success in the region of 99-90% and 95-85% respectively.

The initial better performance of NRBH and NRTS over RBH and RTS may be attributed to the time taken for RBH and RTS to build up their reputation repository to reflect seller agents' resilience. Overall the learning capability of RBH and RTS accounts for their higher SLA compliance over NRBH and NRTS. In contrast, inability of NRBH and NRTS to learn may explain why they do not significantly improve their success rate throughout the simulation.

Scalability of Reputation-aware and Non-Reputation-aware Mechanisms

In both low and high resilience cases, results indicate that RBH, RTS, NRBH, and NRTS scale as the number of jobs increase from 50 to 500. When compared to NRBH and NRTS, the RBH and RTS were able to scale more gracefully given the shape of the curves. In low resilience cases, the results (figure 5.7) indicate more instability (noise) in the behaviour of all four strategies when compared to high resilience cases (figure 5.8). Additionally under

moderate workload (case A and B), where buyers and sellers are of equal population sizes, RBH outperformed RTS, NRBH, and NRTS.

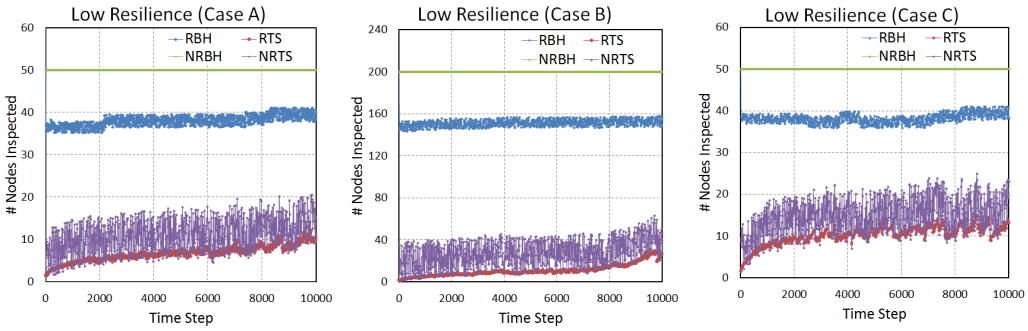


Figure 5.9: Low Resilience Scenarios - Overhead of finding seller agent

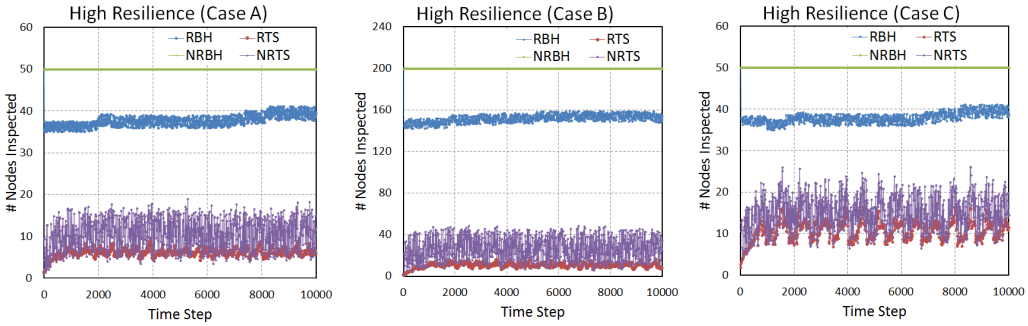


Figure 5.10: High Resilience Scenarios - Overhead of finding seller agent

Overhead of Reputation-aware and Non-Reputation-aware Mechanisms

From figures 5.9 and 5.10, it can be observed that in both lower and higher resilience cases RBH and NRBH incurred higher overhead than RTS and NRTS. This is because NRBH inspected all sellers in each case before selecting a seller based on price. RBH inspected sellers that had $S^{Rep} >= T^{Rep}$ before selecting a seller therefore reducing the set of sellers available to RBH when compared to NRBH. NRTS selected sellers irrespective of their reputation rating hence inspected more sellers than RTS which selected only sellers with $S^{Rep} >= T^{Rep}$.

Figure 5.11 summaries the average SLA compliance, i.e. percentage success, for all cases considered. Overall, we observe a trade-off between the SLA compliance and the

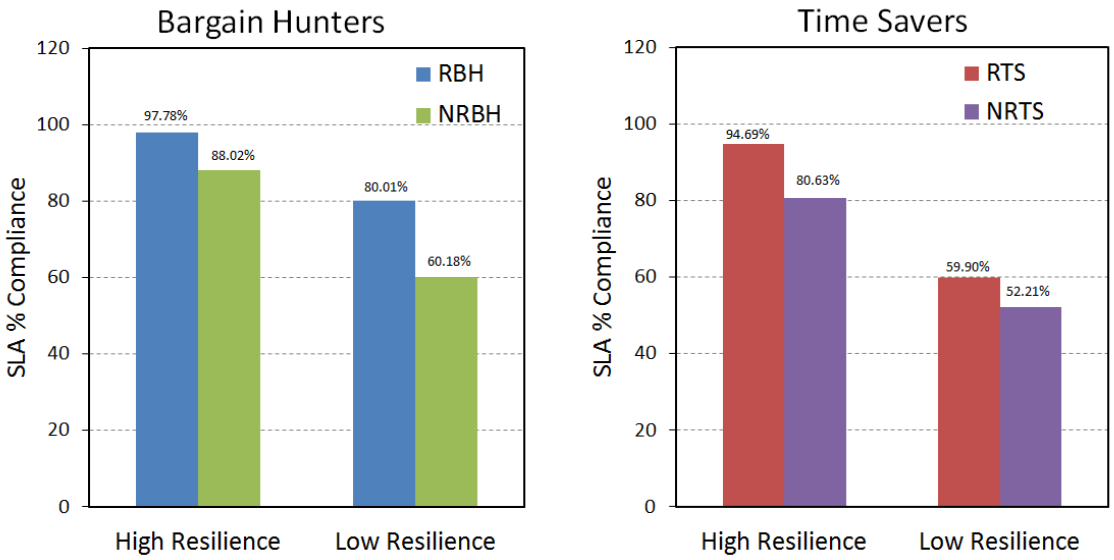


Figure 5.11: SLA Compliance for Bargain Hunters and Time Savers when using Reputation-aware and Non-Reputation-aware Mechanism under synthetic workload

overhead as shown in figure 5.12. In practice, these results can guide the software architecture about how to resolve this trade-off space, i.e., deciding which strategy to adopt for a job given its timeliness constraint and acceptable failure rate.

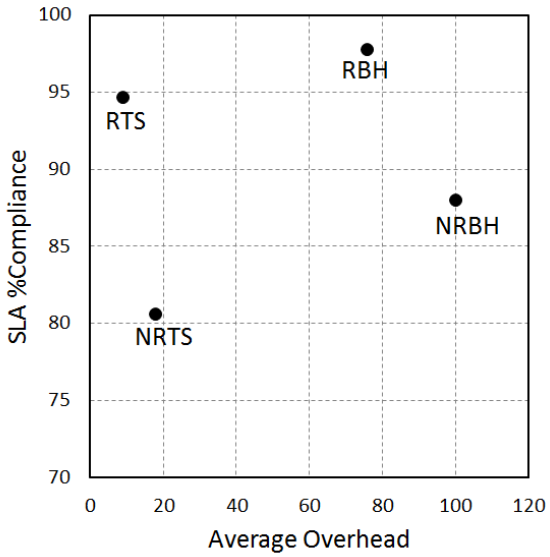


Figure 5.12: Trade-off between SLA Compliance and Average Overhead

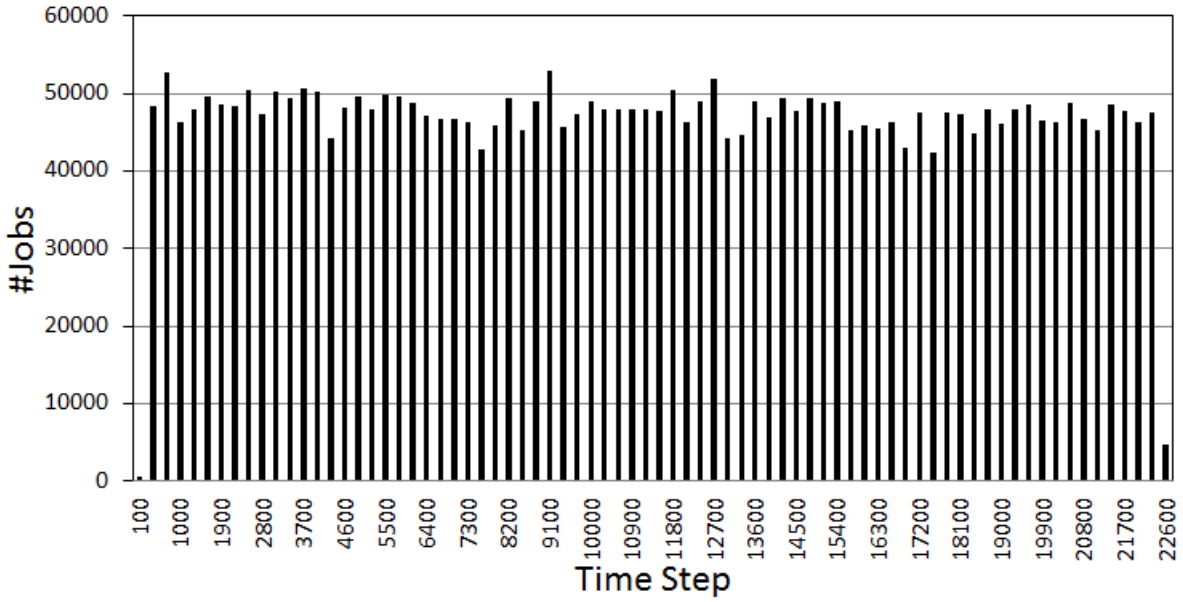


Figure 5.13: Distribution of Jobs in Google Cluster Dataset

5.7.4 Study under Real Workload - Google Cluster Dataset

This section evaluates the reputation-aware and non-reputation-aware posted offer mechanisms using the Google cluster dataset [56]. The Google cluster dataset provides traces over a 7 hour period. Each task in the workload belongs to a single job. Hence, we focus on the allocation of resources at the task-level. The distribution of jobs over the workload period is shown in figure 5.13. Job Type (0, 1, 2) is used as a categorization of work i.e. SLA classes corresponding to Low, Medium, and High respectively.

Experimental Setting

WorkloadData, is a real workload defined by jobs as distributed in figure 5.13. At each time step, N is set to the number of jobs, jb, at that time step in figure 5.13. Each of the N jobs at a time step is assigned to one of the N buyer agents, B. $M = 52800$, to account for the highest workload in the dataset. ResilienceMode is defined for each case (low or high) as defined in figure 5.5. Number of trading round, NumTradingRnd, is set to 76 as defined by WorkloadData (figure 5.13). The priority of each job is defined by its SLA class in the Google Cluster workload dataset. Each seller, S, has its resource provisioning

capacity defined by values defined in table 5.4.

Results

The results for low and high resilience cases under real workload are shown in figure 5.14.

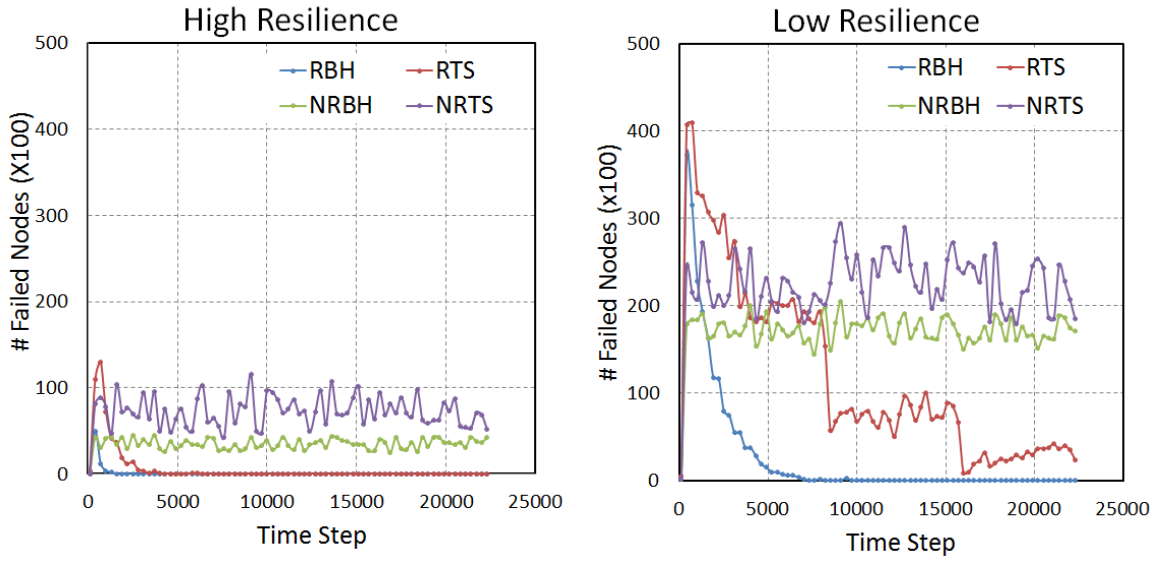


Figure 5.14: Sensitivity to Failure - Google Cluster Dataset

In the high resilience case, RBH performed better than RTS, NRBH, and NRTS, having a percentage success in the region of 100-90%. NRBH and NRTS had percentage success in the region of 95-90% and 85-80% respectively. RTS had percentage success in the region of 100-95% but was worse off than RBH, NRBH, and NRTS in the first (approximately) 300 time steps, however, its success rate improved afterward converging to zero failed nodes. The pattern of failure in the low resilience case is comparable to the high resilience case, however, the behaviour of RTS in the low resilience scenario resembles a step function. That is, consistent small changes in number of failed nodes were recorded at short intervals, followed by a sudden jump (improvement) to much lower number of failed nodes. Figure 5.15 shows the recorded SLA compliance.

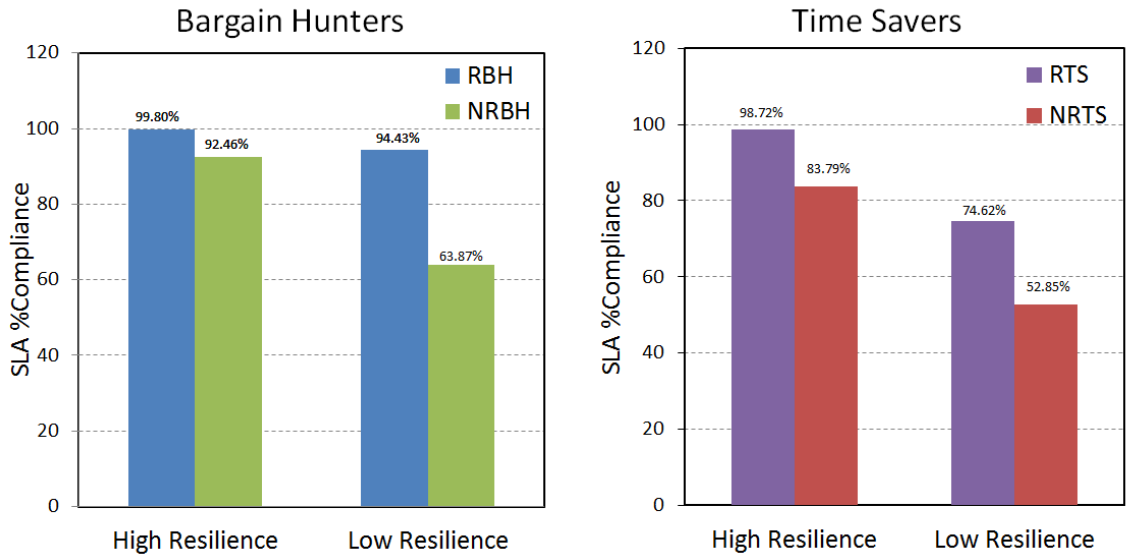


Figure 5.15: SLA Compliance for Bargain Hunter and Time Savers when using Reputation-aware and Non-Reputation-aware Mechanism under real workload

# S	Oscillatory frequency	Transition Time steps
50	1	at 11350
50	5	Every 3783 ticks
50	10	Every 2063 ticks

Table 5.8: Schedule for seller nodes to change their resilience levels. Note: in accordance with the Google Cluster Data, the last time step is 22700, hence, transition time steps are evenly distributed across the simulation life time.

