The Handbook of Engineering Self-Aware and Self-Expressive Systems

Tao Chen[1], Funmilade Faniyi[1], Rami Bahsoon[1], Peter R. Lewis[2],
Xin Yao[1], Leandro L. Minku[1], and Lukas Esterle[3]

[1]University of Birmigham, UK
[2]Aston University, Birmigham, UK
[3]Alpen-Adria Universität Klagenfurt, Austria

5 September 2014

Acknowledgement

This work was partially supported by the European Union Seventh Framework Programme under grant agreement 257906 (EPiCS