Measuring Impact of Fluctuating Cloud Service Resilience

Up until now, the notion of seller agent resilience is initialised at the start of the simulation run, and fixed throughout the simulation. According to [117], the dynamics of real cloud data centres necessitates an approach that is able to manage transitions across several resilience levels. We define such a transition as the *Oscillatory Frequency* of the node's resilience. This is the number of times a seller node makes a transition from one resilience level to another. We do not differentiate between cyclic transitions at this point, i.e., the case where the seller node returns to its initial resilience level after a number of successive transitions.

The enriched failure model firstly sets out the resilience levels, here we consider only

high and low resilience, and secondly set the number of transitions (oscillations) and the time steps when these transitions will occur. An example of the enriched failure model for a population of sellers is shown in table 5.8.

The seller population in table 5.8 is split evenly between the two resilience levels, therefore, there are 50 seller nodes, of which 25 are high resilient and the other 25 are low resilient. At the transition time step, a seller node changes to the opposite resilience level, i.e., a high resilience seller node changes to low, and vice versa. Figure 5.16 shows the impact of the oscillatory frequencies 1, 5, and 10 on the number of failed nodes recorded.

Results

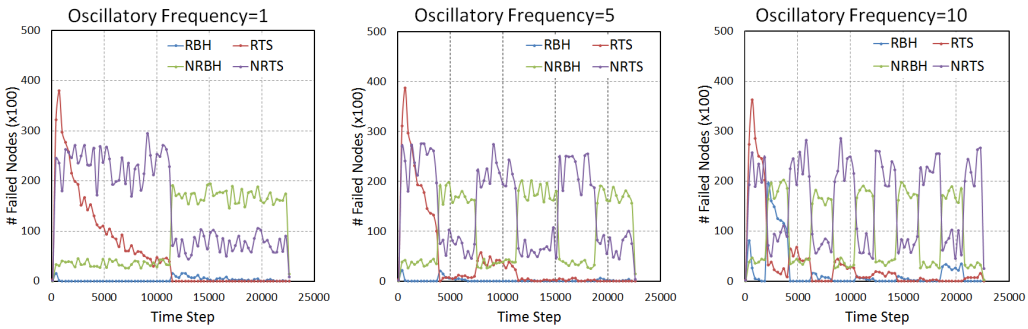


Figure 5.16: Sensitivity to Failure for Oscillatory Frequencies 1, 5, and 10

In the case of one oscillatory frequency RBH recorded significantly smaller number of failed nodes when compared to RTS prior to the transition time step (time tick 11350). However, after the transition time step, the behaviour of RBH and RTS are comparable, as they both record less than 20 failures. NRBH record less than 100 failures in the prior to the transition time step but failure increased afterwards, peaking at approximately 200 failures. NRTS recorded the most failure prior to the transition but significantly improved after the transition, peaking at approximately 100 failures.

In the case of five oscillatory frequency case, RBH and RTS behaved in a way similar to the one oscillatory frequency case up to the first transition time step (time tick 3783). After the first transition step, RBH recorded consistently low number of failure (< 25) for

the rest of the simulation run. The number of failures recorded by RTS declined abruptly after that first transition time step (only one failed node at time step 4000). This number peaks at 12 for the first transition period. Thereafter, the number increases up to 49 failed nodes in the second transition period. The desired minimal number of failed nodes only occurs at the third transition period (after 11349 time ticks), where the number of failed nodes peak at 3. This minimal failed nodes behaviour is sustained in transition periods 4 and 5. On the other hand, NRBH and NRTS are observed to behaved similarly to the one oscillatory frequency case, by oscillating between success and failure across alternate transitions.

The behaviour of RBH, RTS, NRBH, and NRTS in the ten oscillatory frequency case are consistent with those observed in the five oscillatory frequency case as shown in figure 5.16. The achieved SLA compliance across the three oscillatory frequencies (1,5, and 10) is depicted in figure 5.17.

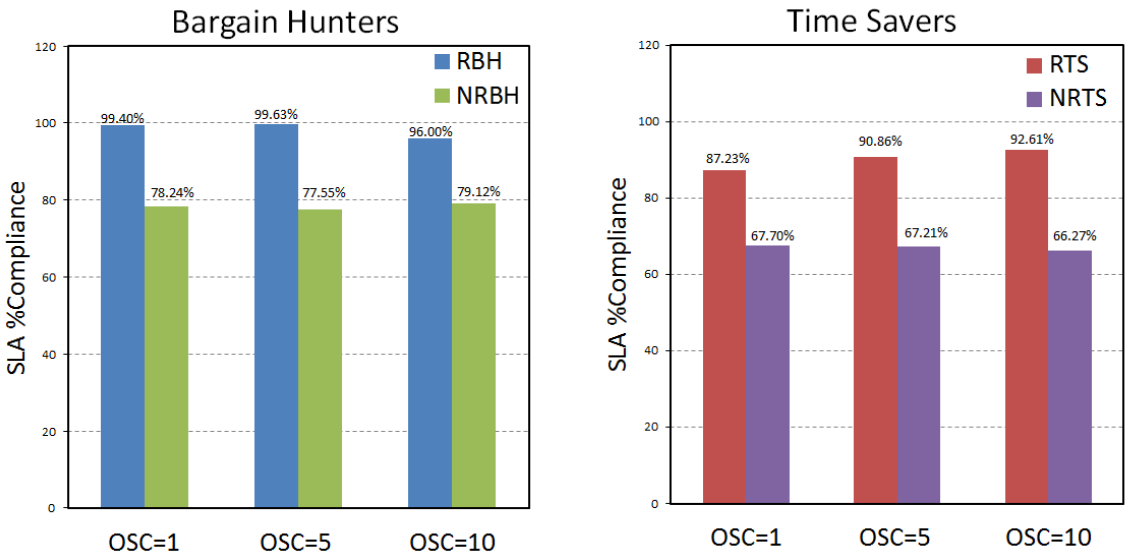


Figure 5.17: SLA Compliance for Oscillatory Frequencies (OSC) 1, 5, and 10

5.8 Conclusion

Self-aware software architecture for decentralised federated cloud applications require a mechanism to allow application components coordinate their interaction. We instantiated a candidate self-aware cloud architecture within the context of the online cloud shopping application introduced in chapter 1. This chapter investigated market-based control as a viable solution concept for coordinating decentralised self-aware architectures. We reviewed the literature on market-based control for cloud and identified the posted offer market mechanism as a candidate mechanism for the purpose of evaluating our self-aware cloud architecture. We proposed our refinement to the canonical mechanism namely *reputation-aware posted offer mechanism* to track changing reliability of seller agents, which are equivalent to cloud services.

We empirically studied the self-aware architecture and its posted offer market adaptation mechanism using both synthetic and realistic workload. The study was conducted to find out the behaviour of the solution approach under different scenarios consisting of cloud services across high and low resilience bands at scales ranging from minimal to high scale. Two self-expressive adaptation strategies namely time-saving and bargain hunting buyer agents were studied. We compared our reputation-aware mechanism with classic non-reputation aware posted offer mechanism for all cases. Results indicated that our reputation-aware mechanism achieved higher SLA compliance than classic non-reputation-aware posted offer mechanism with minimal overhead.

Our approach provides a methodology for architects of self-adaptive cloud applications to study achievable SLA compliance levels under various scenarios. Results such as those obtained from our simulation studies provides guidance for engineering adaptive strategies for applications running on real cloud infrastructure.

We do not claim that our proposed mechanism is optimal for all scenarios, rather, we have chosen the posted offer market mechanism and extended it due to its ability to fit the decentralised nature of self-adaptive architectures studied in this thesis (see chapter 4). Similarly, the reputation model used in our empirical study could be improved upon,

for example by including anticipatory/predictive learning capabilities.

Real deployment of our mechanism will inevitably introduce new implementation constraints that may require revising our solution approach. For example, components and APIs provided by cloud providers differ in their interfaces and offered services. We note that our results exclude these real-world deployment issues and speculate that consideration of these issues may reveal new opportunities for tuning our self-aware cloud architecture to cater for them.

CHAPTER 6

TRADE-OFF AND RISK ANALYSIS OF SELF-AWARE CLOUD SOFTWARE ARCHITECTURE

> "Qualitative analysis transforms data into findings. No formula exists for that transformation. Guidance, yes. But no recipe. Direction can and will be offered, but the final destination remains unique for each inquirer, known only when - and if - arrived at"
>
> *Michael Quinn Patton*

6.1 Overview of the Chapter

The primary focus of this chapter is to evaluate self-aware architecture patterns presented in chapter 4 within the context of a federated cloud application. We perform a two-phased qualitative evaluation to meet the thesis' objective of providing an approach for systematically reasoning about the design and analysis of cloud architectures.

The objective of the first phase is to evaluate a self-aware cloud architecture relative to architectures induced by DDDAS [54] and 3-Layered [105] styles to unveil risk and trade-off points in the architectures. The objective of the second phase is to assess the coverage of the self-aware patterns and the ease of interpreting the patterns in application domains outside cloud. Given the qualitative nature of the evaluation, we take precautionary steps

to eliminate bias in our conclusion by enlisting independent stakeholders and self-adaptive application designers as evaluators in both phases of our evaluation.

The Architecture Trade-off Analysis Method (ATAM) [97][48] was used in the first phase of our evaluation. ATAM is a mature and well validated scenario-based evaluation method that has been successfully applied in many software domains [96][48][16]. The choice of ATAM as a method of evaluating cloud-based applications has been explored by [72]. In [72], we used ATAM for architectural analysis of cloud software deployed in unpredictable, resource-constrained environments.

Two software architectures instantiated based on DDDAS [54] (simulation-based adaptation) and 3-Layered [105] (hierarchical) styles are used as baseline for comparison. The three candidate architectures (self-aware, DDDAS, and 3-Layered) make use of our reputation-aware mechanism (see chapter 5) for coordinating components. Independent stakeholders analysed these architectures to uncover risk and trade-off points in each case.

Findings from independent stakeholders in the qualitative ATAM evaluation suggests that within the context of the evaluated architectures: self-aware style is more likely to offer higher levels of scalability and availability than DDDAS and 3-Layered, and probably comparable to DDDAS in timeliness of adaptation, but most likely to be worse off in data consistency. In conformance with practices in software architecture evaluation [10][97], these findings should be interpreted as outcomes of design-time analysis of candidate architectures and not findings from implemented architectural instances. The findings, though subjective, serve as indicators of expected behaviour of the studied styles.

In the second phase of our evaluation we engaged application designers outside the cloud domain. The main objective was to understand the extent to which the style provided a systematic support for design-time analysis of each application's self-adaptive qualities. Specifically, four independent application designers have architected their respective applications based on principles of self-aware architecture pattern.

Feedback received from independent assessors indicated that the style provided a systematic approach to instantiating self-adaptive architectures and helped them uncover

subtle trade-offs in their application's self-adaptive architectures. However, three gaps were identified in our original self-aware architecture patterns: (i) physical interconnections between components needed to be made explicit, (ii) two additional patterns needed to be added to reduce the complexity of the architecting process for non-interactive applications, and (iii) the methodological approach of the architecting process needed to be improved. These gaps were filled by revising the architecture patterns accordingly, to address limitations (i) and (ii), and an improved methodological approach was derived based on our evaluation method, to address limitation (iii), as presented in [40].

The rest of this chapter is structured as follows. Section 6.2 introduces the ATAM evaluation method. The self-adaptive architectures induced by the 3-Layered, DDDAS, and Self-aware architecture styles are the focus of sections 6.3, 6.4, and 6.5 respectively. Findings from independent stakeholders during the ATAM evaluation are discussed in section 6.6. Evaluation by independent application designers and our findings from this exercise are presented in section 6.7. We conclude the chapter in section 6.8

6.2 Evaluation Method

The candidate self-adaptive architectures analysed in this chapter are instantiated within the context of the online shopping cloud application introduced in chapter 1 (section 1.5). To meet the objective of the first phase our evaluation, we aim to answer the following research question:

> Does design-time risk and trade-off analysis of self-aware cloud architecture indicate better quality of service expectations when compared to architectures induced by DDDAS and 3-Layered?

6.2.1 Architecture Trade-off Analysis Method (ATAM)

This section presents the ATAM evaluation method and the motivation for choosing it for the first phase of our evaluation.

Two approaches to analysing adaptability properties of self-adaptive architectures are quantitative and qualitative approaches. In the former, metrics for analysing trade-off between adaptability and other QoS (e.g. availability, performance, and security) are specified [133]. The work of [153], [58], [94], and [136] subscribe to the quantitative approach, however, this approach is not widely adopted, since comparing adaptive properties of systems, which may have different goals, is both hard and expensive.

In the qualitative approach, metrics such as level of separation of concern, organisation of components (centralised or decentralised), and architectural support for learning are considered. We have already defined these metrics and used them to compare self-adaptive architecture styles in chapter 3. By relying on scenario-based methodology [10], we are able to evaluate self-adaptive architectures using architectural evaluation methods.

The goal of the ATAM is to analyse architectural approaches with respect to scenarios generated from business drivers for the purpose of identifying risk points in the architecture [95]. This is achieved by a disciplined reasoning about software architecture relating to multiple quality attributes [10]. There are two important classifications of risk points in ATAM namely *sensitivity points* and *trade-off points*. A sensitivity point refers to a parameter of the architecture that affects the achievement of one quality attribute. On the other hand, a trade-off point refers to a parameter of the architecture that affects the achievement of more than one quality attribute, where one improves and the other degrades. These risk points, together with documentations of the architecture, scenarios, and quality-attributes analyses are the products of ATAM.

An ATAM evaluation is conducted in a workshop-style setting, where stakeholders (e.g. researchers and implementers) collectively evaluate candidate software architectures. The main principle of ATAM is that probing architectures from multiple perspectives by stakeholders, who did not design the architecture, is likely to unveil design decisions that could potentially pose risks to the software system [96]. Architecture evaluation results can be used to refine architectures, to guide selection of a candidate architecture, and to learn about properties of implemented systems. A comprehensive description of ATAM

is presented in [97].

6.2.2 The ATAM Workshop

A workshop was conducted to evaluate the three architectures presented in this chapter using ATAM. Participants included the author (responsible for designing software architectures) and four stakeholders who are knowledgeable about self-adaptive software systems (responsible for architectural analysis). The architectures presented in this chapter have previously benefited from feedback obtained from two masters projects supervised during this research programme[84] [159]. The feedback has contributed to the understanding and refinement of the architectures. The workshop activities were organised in line with steps of an ATAM evaluation [97]. Each activity and its objective, participants, and result are summarised in table 6.1. A template of the form used to structure feedback from stakeholders is shown in appendix C.

Only two stakeholders had past experience conducting an ATAM evaluation, therefore it was necessary to present the rationale for ATAM and some architectural analysis exercises. The key message to stakeholders was that *the expected outcome of an ATAM is to identify architectural design decisions that could pose potential risks to the software system.* Stakeholders found the online shopping application easy to follow as they claimed to interact with such systems in everyday life. Stakeholders were given an overview of each architecture style and exemplars to illustrate underlying concepts.

Candidate architectures of the online shopping application as induced by each architecture style were presented. Stakeholders were able to grasp the architectural concepts, however, the workings of the market-based adaptation mechanism was not immediately intuitive. Hence it was necessary to explain the market-based mechanism to help stakeholders understand how components interact in the architectures. The architectural documentation was subsequently updated to explicitly capture the workings of the market mechanism. As shown in table 6.1, stakeholders derived the utility tree, analysed the candidate architectures, and presented their findings.

Table 6.1: ATAM Workshop Activities

S/N	Activity	Objective	Participants	Result
1	Presentation of ATAM	To present the underlying concepts and main rationale for conducting an ATAM	Author and stakeholders	N/A
2	Presentation of Business drivers	To present the problem of SLA compliance in the online shopping application case study using scenarios	Author and stakeholders	N/A
3	Presentation of architectural styles	To present the key properties and underlying primitives of the 3 architecture styles to be considered	Author and stakeholders	N/A
4	Presentation of architectures	To explain the architecture and key design decisions of the 3 candidate architectures to be evaluated	Author and stakeholders	N/A
5	Derivation of utility tree	To derive the main quality attribute requirements and constraints for scenarios presented in (2)	Stakeholders	The utility tree presented in section 6.2.3
6	Analysis of Candidate architectures	To analyse each candidate architecture for risks, sensitivity points, and trade off points	Stakeholders	Analyses of candidate architectures as presented in sections 6.3 - 6.5
7	Architectural documentation	To update missing details in the presented architecture as suggested by stakeholders	Author	Description of candidate architectures as presented in sections 6.3 - 6.5
8	Documentation of findings	To document key findings uncovered during analysis as they relate to architectural risks, sensitivity, and trade off points	Stakeholders	Findings as presented in sections 6.3.2, 6.4.2, 6.5.2, and 6.6

6.2.3 Analysing Trade-offs using Utility Tree

The first step in conducting an ATAM analysis is to make the quality attributes to be measured explicit. These attributes form the basis against which candidate architectures would be compared to unveil the extent to which they meet the system's goals. An utility

tree is a tool for explicating quality attributes by categorising scenarios into quality attribute dimensions and prioritise scenarios in the order of their importance to the systems. Three levels of prioritisation are available: (H)igh, (M)edium, and (L)ow. The notation (X,Y) is used to annotate each scenario, where X denotes the importance of the requirement to the system relative to other requirements, and Y is the risk or effort associated with the realisation of that requirement [97].

Within the context of the cloud federation scenario described in section 1.5 presented in chapter 1, stakeholders elicited an excerpt of quality attributes requirements that are crucial for adaptability property of the architecture and represented them using the utility tree shown in figure 6.1. Adaptability here refers to *the ability of the system to successfully accommodate change while executing* [109].

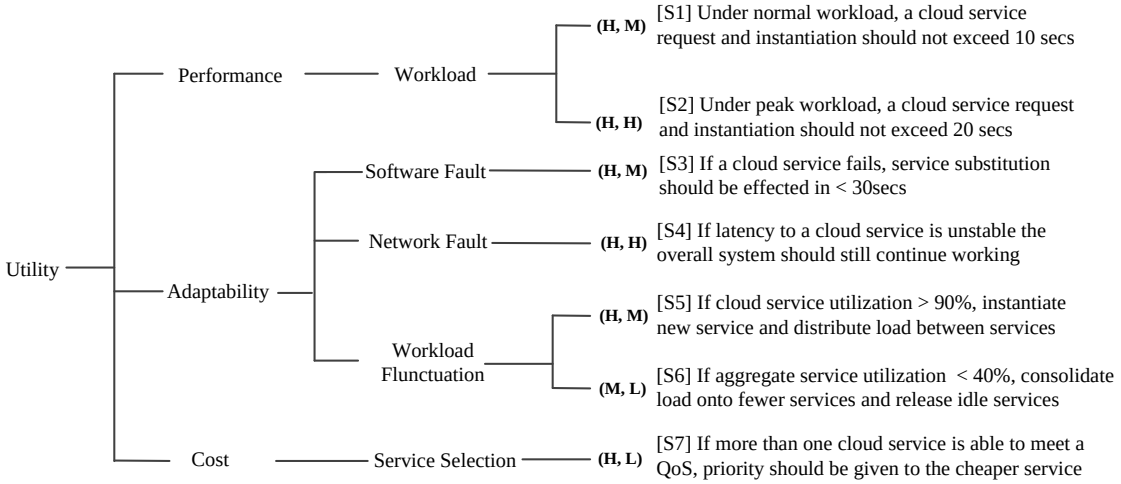


Figure 6.1: Utility tree for Adaptation Engine Subsystem of Online Shopping Application

Using the quality attributes in the utility tree and the requirements, stakeholders were able to probe the self-adaptive architectures using attribute-specific questions to find out architectural decisions that may lead to risks, sensitivity points, and trade-off points.

6.3 Case 1: Evaluation of self-adaptivity and trade-offs in 3-Layered Architecture

This section presents the self-adaptive architecture for the scenario presented in section 1.5 of chapter 1 using the 3-Layered architecture style [105].

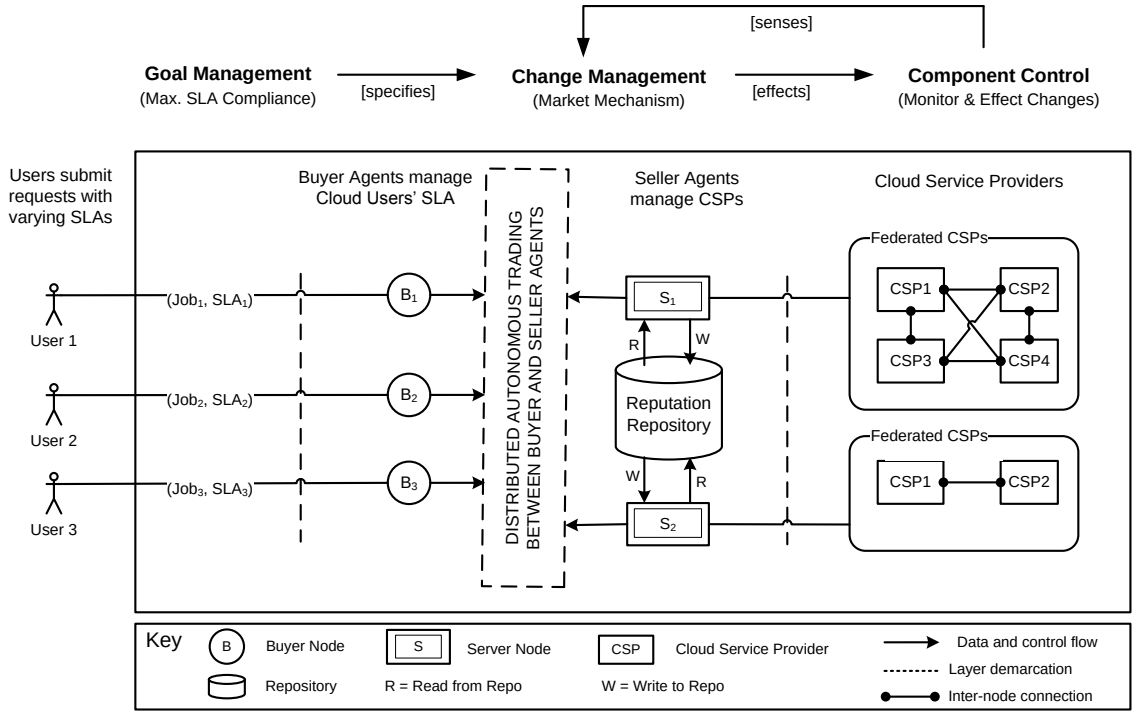


Figure 6.2: Cloud Architecture Induced by 3-Layered Architecture Style. Source: [70]

The reference architecture [105] consists of three distinct layers: goal management, change management and component control layer. The component control layer, at the bottom, is where components are created, deleted, bound and unbound to monitor the managed system and execute adaptation changes. The change management layer, at the middle, consists of pre-compiled plans for responding to change requests from the lower (component control) layer or upper (goal management) layer. The goal management layer is where user goals are specified and new plans are generated to meet unforeseen adaptation needs. Our instantiation of the reference architecture model [105] for decentralised control is depicted in figure 6.2.

In the architecture, the change management layer incorporates a decentralised mech-

anism for making the architecture resilient and scalable in the presence of component failures at the lower layer. Our reputation-aware posted offer market mechanism, presented in chapter 5, is used to realise the change management control. The posted-offer mechanism is preferred because when compared to other economic auction models (e.g. continuous double auction and bilateral bargaining), it has the benefit of saving the time spent on negotiation and provides the flexibility for buyers to rapidly switch among multiple sellers [34].

The layers of the architecture in figure 6.2 are described as follows:

6.3.1 Online Shopping Application Induced by 3-layered Architecture style

- **Goal Management**

  SLAs typically encompass the goals and requirements of cloud users' jobs as agreed with the cloud provider. Therefore, the goals which dictate the objectives of buyer agents in the market mechanism are elicited from these SLAs. The market-based mechanism makes resource allocation decisions based on these goals to make a best-effort attempt to ensure SLAs are not violated.

- **Change Management**

  Once the buyer agents are equipped with knowledge about the goals of the cloud users as defined in the SLAs, buyer agents enter into negotiation with seller agents to make decision about which cloud resource is most capable of executing the cloud user's job with the lowest probability of violating the job's SLA. This negotiation and subsequent resource allocation is carried out via a reputation-aware posted offer market mechanism (see chapter 5) which utilises information about seller agents' reputation to measure their reliability. A high-level description of components in the buyer and seller nodes is shown in figure 6.3.

- **Component Control**

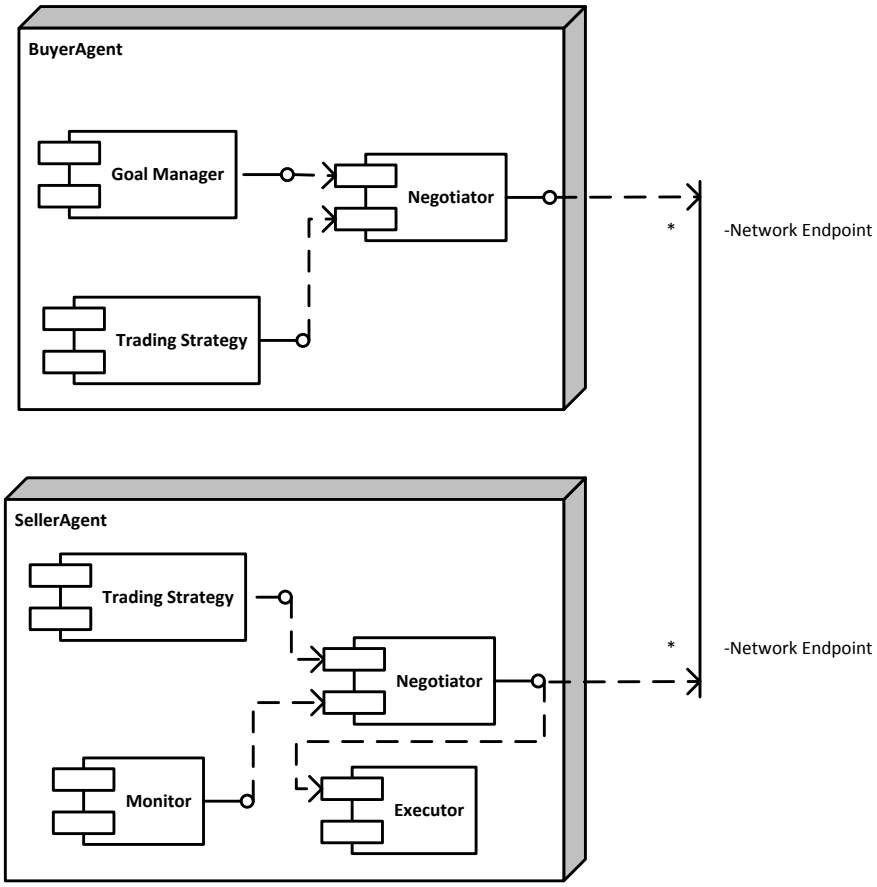


Figure 6.3: Components of Buyer and Seller Nodes. Source: [70]

Cloud resource nodes are analogous to seller agents in the architecture. Seller agents could be designated as manager components of cloud service providers. These seller agents are equipped with components for monitoring the current state of the executing job and their own resource usage or health status. In addition, executor components which carry out resource allocation actions as directed by interaction with buyer agents or other seller agents are present in each seller agent node.

6.3.2 Analysis of Architectural Decisions

Following analysis of the 3-layered cloud architecture of figure 6.2, the following findings were recorded.

Risks

1. In scenario S6, when consolidating load to a service and turning off idle services, there is no mechanism for deciding which of the currently running under-utilized services should be kept running. This decision has implication for overall service provisioning cost, as it is more cost-efficient to keep the cheaper of two services, with comparable QoSes, running than to transfer load to the more expensive one.

2. The mechanism for detecting changes in latency, in scenario S4, on the channel connecting the cloud-reliant application to the cloud service is not specified. It is crucial to explicitly state the mechanism that will be used to detect changes in latency and mechanism for tuning sensing threshold to avoid problems of too frequent sensing or delayed sensing, hence, leading to incorrect adaptation.

Sensitivity Points

1. The performance of the adaptation subsystem is sensitive to the time required to acquire knowledge about customer SLAs from the goal management layer to the change management layer.

2. The availability of the adaptation subsystem is sensitive to the uptime of the reputation repository. Hence, the repository constitutes a single point of failure.

3. The integrity of data written or read from the reputation repository is sensitive to the transactional consistency of the read/write operation. If improperly handled this may lead to corruption of reputation records, and consequently incorrect adaptation.

4. The throughput of the adaptation subsystem (i.e. the number of cloud services instantiated per unit time) is sensitive to the availability of cloud services and the latency of the communication channel used for sourcing services.

Trade-off Points

1. Adaptability versus Performance

   This is one of the key findings of the analysis, as it is manifested in different forms in the architecture. As the occurrence of software faults increases (scenario S3), the performance overhead required to find suitable services also increases. This implies that improving the resilience of the architecture via the adaptive mechanism is likely to degrades performance, especially for the peak workload scenario S2.

2. Adaptability versus Cost

   The high service utilisation induced by spikes in workload (scenario S2) necessitates instantaneous sourcing for additional services (scenario S5). The combination of scenario S2 and S5 implies costly services may be selected to meet emergent application needs due to the performance constraint specified in scenario S2. This situation conflicts with the underlying business rationale of scenario S7. In other words, improving the scalability of the architecture on one hand may compromise the cost-effectiveness of the service provision.

6.4 Case 2: Evaluation of self-adaptivity and trade-offs in DDDAS Architecture

The cloud federation architecture induced by DDDAS is shown in figure 6.4. The DDDAS adaptive layer in the architecture realises the adaptation mechanism use to coordinate and control CSPs. It is composed of distributed simulator instances. These simulators receive requests from cloud users and select cloud providers capable of meeting these requests. The simulators receive control feedbacks from cloud providers signifying successful execution of submitted job requests or *risk alerts* signifying their inability to do so. Risk alerts could be triggered by reasons such as unanticipated resource failure and unforeseen spike in workload. The simulators act on this feedback by taking risk mitigation actions such

as selection of substitute cloud provider(s) to avoid violating the SLA terms of submitted requests. Feedbacks received by the simulator can also be discriminated as: high priority (requires immediate intervention), medium priority (react within a time bound) or low priority (trivial, non-threatening risk). The knowledge acquired from the continuous interaction between simulators and CSPs improves the accuracy of the simulators at selecting reliable CSPs. Next, we zoom into the details of adaptive layer of the architecture.

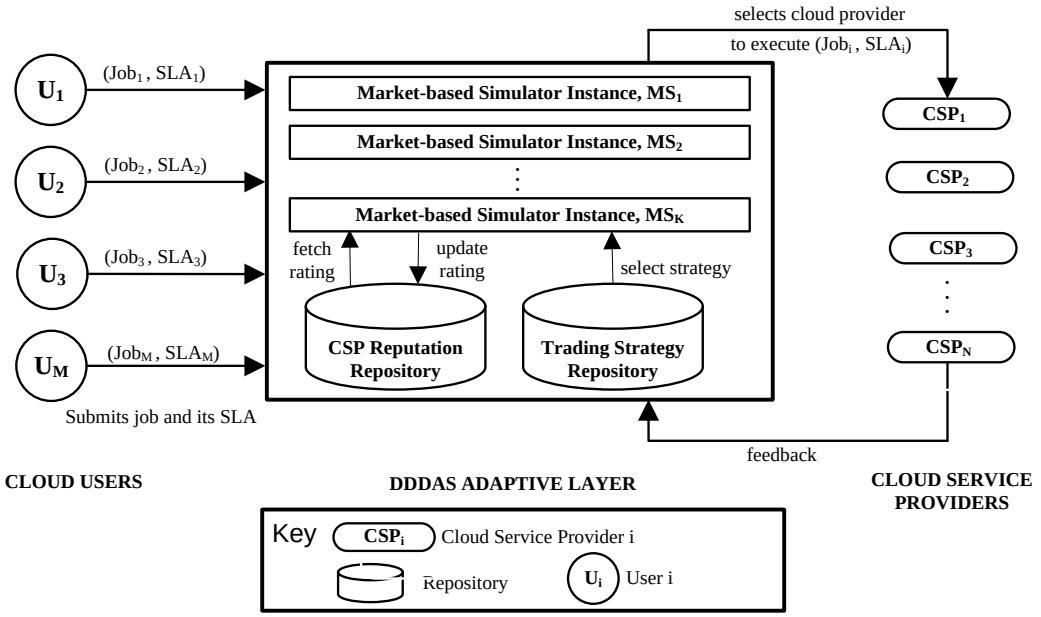


Figure 6.4: Cloud Architecture Induced by DDDAS Architecture Style. Source: [70]

6.4.1 Onling Shopping Application Induced by DDDAS Architecture

The adaptive layer consists of many simulator instances which coordinate the cloud federation. The distributed simulation approach is preferred due to the scale of the cloud federation. A single simulator instance may constitute a bottleneck under heavy workload and thus make the adaptation subsystem less scalable. Distribution of simulator instances improves the scalability of the adaptation subsystem and affords dedication of simulators to multiple concerns. For example, some simulators may be more efficient at selecting CSP for specific types of incoming job requests than others. In accordance

with the DDDAS paradigm, these simulators collect data based on current state of the cloud, perform measurements to detect probable violations, and effect control changes to mitigate them.

More precisely, the simulators performs the following functions:

- receive job requests and associated SLAs as input from cloud users,
- inspect offerings of cloud service providers at specified intervals,
- select CPS(s) to execute job requests based on job type and SLA terms,
- monitor job execution to detect risk alerts from cloud providers, and
- effect actions to prevent the violation of users' SLAs whenever a risk alert is received.

Simulator instances select CSPs by utilising our reputation-aware posted offer market mechanism. The steps of the reputation-aware posted offer mechanism for each trading round is described as follows:

Step 1: sellers (CSPs) publish the prices and service terms of their resource offerings

Step 2: buyers search for active sellers in the reputation repository

Step 3: if buyer finds a matching seller, allocate job to it, then step 4, otherwise step 2

Step 4: buyer (simulator instance) monitors the selected seller (CSP) at intervals

Step 5: if seller successfully executes job then sends success notification else raise alert

Step 6: if buyer detects risk alert or seller (CSP) is not responsive then reputation repository is updated with a negative rating for the CSP and transfers control to step 2 else step 7

Step 7: if CSP violates SLA constraint, reputation repository is updated with a negative rating for the CSP otherwise it is updated with positive rating

Step 8: cloud user is notified of completed job

6.4.2 Analysis of Architectural Decisions

Risk

1. SLA constraints are the drivers for deciding which trading strategy is used to meet a job request, however, the impact of mixing strategies is not explicitly stated. While the mechanism is able to accommodate diverse trading strategies, the overall impact on the architecture's adaptive behaviour is not explicitly stated.

Sensitivity Points

1. The computational overhead of the adaptive layer is sensitive to the number of market simulators active per unit time.

2. As with the 3-layered architecture, the repositories constitute single points of failure. The impact of failure of the trading strategy repository is likely to be catastrophic because it encodes the logic of *how* CSPs are selected.

3. The accuracy of allocation decisions made by the simulators is sensitive to their training time. That is the time spent initialising the system to look ahead, acquiring knowledge about cloud services, before actually making selection decisions.

4. The integrity of data written or read from the repositories are sensitive to the transactional consistency of the read/write operation. If improperly handled this may lead to corruption of reputation records, and consequently incorrect adaptation.

Trade-off Points

1. Adaptability versus Cost

   In order to make correct adaptation actions, simulators should continue running in order to maintain the current state of the real-world. However, this consumes computational resources, which is costly, especially in the variant of the architecture where simulators are dedicated to various job types.

2. Adaptability versus Performance

   During peak workload scenario (S2), the computational overhead incurred by the simulators is likely to negatively impact the performance constraints specified for service instantiation, hence, leading to violation of the SLAs.

6.5 Case 3: Evaluation of self-adaptivity and trade-offs in Self-aware Architecture

In chapter 4, the self-aware architecture style was presented and we reiterated its adherence to a *decentralisation by design* paradigm. Crucially, the style models knowledge at a more granular level when compared to state of the art self-adaptive architecture styles. In chapter 5, we already presented a self-aware architecture that realises the requirements of the cloud service provisioning problem highlighted in section 6.2.3. The rest of this section overviews the key principles of self-awareness as they relate to the architecture presented in section 5.6 of chapter 5.

6.5.1 Online Shopping Application Induced by Self-aware Architecture

- **Stimulus-aware Component** This characterises spikes or dwindles in user request traffic. It models the workload and distinguishes them across different SLA classes (e.g. premium and normal SLA).

- **Interaction-aware Component**

  Knowledge about the interaction between the adaptation subsystem and cloud services is captured by this component using the performance repository. This functionality is realised by storing location information (e.g. RESTful service URI) about cloud services and facilitate the connection to these services.

- **Time-aware Component**

This component makes use of a locally managed performance repository to store rating of services, in terms of the level to which the service met promised QoS, and the duration of their use. The posted offer mechanism encapsulates the rule for computing the reputation rating of cloud services. One of the implications of a decentralised architecture is that each adaptation subsystem maintains different service performance repositories. This raises the issue of timeliness or recency of ratings. Relying on obsolete reputation rating may lead to poor adaptation.

- **Goal-aware Component**

  Given an application QoS, this component makes use of an utility function to deduce the candidate services which are likely to provide optimal QoS. An implication of the decentralised architecture is that utility functions used for goal representation may be formulated in such a way that different adaptation subsystems value SLAs differently. A good application of this is when buyer agents are distinguished based on the workload across different SLA classes. Premium users may be serviced by more strategic and sophisticated buyer agents than normal users.

- **Self-Expression Component**

  It makes service selection decisions based on allocation strategies. Similar to the 3-layered architecture, service selection strategy is managed locally. This has the advantage that the adaptation subsystem can be easily specialised, by adding or removing trading strategies, without affecting the rest of the system. As discussed in chapter 5, two strategies studied in this thesis are bargain hunter and time saver strategies. Bargain hunters choose service with the lowest price possible. That is, the selling price must be the lowest among available services. If more than one service offers the lowest price, then one is chosen at random. Time savers choose services at random, provided the price is acceptable, i.e., price less or equals application's budget.

6.5.2 Analysis of Architectural Decisions

Sensitivity Points

1. Unlike the 3-layered and DDDAS cloud architectures, the service performance repository is locally maintained by each adaptation subsystem. The currency, i.e. up-to-datedness, of each adaptation subsystem is sensitive to the amount of interactions with (possibly different) cloud services in the market.

Trade-off Points

1. Adaptability versus Cost

   Depending on the strategy selected by the self-expression component, there exist a trade-off between the time to select a service and the cost of the service. For example the bargain hunter and time saver strategies are at different points in this trade-off space as shown in figure 6.5. While adaptability makes timely service instantiation possible, the cost of the adaptation is high.

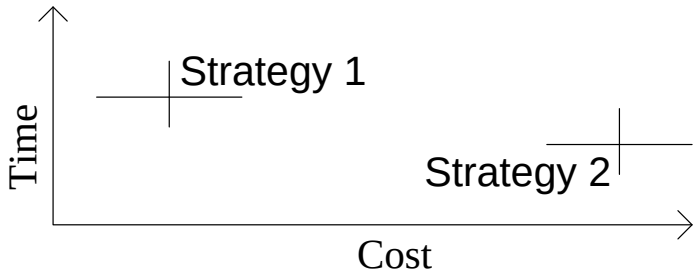


Figure 6.5: Illustrates the trade-off space between time to select a service and the cost of searching for that service using Bargain Hunter and Time Saver strategies

2. Adaptability versus Accuracy

   The decentralised nature of the architecture means that each adaptation subsystem only has a local view of the cloud service market. This limits its ability to make optimal service selection decision, as it may have little historic knowledge about a candidate cloud service, whereas another application's adaptation subsystem might

have knowledge about the service's recent performance. This means improving resilience of the architecture via adaptability by eliminating single points of failure impacts the ability of adaptation subsystems to make optimal resource allocation decisions.

3. Adaptability versus Communication Load

   To address the trade-off point between adaptability and accuracy, there is a need to communicate reputation ratings of cloud services as observed by one adaptation subsystem to others. This way knowledge about a cloud service's *true* performance can be propagated among adaptation subsystems. However, this introduces a trade-off between adaptability and communication, as the propagation of knowledge to improve adaptability imposes higher communication overhead on the network than usual. A key design decision in the self-aware architecture is to use a knowledge propagation mechanism such as gossip protocol that is less communication intensive.

6.6 Comparative Analysis of ATAM Results

Based on findings from qualitative evaluation using ATAM in section 6.3, 6.4, and 6.5, this section presents implications of these findings from perspective of stakeholders. A comparative analysis is presented to compare and contrast the studied architectures in order to find out how the essential features of their underlying styles answers the question that triggered the qualitative evaluation exercise:

> Does design-time risk and trade-off analysis of self-aware cloud architecture indicate better quality of service expectations when compared to architectures induced by DDDAS and 3-Layered?

Table 6.2 summaries the findings from the qualitative evaluation. The analysis by stakeholders of the implications of the findings follows.

	Risks	Sensitivity Points	Trade-off Points
3-Layered	2	4	2
DDDAS	1	4	2
Self-aware	0	1	3

Table 6.2: Summary of Findings from Qualitative Analysis. In conformance with practices in software architecture evaluation[10][97], these findings should be interpreted as outcomes of design-time analysis of candidate architectures and not findings implemented architectural instances. The findings serve as indicators of expected behaviour of the studied styles.

6.6.1 Risks

Both 3-Layered and DDDAS cloud architectures recorded 2 and 1 risks respectively. It is interesting to note that the risk identified in both architectures affect different quality attributes. Specifically, it was found that mechanisms were missing in the 3-Layered architecture for addressing cost implication of architectural decision in the service consolidation scenario (S6) and detecting changes in latency in scenario S4. The former may lead to costly adaptation, while the latter may lead to incorrect adaptation in unstable network scenario (S4). On the other hand, the sharing of trading strategy repository in the DDDAS architecture has the potential of causing unintended emergent adaptive behaviour, which is undesirable.

No risks were identified in the self-aware architecture. Stakeholders suggested two possible explanations for this: (i) By the nature of the evaluation it is hard to deduce implications of concurrency in the decentralised self-aware architecture. For example, this could be studied empirically as presented in chapter 5, (ii) From an architectural design perspective, fine-grained decomposition of knowledge effectively captured the dimensions of architectural concerns within the context of the case study. It is therefore expected that more complex applications may unveil risks which are not observed in this case study.

The second point is in line with the aim of this thesis, which is to provide a systematic *design-time architectural approach to explicitly capture design decisions and reduce risks* such as unintended, incorrect, and costly adaptive behaviour.

6.6.2 Sensitivity Points

Figure 6.6 summarises the stimuli (triggers for architecture decisions as they affect parameters of the architecture) and responses (affected quality attributes).

	Stimulus	Response
3-Layered	Propagation of SLA from goal to change management layer	Performance
	Uptime of reputation repository	System availability
	Transactional integrity of read/write operation	Data integrity
	Availability of cloud services and latency of communication channel	Throughput
DDDAS	Number of market simulators	Computational overhead
	Uptime of reputation and trading repositories	System availability
	Market simulator training time	Accuracy
	Transactional integrity of read/write operation	Data integrity
Self-aware	Activity level of adaptation subsystem	Currency (up-to-dateness)

Figure 6.6: Comparison of Sensitivity Points by Architecture Style

It can be observed from figure 6.6 that the reputation repository is most critical to the adaptability of the 3-Layered and DDDAS architectures. This can be attributed to the hierarchical and centralised nature of these architectures. Importantly, system availability in both cases is dependent on uptime of the repository and its corruption will affect the integrity of data used for making adaptive decisions.

The 3-Layered and DDDAS architecture exhibit sensitivity to time in different ways. The performance of the adaptation subsystem in the 3-Layered architecture is affected by time required to propagate SLA information from one layer to another. On the other hand, DDDAS market simulators suffer a delay in time spent in training at start up to acquire knowledge about cloud services. Further, the number of simulators impacts the computational overhead incurred by the system.

The self-aware architecture's decentralised design reduces the impact of unavailability of reputation repository in one adaptation subsystem on continued working of other parts of the system. That is, no one point in the architecture is weaker than another. However, the self-aware architecture's reputation repository may become obsolete quicker than those of the other two architectures. This is because the adaptation subsystem in the self-aware

architecture is responsible for acquiring knowledge about cloud services by itself, unlike the other approaches where a shared central repository is present.

It follows that the self-aware architecture can potentially offer higher levels of robustness, availability and scalability when compared to the 3-Layered and DDDAS architectures, however, it offers poorer data consistency when compared to them. An important caveat is that these benefits of self-aware architecture may not be realisable in a centralised deployment, due to likelihood of outdated reputation repository.

6.6.3 Trade-off Points

Figure 6.7 shows the trade-off points uncovered by the analysis of the 3 architectures.

	Tradeoff Points
3-Layered	Adaptability versus Performance
	Adaptability versus Cost
DDDAS	Adaptability versus Cost
	Adaptability versus Performance
Self-aware	Adaptability versus Cost
	Adaptability versus Accuracy
	Adaptability versus Communication

Figure 6.7: Comparison of Trade-off Points by Architecture Style

The trade-off between adaptability and cost is common to all three architectures, although manifested in different forms. In the 3-Layered and Self-aware architectures the improved scalability achieved by automatically selecting cloud services due to spikes in workload may result in costly adaptation. On the other hand, the DDDAS architecture is more likely to accrues cost because simulators are kept running to maintain a real-time view of the world.

Achieving adaptability is likely to conflict with performance in the 3-Layered and DDDAS architectures. In the former the conflict may result from performance overhead imposed on fewer running services when a software fault takes place, while in the latter it may result from performance demands caused by peak workload. This possibilities is likely to limit the ability of these architectures to scale under peak workload when

compared to the self-aware architecture.

Two related trade-off points found in the analysis of the self-aware architecture are adaptability versus accuracy and adaptability versus communication. It was found that the resolution tactics for the former results in the latter. This may be attributed to the decentralised systems in general, where achieving a fully consistent view of distributed data is not achievable. It was suggested that a robust information sharing mechanism is crucial to ensure these trade-off points does not significantly degrade the ability of the architecture to adapt correctly.

Given the findings from the ATAM qualitative evaluation, stakeholders conclude that the self-aware style is more likely to offer higher levels of scalability and availability than DDDAS and 3-Layered, and probably comparable to DDDAS in timeliness of adaptation, but most likely to be worse off in data consistency.

6.6.4 Threat to Validity

In this section, we present the threats to the validity of the result.

- Stakeholders in our ATAM evaluation have formulated use case scenarios and quality attribute requirements to steer the evaluation. Consideration of complex/extreme scenarios beyond the scope of work presented here is therefore a threat to our results.

- Studying two competing self-adaptive styles (DDDAS and 3-Layered) in our ATAM evaluation limits the generality of our results. Studies including other representative self-adaptive styles (e.g. those presented in chapter 3) may show additional benefits and limitations of the self-aware style.

- Stakeholders have analysed the architectures within the limits of the posted offer mechanism and the trading strategies presented in chapter 5. The use of another adaptation mechanism with different properties may reveal additional insights about component interactions and the resultant risks, sensitivity, and trade-off points.

6.7 Reflection on Self-aware Architecture Patterns through External Application Designers

Following the inception of self-ware architecture patterns, four application partners within the EPiCS project were enlisted to independently architect their applications using the patterns and provide feedback on their experience. These applications are: computation of financial pricing algorithms on clusters, distributed smart camera network management, dynamic protocol stack configuration, and operating system for heterogeneous multi-core software/hardware platform. This additional step is done to further eliminate bias from our findings reported in previous sections. Specifically, application designers acted as assessors with a view of answering the following questions:

- How easy is it to systematically interpret self-aware architecture patterns and use them to instantiate architectures within the context of their applications?
- Do the proposed self-aware architecture patterns cover the scope of their applications' architectures and inform a methodical design-time analysis process?

6.7.1 Approach

The approach taken was to disseminate the documentation of the self-architecture pattern to application experts together with questionnaires to capture their feedback about the architecture pattern(s) instantiated in their respective applications. The activities shown in table 6.3 were carried out to assess self-aware architecture patterns.

6.7.2 Key Findings

Need to make Physical Interconnection Explicit

In the original self-aware patterns description, interconnection among self-aware subcomponents in the same node (intra) or across different nodes (inter) were expressed as logical

Table 6.3: Activities for Independent Assessment of Self-aware Patterns

Activity	**Actors**	**Dissemination Channel**
Conception and documentation of self-aware architecture patterns	The author	Report
Clarification of ambiguities and understanding of self-aware architecture pattern	The author and EPiCS teams	Meetings
Dissemination of self-aware architecture patterns	The author and EPiCS teams	Report
Design of self-aware application architectures	Application experts	Independently done by application experts
Collection of feedback from application designers about self-aware patterns and discussion of findings	EPiCS teams and application experts	Workshop with four application experts and other self-adaptive researchers in attendance
Reflection on findings from self-aware patterns	The author and EPiCS teams	Report

interconnection with consideration of actual physical connectors left for architectural instances. However, it was found that showing physical interconnection at the architectural pattern level of abstraction may serve to guide designers about which self-aware subcomponent can exchange data with other subcomponents. Hence, physical interconnections among self-aware subcomponents were expressed in all the patterns as presented in [40].

Incompleteness of Catalogue of Architecture Patterns

It was found that some of the original self-aware patterns as presented in chapter 4 could introduce complexity in certain contexts, hence two of the styles were further elaborated to arrive at simpler variants of those styles. The revised patterns are: Temporal Knowledge Sharing pattern and the Goal Disseminating pattern.

In the case of the Temporal Knowledge Sharing pattern, it was found that for applications where sensitivity to time is required but not interaction with external subsystems, the Interaction-awareness subcomponent could constitute an overhead. Therefore a new variant of the Temporal Knowledge Sharing pattern namely *Temporal Knowledge Aware*

Pattern was derived, without the interaction-aware subcomponent, as shown in figure 6.8.

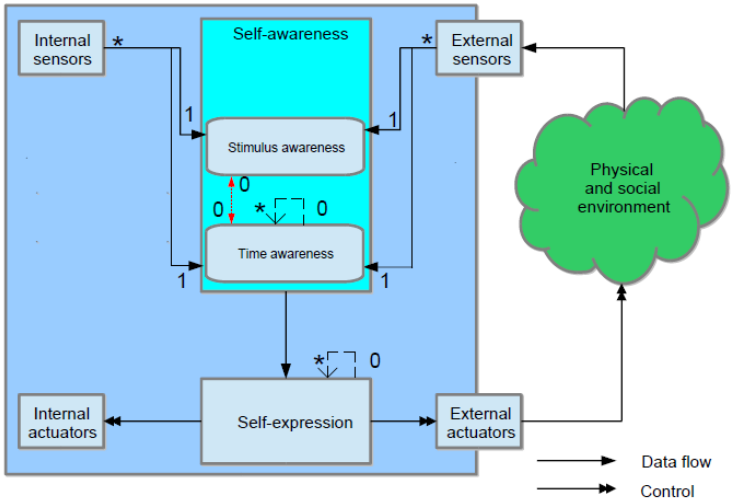


Figure 6.8: Temporal Knowledge Aware Pattern. Source: [40]

Similarly in the case of the Goal Disseminating pattern, it was found that interaction-awareness subcomponent may be an overkill when acceptable adaptation decisions could be reached without sharing goal and time information with other subsystems. A new variant of the Goal Disseminating pattern called *Temporal Goal Aware Pattern* was derived, as shown in figure 6.9.

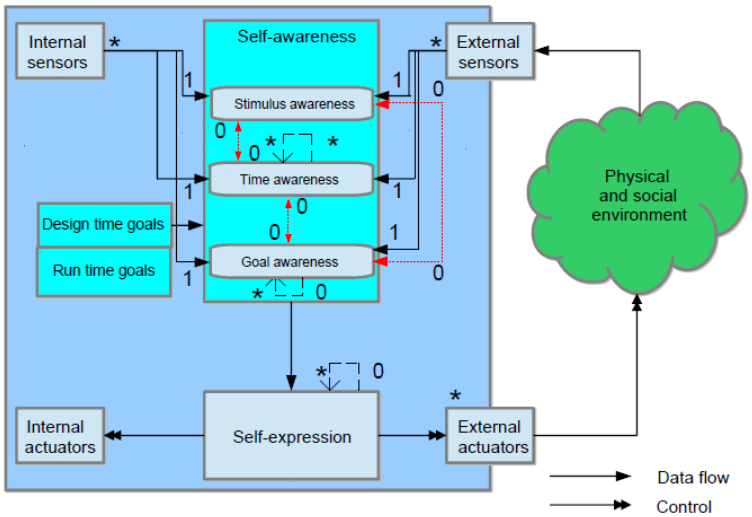


Figure 6.9: Temporal Goal Aware Pattern. Source: [40]

These new architecture patterns and their detailed description can be found in [40].

Need for Improved Methodological Approach to Architecting Self-awareness

Application designers found our original idea of providing a systematic process for engineering self-aware application by confirming the ease of following the process to arrive at instances of architectures for their respective applications. However, application designers argued that a more elaborate methodological approach could help improve the rigour of the architecture design process. Hence, such a methodological approach was proposed in [40], where a question-answer style method was used to elicit designer's intent and score alternative options at every step of the architecting process. It was found that these methodological approach improved the disciplined reasoning about the architecting process and helped application designer explore design decisions more systematically.

6.8 Summary

In this chapter, we have realised the thesis' objective of providing an approach for systematically reasoning about the design and analysis of cloud architectures. A two-phased evaluation approach was used to evaluate the self-aware architecture patterns relative to two competing styles namely DDDAS [54] and 3-Layered [105] architecture styles.

In the first phase of our evaluation, we sought to identify risk and trade-off points in the three candidate architectures using ATAM [97]. Findings from independent stakeholders in the qualitative ATAM evaluation suggests that within the context of the evaluated architectures: self-aware style is more likely to offer higher levels of scalability and availability than DDDAS and 3-Layered, and probably comparable to DDDAS in timeliness of adaptation, but most likely to be worse off in data consistency.

In the second phase of our evaluation, we sought feedback from independent assessors (researchers and application experts) to: (i) find out if external application designers are able to interpret the architecture patterns and instantiate self-adaptive architectures methodologically following the pattern description, and (ii) find out if the self-aware architecture patterns, presented in chapter 4, covered the scope of applications beyond our

online shopping cloud application. Feedback received from application designers indicated three gaps in our original patterns: a need to make physical interconnections explicit, addition of two patterns to the pattern catalogue, and an improved methodological approach to the architecting process. These gaps were filled by revising the architecture patterns and proposing an improved methodological approach as presented in [40].

By reasoning about the architecturing process in terms of our self-aware architecture patterns, we have provided software architects a principled approach to guide their decision about which architecture pattern to instantiate. As it was observed in our self-aware cloud architecture and the four applications evaluated by external application designers, the self-aware patterns are easy to interpret and compare with the underlying characteristics of an application's problem domain. By following our structured thinking approach, ad-hoc trial-and-error selection of patterns is eliminated as the architect is fully aware of the capabilities and limitations of each pattern before proceeding to use it. This contribution is in line with an important open challenge in the self-adaptive architecture community, i.e. the problem of cataloguing available design pattern for self-adaptation and promoting a principled approach to pattern selection and applicability [111].

CHAPTER 7

CONCLUSION AND FUTURE WORK

"There is no real ending. It's just the place where you stop the story."

Frank Herbert

7.1 Overview of the Chapter

This thesis has researched the use of an architecture-centric self-adaptive approach grounded on the principles of self-awareness and market coordination to address the problem of SLA violation in cloud-based applications. The approach termed *self-aware architecture patterns* gains inspiration from concepts of computational self-awareness. Computational self-awareness enables sophisticated autonomic behaviour to be realised by explicitly modelling the knowledge about the system, its environment, and its level of awareness. This thesis has studied the problem of self-adaptation from an *architecture style* perspective [134]. The primary motivation for this approach is to abstract the architecting process to a high level of abstraction that affords studying the underpinning principles that makes one software architecture different from another.

The architectural patterns proposed in chapter 4 and evaluated in chapter 6 has met the thesis' objectives of innovating self-aware architectural patterns and designing cloud software architectures based on the patterns. The reputation-aware market mechanism

proposed and evaluated in chapter 5 has met the thesis' objective of designing a scalable mechanism for coordinating decentralised components of federated cloud applications.

The rest of this chapter is structured as follows. The main contributions of the thesis are summarised in section 7.2. A reflection on the consequence of the research beyond the scope of this thesis and the implication of the results for the self-adaptive architecture community is presented in section 7.3. Avenues for developing the research further is presented in section 7.4. Section 7.5 concludes the chapter.

7.2 Contributions of the Thesis

The following are the main contributions of the thesis.

7.2.1 Self-aware Architecture Patterns

This thesis has contributed a new architectural pattern namely *self-aware architecture pattern* based on the self-aware style to promote a systematic selection and instantiation of architectures. The self-aware architecture pattern has been partly described in [68] and [40]. The patterns was primarily inspired by gaps in the state of the art and a lack of systematic approach to architecting self-adaptive cloud applications. The pattern improves the state of the art by promoting the notions of explicit knowledge modelling for self-adaptive concerns in terms of:

1. Stimuli-awareness - captures knowledge about awareness of input from external sources or components internal to the self-adaptive system

2. Goal-awareness - encodes knowledge about design- and run-time goals of the self-adaptive systems and how these goals evolve throughout the system's lifetime

3. Interaction-awareness - captures knowledge about a self-aware system's interaction with other computational entities or the environment

4. Time-awareness - refers to knowledge about temporal information and how this can be used for time modelling, reactive or proactive adaptation

5. Meta-self-awareness - is the most advanced form of awareness, where a node is aware of its own level of awareness and can coordinate lower levels of awareness (i.e. stimuli, goal, interaction, or time) based on its meta-level knowledge

The benefits of decomposing knowledge to these fine-grains is that it promotes a systematic reasoning about how architectural components realise each quality concern and how one concern complements or conflicts against another. Effectively representing trade-off spaces early in the architecting process makes it possible to engineer solutions for most scenarios the system is likely to encounter at run-time.

We have contributed a catalogue of five architecture patterns for architecting self-aware systems. These patterns are designed to progressively build on each other, making it possible for architects to assess the suitability of each pattern to the characteristics of their problem domain. Experience from architecting applications using our architecture pattern has been positive and it has already been applied beyond the cloud application domain [40]. Although independent experts have identified few notable gaps in the original version of our patterns, the patterns have promoted a systematic way of reasoning about the architecting process. Consequently, opportunities for introducing errors and wrong adaptive behaviour to the architected systems has been reduced.

7.2.2 Decentralised Market Mechanism for Component Coordination

One of the key architectural primitives for architecting self-aware federated cloud applications is the mechanism for coordinating decentralised components. Federated cloud applications face the challenge of operating in an open and highly dynamic environment, hence the need for a robust coordination mechanism. This thesis has contributed a decentralised market mechanism namely *Reputation-aware posted offer market mechanism*

to fill this gap. Importantly, unlike decentralised mechanisms in the literature that cater solely to the problem of coordination, our market mechanism address the problem of filtering out unreliable services by relying on reputation ratings acquired from historic interactions. We have validated the convergence, resilience, scalability, and incurred overhead properties of our mechanism through empirical studies.

While this thesis has mainly considered the use of the proposed market mechanism within the context of three representative self-adaptive architecture styles (DDDAS, 3-Layered, and Self-aware), we hope future work will explore its applicability beyond these styles to validate to what extent it is able to support the primitive of other architecture styles. As suggested in chapter 5, there is room for improving our market mechanism itself. For example, we have used an historic reputation model for the purpose of forecasting likely behaviour of computational entities. A future research direction is to study our market mechanism while incorporating a predictive reputation model (e.g. [90]).

7.2.3 Systematic Review of SLA-based Cloud Research

This thesis has adopted the philosophy of identifying good principles and lessons learnt from related work before tackling research questions pertinent to the thesis. We initially set out with the aim of learning about the gap in SLA-based cloud resource allocation research domain. To achieve this we carried out a systematic literature review (SLR) following the methodology proposed by Kitchenham et al. [103]. Accordingly, this thesis has contributed to the body of systematic reviews in cloud research. We have motivated the core research problems tackled in this thesis based on findings from the systematic review i.e. a lack of systematic architecting process for self-adaptive architectures and limited decentralised coordination mechanisms to support self-adaptive architectures.

7.2.4 Classification Framework for Self-adaptive Architecture Styles

In accordance with the philosophy of this thesis, we explored the literature on self-adaptive architecture styles to know if existing styles offered primitives that met the requirement of SLA-based cloud resource allocation. Within the context of the literature on self-adaptive architecture styles, we noted that the literature is not well structured and the terminologies for describing styles vary largely depending on the research community and problem domain where the style is predominantly used. This thesis has provided a template for structuring the literature in self-adaptive architecture style domain by contributing a classification framework that captures underlying properties of self-adaptive architecture styles and provides a basis for comparative analysis of these styles. In chapter 3, we have used this framework to classify eight representative architecture styles.

7.3 Reflection on the Research

This section reflects on the generality of this thesis' contribution within the Software Engineering (SE) discipline and the usefulness of the claimed results within the specialised domain of self-adaptive software architecture.

7.3.1 Generality of results

It is known that research in Software Engineering undergo a gestation period before they become widely applicable. According to Redwine and Riddle's [139] technology maturity model, software technology ideas progress through six phases: basic research, concept formulation, development and extension, internal enhancement, external enhancement and exploration, and popularisation [139]. Progression from basic research to popularisation typically take 15-20 years [139] [149].

This research work aimed to design a generally applicable self-adaptive architecture pattern and a market-based mechanism to support coordination of components in variants

of the pattern. The author has demonstrated the benefit of the self-aware architecture patterns and market mechanism within the context of SLA-based cloud-based applications. Independent application designers and researchers external to this thesis have adopted and explored the use of the architecture patterns within application domains outside the scope of this thesis namely: computation of financial pricing algorithms on clusters, distributed smart camera network management, dynamic protocol stack configuration, and operating system for heterogeneous multi-core software/hardware platform. The exploration of the self-aware architecture patterns remains a subject of ongoing research [40].

7.3.2 Usefulness of results

We reflect on the usefulness of the results of this thesis within the context of architecting self-adaptive systems in general. The main objective of this thesis is to provide a systematic and principled approach to architecting self-adaptive systems in a way that promotes fine-grained analysis of risks and trade-offs of federated cloud applications. This objective has been realised by contributing self-aware architecture patterns, which the author claims are useful and timely towards easing the architecting process of self-adaptive systems. By decomposing the knowledge concerns of a self-adaptive system to fine-grains (stimuli, time, interaction, goal, and meta), the architect is able to reason about each concern independent of others to uncover sensitivity points and also reasoning about how each concern promotes or conflicts with others to uncover trade-off points. Additionally, the self-aware architecture patterns provide a catalogue from which architects can choose which pattern(s) meet the requirements of their problem domain based on intrinsic properties of the patterns and exemplars of their use in other applications.

7.4 Future Work

This section explores some avenues for building on this thesis for future research work.

7.4.1 Tool Support for Self-aware Architecture Design

Computer Aided Software Engineering (CASE) tools provide a means of analysing and verifying quality of designed software before implementation. In the software architecture community, CASE tools have been used to support so-called Architectural Description Languages (ADLs) [49] such that architects are able to visually explore the design space of the system-to-be. Acme [77] and LTSA [163][74] are some notable examples of tools that have aided adoption of architectural practices.

One important future work is to provide tool support for modelling and verifying designs of self-aware systems. This could be achieved by complementing existing ADLs with profiles to support visual design and modelling of self-aware architectures. Each design pattern proposed in this thesis will therefore correspond to a profile. New patterns can be abstractly added to such tools by creating new profiles corresponding to the new patterns. Successful implementation of such a tool will improve exploitation of our architectural patterns among researchers and practitioners.

7.4.2 Heterogeneity of Architectural Patterns among Self-aware Nodes

The self-aware software architecture of a decentralised system, where nodes cooperate or compete, to achieve some specified goal(s) implicitly assumes that interacting nodes are structurally identical. However, in cases where self-aware nodes belong to different entities, it is not uncommon for nodes to instantiate structurally distinct self-aware architecture patterns. For example, one node may adhere to the Goal Disseminating pattern (with time-awareness capability) while the other realises the same pattern without time-awareness capability. This introduces a challenge of how to explicitly map self-aware subcomponents of structurally distinct self-aware nodes to one another. One potential approach to address this challenge is for self-aware nodes to declare beforehand how they are structurally organised; suppose they are structurally dissimilar, we suggest utilising a negotiation protocol to map out their differences and resolve structural conflicts. It

expected that some of these resolutions would be invalid. Therefore, it is crucial to devise a mechanism for checking the space of structural resolutions and filter out invalid ones.

7.4.3 Systematic Market Mechanism Selection for Cloud Architectures

This thesis has demonstrated that market-based approaches are promising at adapting to changing user requirements and operating conditions in cloud-based application domain. However, there is a lack of guidance on how to map these market mechanisms to specific cloud architectures. The question of which market mechanism is suitable for different architecture patterns and in what context remains open. In order to demonstrate the feasibility of market-based approaches in general and to gain assurance about their behaviour at run-time, it is important to evaluate representative architecture patterns which fit the design abstraction of the problem. This we have done using simulation studies. The extent to which this result holds in deployed systems can be argued. Kevin Lai [108] suggests that the process of system design and market mechanism should be integrated, not disjoint. This way, the software architect can align the underlying assumptions of the mechanism with those of the architecture at hand. Even at this, much is to be learned from a real deployed system, where some of the assumptions may fail or change in ways not anticipated at design time.

7.4.4 Improved Reputation-aware Posted Offer Market Mechanism

In our reputation-aware posted offer market mechanism, two representative buyer trading strategies (time savers and bargain hunters) were studied. However, the literature in Computational Economics (e.g. [161]) has recently highlighted the importance of evolutionary market agents that are able to behave more rationally than hand-crafted strategies. We suggest the use of more sophisticated buyer strategies as another direction for future work in improving our market mechanism. Lastly, an important result in the mechanism design

domain is the *impossibility result* due to Myerson and Satterthwaite [125]. The impossibility result implies that no single mechanism is dominant in a system where interacting agents are able to make bids and counter offers. This suggests that while our mechanism indicate satisfactory result within the scope of our evaluation, it is needless to say a more advanced market mechanism may dominate it. Therefore, another area for future work is to compare decentralised market mechanisms using common datasets and experimental scenarios to deduce their niches, in order to guide mechanism selection or engineer self-adaptive systems to use a combination of market mechanisms appropriately.

7.4.5 Socially-aware Adaptive Cloud Software Architecture

This thesis has taken an architectural perspective to self-adaptation, specifically, we have studied architectural artefact at the *style* level of abstraction. A complementary approach is requirement-based self-adaptation (e.g. [14][145]), where users are treated as first-class entities in the adaptation loop. As noted by [5], combining requirement-based adaptation and architectural-based adaptation provides an improved model of system's goals (requirements) and deployment variabilities (architecture).

A specialised form of requirement-based self-adaptation is Social Adaptation [3] [2] which is defined as "a system's autonomous ability to analyse users' feedback and choose an alternative behaviour which is collectively shown to be the best for meeting requirements in a context." [2]. Social Adaptation is unique in the sense that instead of catering to the requirement of a user or subset of users at run-time, it harnesses the "wisdom of the crowd" to adapt the system in a way that is deemed best by end-users' collective judgement rather than the decisions of an elite group of users or those of developers of the self-adaptive system.

Combining the self-aware architectural style with a socially-derived user feedback loop will ensure the self-adaptive systems has a collective view of users' perception of the system's goal. This is especially applicable to crowdsourced applications where a large group of users collectively solve complex problems. To achieve this objective the following

research questions need to be answered: How do we explicitly model socially-derived feedback in the self-aware style? What aggregation mechanism should be used to combine individual user's views of the system? How do we resolve conflicts between a user's (personalised) and group's (community) adaptive goals?

7.5 Closing Remarks

This thesis has presented the self-aware architecture patterns that offers a systematic and principled approach to architecting software systems that must adapt to changes in user requirements, system, and environmental conditions without external intervention. The self-aware architecture patterns achieves this by modelling knowledge concerns at fine-grained levels, thereby promoting improved analysis of trade-offs and risk points in the architecture. Additionally, we contributed a market mechanism for coordinating decentralised components of federated cloud applications.

The author hopes the findings presented in this thesis improves our understanding of architecting self-adaptive systems that copes with dynamic, large scale systems such as cloud. As identified in roadmaps [42] [111] many open problems remain to be solved before the vision of truly robust, scalable, and reliable self-adaptive systems comes to reality. The author hopes the work presented in this thesis will pave way for future research work that solves some of these problems and moves the discipline forward.

LIST OF REFERENCES

[1] Nadeem Abbas, Jesper Andersson, and Welf Löwe. Autonomic software product lines (ASPL). In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 324–331, New York, NY, USA, 2010. ACM.

[2] Raian Ali, Carlos Solís, Inah Omoronyia, Mazeiar Salehie, and Bashar Nuseibeh. Social adaptation - when software gives users a voice. In Joaquim Filipe and Leszek A. Maciaszek, editors, *ENASE*, pages 75–84. SciTePress, 2012.

[3] Malik Almaliki, Funmilade Faniyi, Rami Bahsoon, Keith Phalp, and Raian Ali. Requirements-driven social adaptation: Expert survey. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, volume 8396 of *Lecture Notes in Computer Science*, pages 72–87. Springer International Publishing, 2014.

[4] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). In *Global Grid Forum*, volume 2, 2004.

[5] Konstantinos Angelopoulos, Vítor E. Silva Souza, and João Pimentel. Requirements and architectural approaches to adaptive software systems: a comparative study. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 23–32, Piscataway, NJ, USA, 2013. IEEE Press.

[6] Danilo Ardagna, Barbara Panicucci, and Mauro Passacantando. A game theoretic formulation of the service provisioning problem in cloud systems. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 177–186, New York, NY, USA, 2011. ACM.

[7] Danilo Ardagna and Barbara Pernici. Global and local qos constraints guarantee in web service selection. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Electrical Engineering and Computer Sciences University of California at Berkeley, 2009. Available Online at http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html [Last Accessed: 28-Sep-2014].

[10] Muhammad Ali Babar and Ian Gorton. Comparison of scenario-based software architecture evaluation methods. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 600–607, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Rami Bahsoon. A framework for dynamic self-optimization of power and dependability requirements in green cloud architectures. In *Software Architecture*, pages 510–514. Springer, 2010.

[12] Adam Barker, Blesson Varghese, Jonathan Stuart Ward, and Ian Sommerville. Academic cloud computing research: Five pitfalls and five opportunities. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014. USENIX Association.

[13] T. Becker, A. Agne, P.R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A.R. Jensenius, and S.C. Stilkerich. EPiCS: Engineering proprioception in computing systems. In *Proc. of the 15th IEEE International Conf. on Computational Science and Engineering (CSE)*, pages 353–360, 2012.

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

[15] N. Bonvin, T.G. Papaioannou, and K. Aberer. Autonomic sla-driven provisioning for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 434–443, 2011.

[16] Nelis Boucké, Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Applying the atam to an architecture for decentralized control of a transportation system. In *Quality of Software Architectures*, pages 180–198. Springer, 2006.

[17] J. Bouman, J. Trienekens, and M. van der Zwan. Specification of service level agreements, clarifying concepts on the basis of practical research. In *Software Technology and Engineering Practice, 1999. STEP '99. Proceedings*, pages 169–178, 1999.

[18] Ivona Brandic, Vincent C. Emeakaroha, Michael Maurer, Schahram Dustdar, Sandor Acs, Attila Kertesz, and Gabor Kecskemeti. Laysi: A layered approach for sla-violation propagation in self-manageable cloud infrastructures. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, COMPSACW '10, pages 365–370, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Ivona Brandic, Dejan Music, Philipp Leitner, and Schahram Dustdar. Vieslaf framework: Enabling adaptive and versatile sla-management. In *Grid Economics and Business Models*, volume 5745 of *Lecture Notes in Computer Science*, pages 60–73. Springer Berlin / Heidelberg, 2009.

[20] J. Branke, M. Mnif, C. Muller-Schloer, and H. Prothmann. Organic computing; addressing complexity by controlled self-organization. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 185–191, 2006.

[21] Sonja Buchegger and Jean yves Le Boudec. A robust reputation system for mobile ad-hoc. Technical report, EPFL-IC-LCA; CH-1015 Lausanne, Switzerland, Nov 2003.

[22] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing sla management based upon business objectives. *IBM Systems Journal*, 43(1):159 –178, 2004.

[23] Frank Buschmann, Kevlin Henney, and Schmidt C. Douglas. *Pattern-oriented software architecture: On patterns and pattern languages*. John Wiley and Sons, 2007.

[24] R. Buyya, S.K. Garg, and R.N. Calheiros. Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 1–10, Dec 2011.

[25] R. Buyya, R. Ranjan, and R.N. Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 1 –11, June 2009.

[26] Rajkumar Buyya. Market-oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing as the 5th Utility. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 1, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[28] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[29] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25:599–616, June 2009.

[30] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[31] V. Cardellini, E. Casalicchio, F. Lo Presti, and L. Silvestri. Sla-aware resource management for application service providers in the cloud. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 20–27, Nov 2011.

[32] Hugo E. T. Carvalho and Otto Carlos M. B. Duarte. Voltaic: volume optimization layer to assign cloud resources. In *Proceedings of the 3rd International Conference on Information and Communication Systems*, ICICS '12, pages 3:1–3:7, New York, NY, USA, 2012. ACM.

[33] E. Casalicchio and L. Silvestri. An inter-cloud outsourcing model to scale performance, availability and security. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 151–158, 2012.

[34] Timothy N. Cason, Daniel Friedman, and Garrett H. Milam. Bargaining versus posted price competition in customer markets. *International Journal of Industrial Organization*, 21(2):223 – 251, 2003.

[35] Arjun Chandra, Kristian Nymoen, Arve Voldsund, AlexanderRefsum Jensenius, Kyrre Glette, and Jim Torresen. Market-based control in interactive music environments. In Mitsuko Aramaki, Mathieu Barthet, Richard Kronland-Martinet, and S?lvi Ystad, editors, *From Sounds to Music and Emotions*, volume 7900 of *Lecture Notes in Computer Science*, pages 439–458. Springer Berlin Heidelberg, 2013.

[36] Arjun Chandra, Kristian Nymoen, Arve Volsund, Alexander Refsum Jensenius, Kyrre Glette, and Jim Torresen. Enabling participants to play rhythmic solos within a group via auctions. In *Proc. Int. Symp. on Computer Music Modeling and Retrieval (CMMR)*, pages 674–689, jun 2012.

[37] Anthony Chavez, Alexandros Moukas, and Pattie Maes. Challenger: a multi-agent system for distributed resource allocation. In *Proceedings of the first international conference on Autonomous agents*, AGENTS '97, pages 323–331, New York, NY, USA, 1997. ACM.

[38] A. Chazalet. Service level checking in the cloud computing context. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 297 – 304, July 2010.

[39] Tao Chen and Rami Bahsoon. Symbiotic and sensitivity-aware architecture for globally-optimal benefit in self-adaptive cloud. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 85–94, New York, NY, USA, 2014. ACM.

[40] Tao Chen, Funmilade Faniyi, Rami Bahsoon, Peter R. Lewis, Xin Yao, Leandro L. Minku, and Lukas Esterle. The handbook of engineering self-aware and self-expressive systems. Technical report, EPiCS EU FP7 project consortium, August 2014. Available online at http://arxiv.org/abs/1409.1793 [Last Accessed: 28-Sep-2014].

[41] Yee-Ming Chen and Hsin-Mei Yeh. Autonomous adaptive agents for market-based resource allocation of cloud computing. In *Machine Learning and Cybernetics (ICMLC), 2010 International Conference on*, volume 6, pages 2760 –2764, July 2010.

[42] Betty Cheng, Rogrio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In Betty Cheng, Rogrio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2009.

[43] Shang-Wen Cheng. *Rainbow: cost-effective software architecture-based self-adaptation*. PhD thesis, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, 2008.

[44] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 132–141. IEEE, 2009.

[45] Shang-Wen Cheng, An-Cheng Huang, D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. In *Autonomic Computing, 2004. Proc. Int. Conf. on*, pages 276–277, 2004.

[46] P. Ciancarini and M. Wooldridge. Agent-oriented software engineering. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 816–817, 2000.

[47] Scott H. Clearwater, editor. *Market-based control: a paradigm for distributed resource allocation*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[48] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[49] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[50] D. Cliff and J. Bruten. Minimal-intelligence agents for bargaining behaviors in market-based environments. Technical report, HP LABORATORIES, 1997.

[51] Dave Cliff. Biologically-inspired computing approaches to cognitive systems: a partial tour of the literature. Technical report, Hewlett-Packard Company, 2003.

[52] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Systems Journal*, 43(1):136 –158, 2004.

[53] Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *Computational Science-ICCS 2004*, pages 662–669. Springer, 2004.

[54] Frederica Darema. Dynamic data driven applications systems: New capabilities for application simulations and measurements. In *Computational Science–ICCS 2005*, pages 610–615. Springer, 2005.

[55] Rajdeep K. Dash, Sarvapali D. Ramchurn, and Nicholas R. Jennings. Trust-based mechanism design. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '04, pages 748–755, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Google Cluster Data. http://googleresearch.blogspot.com/2010/01/google-cluster-data.html [Last Accessed: 28-Sep-2014].

[57] Rogrio de Lemos, Holger Giese, Hausi Mller A., and Mary Shaw, editors. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap (Draft Version of May 20, 2011)*, Dagstuhl Seminar Proceedings 10431, 2010.

[58] Scott A DeLoach, Valeriy A Kolesnikov, et al. Using design metrics for predicting system flexibility. In *Fundamental Approaches to Software Engineering*, pages 184–198. Springer, 2006.

[59] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[60] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[61] Craig C Douglas. Dynamic data driven applications systems–dddas 2008. In *Computational Science–ICCS 2008*, pages 3–4. Springer, 2008.

[62] Schahram Dustdar, Christoph Dorn, Fei Li, Luciano Baresi, Giacomo Cabri, Cesare Pautasso, and Franco Zambonelli. A roadmap towards sustainable self-aware service systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 10–19. ACM, 2010.

[63] X. Dutreilh, N. Rivierre, A Moreau, J. Malenfant, and I Truck. From data center resource allocation to control theory and back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 410–417, July 2010.

[64] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[65] N. Elprince. Autonomous resource provision in virtual data centers. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 1365–1371, May 2013.

[66] EPiCS. Engineering proprioception in computing systems (EPiCS). http://www.epics-project.eu/ [Last Accessed: 28-Sep-2014].

[67] Lukas Esterle, Peter R. Lewis, Xin Yao, and Bernhard Rinner. Socio-economic vision graph generation and handover in distributed smart camera networks. *ACM Trans. Sen. Netw.*, 10(2):20:1–20:24, January 2014.

[68] F. Faniyi, P.R. Lewis, R. Bahsoon, and Xin Yao. Architecting self-aware software systems. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 91–94, April 2014.

[69] Funmilade Faniyi and Rami Bahsoon. Engineering Proprioception in SLA Management for Cloud Architectures. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture, WICSA*, June 2011.

[70] Funmilade Faniyi and Rami Bahsoon. Self-managing sla compliance in cloud architectures: a market-based approach. In *Proc. of the 3rd Int. ACM SIGSOFT symposium on Architecting Critical Systems*, ISARCS'12, pages 61–70, 2012.

[71] Funmilade Faniyi and Rami Bahsoon. Economics-driven software architecting for cloud. In Ivan Mistrik, Rami Bahsoon, Rick Kazman, and Yuanyuan Zhang, editors, *Economics-Driven Software Architecture*, pages 83 – 103. Morgan Kaufmann, Boston, 2014.

[72] Funmilade Faniyi, Rami Bahsoon, Andy Evans, and Rick Kazman. Evaluating Security Properties of Architectures in Unpredictable Environments: A Case for Cloud. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture, WICSA*, June 2011.

[73] Funmilade Faniyi, Rami Bahsoon, and Georgios Theodoropoulos. A dynamic data-driven simulation approach for preventing service level agreement violations in cloud federation. *Procedia Computer Science*, 9:1167–1176, 2012.

[74] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Ltsa-ws: A tool for model-based verification of web service compositions and choreography. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 771–774, New York, NY, USA, 2006. ACM.

[75] Ian Foster, Yong Zhao, Ioan Raicu, and hiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *GCE: 2008 Grid Computing Environments Workshop*, pages 60–69, 2008.

[76] Ikki Fujiwara, Kento Aida, and Isao Ono. Applying double-sided combinational auctions to resource allocation in cloud computing. In *Proceedings of the 2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, SAINT '10, pages 7–14, Washington, DC, USA, 2010. IEEE Computer Society.

[77] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 159–173, Riverton, NJ, USA, 2010. IBM Corp.

[78] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[79] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21, Carnegie Mellon University Software Engineering Institute, January 1994.

[80] Erann Gat. On three-layer architectures. In David Kortenkamp, R. Peter Bonnasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997.

[81] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 33–38, New York, NY, USA, 2002. ACM.

[82] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Morgan Kaufmann, 2004.

[83] D. Gil, J. Andersson, M. Milrad, and H. Sollervall. Towards a decentralized and self-adaptive system for m-learning applications. In *Wireless, Mobile and Ubiquitous Technology in Education (WMUTE), 2012 IEEE Seventh International Conference on*, pages 162–166, 2012.

[84] Carlos Mera Gmez. Simulation tool for market-based cloud resource allocation. Master's thesis, School of Computer Science, The University of Birmingham, UK, September 2011.

[85] M. Hamze, N. Mbarek, and O. Togni. Autonomic brokerage service for an end-to-end cloud networking service level agreement. In *Network Cloud Computing and Applications (NCCA), 2014 IEEE 3rd Symposium on*, pages 54–61, Feb 2014.

[86] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A framework for self-aware computing. Technical Report MIT-CSAIL-TR-2010-049, Massachusetts Institute of Technology, Cambridge, October 2010.

[87] Ye Huang, Nik Bessis, Peter Norrington, Pierre Kuonen, and Beat Hirsbrunner. Exploring decentralized dynamic scheduling for grids and clouds using the community-aware scheduling algorithm. *Future Gener. Comput. Syst.*, 29(1):402–415, January 2013.

[88] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 90–99, New York, NY, USA, 2011. ACM.

[89] Markus C Huebscher and Julie A McCann. A survey of autonomic computingdegrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[90] F. Hussain, E. Chang, and O. Hussain. A robust methodology for prediction of trust and reputation values. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 97–108, 2008.

[91] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 95–104, New York, NY, USA, 2014. ACM.

[92] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 471–480, New York, NY, USA, 2010. ACM.

[93] Joanna Juziuk, Danny Weyns, and Tom Holvoet. Design patterns for multi-agent systems: A systematic literature review. In Onn Shehory and Arnon Sturm, editors, *Agent-Oriented Software Engineering*, pages 79–99. Springer Berlin Heidelberg, 2014.

[94] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. Criteria for the evaluation of self-* systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 29–38, New York, NY, USA, 2010. ACM.

[95] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 68–78, Monterey, CA, 1998. IEEE Computer Society.

[96] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, and Steven G. Woods. Experience with performing architecture tradeoff analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 54–63, New York, NY, USA, 1999. ACM.

[97] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical report, Carnegie Mellon University, Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213, August 2000.

[98] Ariane Keller, Stephan Neuhaus, Markus Happe, Daniel Borkmann, and Red Hat. Autonomic configuration of dynamic protocol stacks. In *2 nd Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS13)*, page 3, 2013.

[99] Jeffrey O Kephart. Learning from nature. *Science*, 331(6018):682–683, 2011.

[100] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[101] Jon Ketcham, Vernon L Smith, and Arlington W Williams. A comparison of posted-offer and double-auction pricing institutions. *The Review of Economic Studies*, 51(4):595614, 1984.

[102] B.A. Kitchenham, Tore Dyba, and M. Jorgensen. Evidence-based software engineering. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 273–281, 2004.

[103] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33:2004, 2004.

[104] Paul Klemperer. Auction theory: A guide to the literature. *Journal of Economic Surveys*, 13(3):227–286, 1999.

[105] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[106] Jeff Kramer and Jeff Magee. A rigorous architectural approach to adaptive software engineering. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, 24(2):183–188, Mar 2009.

[107] Kazuhiro Kuwabara, Toru Ishida, Yoshiyasu Nishibe, and Tatsuya Suda. An equilibratory market-based approach for distributed resource allocation and its applications to communication network control. In *Market-based control: A paradigm for distributed resource allocation*, pages 53 –73. World Scientific, Singapore, 1996.

[108] Kevin Lai. Markets are dead, long live markets. *SIGecom Exch.*, 5:1–10, July 2005.

[109] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, 2008.

[110] Young Choon Lee, Chen Wang, Albert Y. Zomaya, and Bing Bing Zhou. Profit-driven service request scheduling in clouds. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 15–24, Washington, DC, USA, 2010. IEEE Computer Society.

[111] Rogrio et al. Lemos. Software engineering for self-adaptive systems: A second research roadmap. In Rogrio Lemos, Holger Giese, HausiA. Mller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2013.

[112] L. Lewis and P. Ray. Service level management definition, architecture, and research challenges. In *Global Telecommunications Conference, 1999. GLOBECOM '99*, volume 3, pages 1974–1978 vol.3, 1999.

[113] Lundy Lewis. *Service Level Management for Enterprise Networks*. Artech House, Inc., Norwood, MA, USA, 1st edition, 1999.

[114] Peter Lewis, Paul Marrow, and Xin Yao. Resource allocation in decentralised computational systems: an evolutionary market-based approach. *Autonomous Agents and Multi-Agent Systems*, 21:143–171, 2010. 10.1007/s10458-009-9113-x.

[115] Peter R. Lewis, Arjun Chandra, Shaun Parsons, Edward Robinson, Kyrre Glette, Rami Bahsoon, Jim Torresen, and Xin Yao. A survey of self-awareness and its application in computing systems. In *Proc. Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 102–107, 2011.

[116] Peter R. Lewis, Funmilade Faniyi, Rami Bahsoon, and Xin Yao. Markets and clouds: Adaptive and resilient computational resource allocation inspired by economics. In Niranjan Suri and Giacomo Cabri, editors, *Adaptive, Dynamic, and Resilient Systems*, pages 285–318. Taylor & Francis, 2013.

[117] Qianhui Liang and Bu-Sung Lee. Delivering high resilience in designing platform-as-a-service clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 676–683, 2011.

[118] Qi Liu, Dilma Da Silva, Georgios K. Theodoropoulos, and Elvis S. Liu. Towards an agent-based symbiotic architecture for autonomic management of virtualized data centers. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 147:1–147:13. Winter Simulation Conference, 2012.

[119] P Marrow. Nature-inspired computing technology and applications. *BT Technology Journal*, 18(4):13–23, 2000.

[120] Pedro Martins and Julie A. McCann. ajme: making game engines autonomic. In *Proceedings of the 3rd International Conference on Fun and Games*, Fun and Games '10, pages 48–57, New York, NY, USA, 2010. ACM.

[121] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Adaptive resource configuration for cloud infrastructure management. *Future Generation Computer Systems*, 29(2):472 – 487, 2013. Special section: Recent advances in e-Science.

[122] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011.

[123] Daniel A. Menasce, João P. Sousa, Sam Malek, and Hassan Gomaa. Qos architectural patterns for self-architecting software systems. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 195–204, New York, NY, USA, 2010. ACM.

[124] C Muller-Schloer. Organic computing-on the feasibility of controlled emergence. In *Hardware/Software Codesign and System Synthesis, 2004. CODES+ ISSS 2004. International Conference on*, pages 2–5. IEEE, 2004.

[125] Roger B Myerson and Mark A Satterthwaite. Efficient mechanisms for bilateral trading. *Journal of economic theory*, 29(2):265–281, 1983.

[126] Vivek Nallur and Rami Bahsoon. Design of a market-based mechanism for quality attribute tradeoff of services in the cloud. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 367–371, New York, NY, USA, 2010. ACM.

[127] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[128] Olufunmilola O. Onolaja. *Dynamic Data-Driven Framework for Reputation Management*. PhD thesis, The University of Birmingham, October 2012.

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

[130] Ying Ouyang, Jia En Zhang, and Shi Ming Luo. Dynamic data driven application system: Recent development and future perspective. *ecological modelling*, 204(1):1–8, 2007.

[131] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Commun. ACM*, 46:24–28, October 2003.

[132] T. Patikirikorala, A. Colman, J. Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 33–42, 2012.

[133] Diego Perez-Palacin, Raffaela Mirandola, and Jos Merseguer. On the relationships between qos and software adaptability at the architectural level. *Journal of Systems and Software*, 87(0):1 – 17, 2014.

[134] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct 1992.

[135] A.G.P. Rahbar and O. Yang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(4):460 –473, April 2007.

[136] C. Raibulet and L. Masciadri. Evaluation of dynamic adaptivity through metrics: an achievable target? In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 341–344, 2009.

[137] Sarvapali D. Ramchurn, Claudio Mezzetti, Andrea Giovannucci, Juan A. Rodriguez-Aguilar, Rajdeep K. Dash, and Nicholas R. Jennings. Trust-based mechanisms for robust and efficient task allocation in the presence of execution uncertainty. *J. Artif. Int. Res.*, 35:119–159, June 2009.

[138] R. Ranjan, A. Harwood, and R. Buyya. Sla-based coordinated superscheduling scheme for computational grids. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1 –8, Sept. 2006.

[139] Samuel T. Redwine, Jr. and William E. Riddle. Software technology maturation. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, pages 189–200, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[140] Willi Richert and Bernd Kleinjohann. Adaptivity at every layer: a modular approach for evolving societies of learning autonomous systems. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 113–120, New York, NY, USA, 2008. ACM.

[141] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4:1 –4:11, july 2009.

[142] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, 3 edition, 2010.

[143] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[144] Tuomas W. Sandholm. *Distributed rational decision making*, pages 201–258. MIT Press, Cambridge, MA, USA, 1999.

[145] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 95–103, 2010.

[146] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. Adaptivity and self-organization in organic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 5(3):10:1–10:32, September 2010.

[147] Shifeng Shang, Jinlei Jiang, Yongwei Wu, Guangwen Yang, and Weimin Zheng. A knowledge-based continuous double auction model for cloud market. In *Proceedings of the 2010 Sixth International Conference on Semantics, Knowledge and Grids*, SKG '10, pages 129–134, Washington, DC, USA, 2010. IEEE Computer Society.

[148] Mary Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 656–664. IEEE Computer Society, 2001.

[149] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[150] Weiming Shi and Bo Hong. Resource allocation with a budget constraint for computing independent tasks in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 327 –334, 30 2010-dec. 3 2010.

[151] Moshe Sipper. *Machine nature: The coming age of bio-inspired computing*, volume 11. McGraw-Hill New York, 2002.

[152] Biao Song, M. M. Hassan, and Eui-nam Huh. A novel heuristic-based task selection and allocation framework in dynamic collaborative cloud service platform. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 360–367, Washington, DC, USA, 2010. IEEE Computer Society.

[153] Nary Subramanian and Lawrence Chung. Metrics for software adaptability. *Proc. Software Quality Management (SQM 2001), April*, 2001.

[154] Dawei Sun, Guiran Chang, Chuan Wang, Yu Xiong, and Xingwei Wang. Efficient nash equilibrium based cloud resource allocation by using a continuous double auction. In *Computer Design and Applications (ICCDA), 2010 International Conference on*, volume 1, pages V1–94 –V1–99, June 2010.

[155] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed architectural change for autonomous systems. In *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 15–21, New York, NY, USA, 2007. ACM.

[156] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 1–8, New York, NY, USA, 2008. ACM.

[157] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 431–438, New York, NY, USA, 2010. ACM.

[158] Daniel Sykes, Jeff Magee, and Jeff Kramer. Flashmob: distributed adaptive self-assembly. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109, New York, NY, USA, 2011. ACM.

[159] Saltanat Tangbayeva. Maintaining optimal quality of service in cloud applications using market-based techniques. Master's thesis, School of Computer Science, The University of Birmingham, UK, September 2012.

[160] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009.

[161] Leigh Tesfatsion and Kenneth L Judd. *Handbook of computational economics: agent-based computational economics*, volume 2. Elsevier, 2006.

[162] W.F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[163] Sebastian Uchitel, Robert Chatley, Jeff Kramer, and Jeff Magee. Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 597–601. Springer Berlin Heidelberg, 2003.

[164] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009.

[165] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM.

[166] Xiaoying Wang, Zhihui Du, and Yinong Chen. An adaptive model-free resource and power management approach for multi-tier cloud environments. *Journal of Systems and Software*, 85(5):1135 – 1146, 2012.

[167] Danny Weyns, Sam Malek, and Jesper Andersson. On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 84–93, New York, NY, USA, 2010. ACM.

[168] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and KarlM. Gschka. On patterns for decentralized control in self-adaptive systems. In Rogrio Lemos, Holger Giese, HausiA. Mller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer Berlin Heidelberg, 2013.

[169] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pages 3:27–3:41, New York, NY, USA, 2008. ACM.

[170] Fred E. Williams. The effect of market organization on competitive equilibrium: the multi-unit case. *The Review of Economic Studies*, 40(1):97–113, 1973.

[171] Rich Wolski, James S. Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational grid. *Int. J. High Perform. Comput. Appl.*, 15:258–281, August 2001.

[172] Micaela Wuensche, Sanaz Mostaghim, Hartmut Schmeck, Timo Kautzmann, and Marcus Geimer. Organic computing in off-highway machines. In *Proceedings of the second international workshop on Self-organizing architectures*, SOAR '10, pages 51–58, New York, NY, USA, 2010. ACM.

[173] Lijuan Xiao, Yanmin Zhu, L.M. Ni, and Zhiwei Xu. Gridis: An incentive-based grid scheduling. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 65b, april 2005.

[174] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 87–98, 2011.

[175] Pengcheng Xiong, Zhikui Wang, S. Malkowski, Qingyang Wang, D. Jayasinghe, and C. Pu. Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 571–580, 2011.

[176] Yagiz Onat Yazir, Chris Matthews, Roozbeh Farahbod, Stephen Neville, Adel Guitouni, Sudhakar Ganti, and Yvonne Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 91–98, Washington, DC, USA, 2010. IEEE Computer Society.

[177] Xindong You, Xianghua Xu, Jian Wan, and Dongjin Yu. Ras-m: Resource allocation strategy based on market mechanism in cloud computing. In *ChinaGrid Annual Conference, 2009. ChinaGrid '09. Fourth*, pages 256 –263, Aug. 2009.

[178] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, pages 108–113. IEEE, 2011.

[179] Ying Zhang, Gang Huang, Xuanzhe Liu, and Hong Mei. Integrating resource consumption and allocation for infrastructure resources on-demand. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 75–82, July 2010.

[180] Jie Zhu, Bo Gao, Zhihu Wang, B. Reinwald, ChangJie Guo, Xiaoping Li, and Wei Sun. A dynamic resource allocation algorithm for database-as-a-service. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 564–571, July 2011.

Appendices

APPENDIX A

LIST OF PUBLICATIONS

The work presented in this thesis is based on and extends the followings papers that have been published during the course of the PhD programme.

1. "Architecting Self-aware Software Systems", F. Faniyi, P. R. Lewis, R. Bahsoon, X. Yao, in 11th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2014.

2. "Engineering Proprioception in SLA Management for Cloud Architectures", F. Faniyi, R. Bahsoon, in Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011.

3. "EPiCS: Engineering Proprioception in Computing Systems", T. Becker, A. Agne, P. R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. R. Jensenius, S. C. Stilkerich, in Proceedings of the 10th IEEE/IFIP Conference on Embedded and Ubiquitous Computing, 2012.

4. "Self-Managing SLA Compliance in Cloud Architectures: A Market-based Approach", F. Faniyi, R. Bahsoon, in the Proceedings of the International ACM Sigsoft Symposium on Architecting Critical Systems, ISARCS, 2012.

5. "A Dynamic Data-Driven Simulation Approach for Preventing SLA Violations in Cloud Federation", F. Faniyi, R. Bahsoon, G. Theodoropoulos, in the Procedia:

Procedia Computer Science, Proceedings of the International Conference on Computational Science, ICCS, 2012.

6. "Economics-driven Software Architecting for Cloud", F. Faniyi, R. Bahsoon, in Economics-driven Software Architecture, 2013, Elsevier.

7. "Markets and Clouds: Adaptive and Resilient Computational Resource Allocation inspired by Economics", P. R. Lewis, F. Faniyi, R. Bahsoon, X. Yao, in Niranjan Suri and Giacomo Cabri (Eds.), Adaptive, Dynamic, and Resilient Systems, 2013, Taylor & Francis.

8. "Evaluating Security Properties of Architectures in Unpredictable Environments: A Case for Cloud", F. Faniyi, R. Bahsoon, A. Evans, R. Kazman, in Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011.

9. "Architectural Aspects of Self-aware and Self-expressive Computing Systems", P. R. Lewis, A. Chandra, F. Faniyi, K. Glette, T. Chen, R. Bahsoon, J. Torresen, X. Yao in IEEE Computer, 2015 (*to appear*).

10. "Requirements-driven Social Adaptation: Expert Survey", M. Almaliki, F. Faniyi, R. Bahsoon, K. Phalp, R. Ali, in the 20th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), 2014.

11. "Using Obstacles for Systematically Modelling, Analysing and Mitigating Risks in Cloud Adoption", S. Zardari, F. Faniyi, R. Bahsoon, in the book on Aligning Enterprise, System and Software Architectures, 2012. IGI Global.

This thesis should be regarded as the definitive account of the work.

APPENDIX B

SYSTEMATIC REVIEW PROCESS

A systematic review of SLA-based resource allocation was carried out to find out about the strengths and weaknesses of state of the art. This section documents the review process. The research questions that triggered the review are presented in chapter 2.

B.1 Acronyms and Meaning

In this section, we define some acronyms used in the discussion of the SLR.

- Infrastructure as a Service (IaaS): In this model, cloud users are provided virtual machine instances, which are configured to meet user's preferences. Amazon Elastic Compute Cloud (EC2) is a notable example.
- Software as a Service (SaaS): Cloud providers manage application and data on behalf of users in this model. Notable examples are Google Docs and Salesforce CRM.
- Platform as a Service (PaaS): Here, a platform is provided for users to develop and deploy applications using specialised APIs, while providers manage scalability and load-balancing. Google AppEngine and Microsoft Azure are PaaS providers.
- Data as a Service (DaaS): Cloud users are provided a service for creating, storing, and accessing their database using this model.

- Network as a Service (NaaS): Cloud users are able to configure low-level network protocols, packets, and routing, as a way of improving caching and data aggregation.
- Storage as a Service (StaaS): Cloud users are provided service for storing data (structured or unstructured) in third-party cloud systems. The burden of backup and recovery of data is managed by the provider (e.g. DropBox and Amazon S3).
- Cloud Federation: In this model multiple cloud providers are used to flexibly respond to variations in workload and prevent a single point of failure. It addresses the limited scalability of single cloud providers and lack of interoperability among them.

B.2 Review Protocol

The review was conducted using the systematic literature review (SLR) methodology proposed by Kitchenham et al. [102]. The protocol consists of the following steps: selection of indexing services, definition of search query, inclusion, and exclusion criteria.

B.3 Inclusion and Exclusion Criteria

B.3.1 Inclusion Criteria

The following document types were included: journal, conference, technical reports, and workshop papers. Also, we consider papers that claim SLM solutions applicable to both grid and cloud computing. The rationale for former studies is because we aim to understand the resource allocation methods used not necessarily the application context.

B.3.2 Exclusion Criteria

The following category of studies were excluded from our initial search.

- **E1:** Abstracts of keynotes; Title pages of conference proceedings.

- **E2:** Duplicate entries. For examples, papers with same concepts and results appearing in multiple venues. Also, longer version (e.g. journal) of papers that previously appeared in workshop or conference proceedings are preferred to shorter versions.

- **E3:** Papers that do not consider SLAs at cloud adoption or deployment phases.

- **E4:** Papers that were published before year 2008. This is because cloud computing was formally incepted in 2008 [164][9]. Our search for the term "Cloud computing" revealed that papers returned before 2008 were are not related to the topic as conceived in this thesis.

We further categorised papers as "relevant" or "not relevant". This distinction is crucial to the review, since we do not want to be too stringent such that we miss out papers that are related to our research. On the other hand, we do not want to include closely related work that focus on a different research theme from ours.

- **E5: Cloud adoption papers.** Our research problem is not about deciding whether to use cloud computing or not. We already assume that cloud is the studied deployment environment. Therefore, we do not include papers focusing on cloud adoption.

- **E6: Papers that do not use SLAs for architecture adaptation.** Since the research problem of this thesis is about investigating approaches to architecting self-adaptive cloud architectures. Papers that capture cloud SLAs but do not use them in actual resource provisioning are not relevant to our problem.

- **E7: Cloud technology-specific papers.** Papers that report improvement to cloud specific technologies (e.g. Amazon Web Services and Windows Azure) in aspects outside SLA-based resource management are not included

- **E8: Cloud experimentation papers.** Similarly, researches that mainly use the cloud for experimental purposes that fall outside the scope of SLA-based resource allocation are excluded.

B.4 Search Process

B.4.1 Choice of Indexing Service

The following indexing services were used for the review process, since they are known to cover a broad range of Computer Science topics.

- IEEE Xplore (http://ieeexplore.ieee.org/)
- ACM Digital Library (http://portal.acm.org)
- Science Direct (http://www.sciencedirect.com/)
- ISI Web of Knowledge (http://apps.webofknowledge.com/)
- Engineering Village, which also indexes INSPEC and COMPENDEX (http://www.engineeringvillage.com)
- Google Scholar (http://scholar.google.co.uk/)

The search date was bounded between 2008 and 2013 (inclusive). Section 2.3.3 (recency of findings) details our effort to assert the validity of the review's findings for papers published after the search date.

B.4.2 Search Query

SLA-based resource allocation in cloud computing is saturated with many methods, also contributions may focus on one or more phases of the SLA life cycle. To accommodate the largest possible pool of papers we opted for an open ended search term, which was (‘‘Cloud computing’’ AND ‘‘SLA’’ AND ‘‘Resource allocation’’).

B.4.3 Search Result

Table B.1 shows the result of the initial search from all chosen indexing services. Given the limited time, we were unable to review all papers from the initial result set. A pragmatic

approach of focusing on the top 100 papers, when filtered by relevance, from each indexing service was adopted. Table B.2 shows the result after applying the exclusion criteria.

Indexing Service	**Start Year**	**End Year**	**# of Papers**
IEEE Xplore	2008	2013	105
ACM Digital Library	2008	2013	165
Science Direct	2009	2013	121
ISI Web of Knowledge	2009	2013	4
Engineering Village	2009	2013	207
Google Scholar	2009	2013	2990
Total			**3592**

Table B.1: Search Result (Initial)

Indexing Service	**Start Year**	**End Year**	**# of Papers**
IEEE Xplore	2008	2013	49
ACM Digital Library	2008	2013	11
Science Direct	2009	2013	16
ISI Web of Knowledge	2009	2013	10
Engineering Village	2009	2013	6
Google Scholar	2009	2013	14
Total			**106**

Table B.2: Search Result (After Applying Exclusion Criteria). Section 2.3.3 details our effort to assert the validity of the review's findings for papers published after 2013.

APPENDIX C

ARCHITECTURAL ANALYSIS FORM

Architecture style	
Summarise the architectural solution / design rationale	
What are the identified quality attribute priorities?	
What are the supported quality attribute scenarios?	
What are the identified risks and their implication?	
What are the identified sensitivity points and their implication?	
What are the identified trade-off points and their implication?	
Additional comments	.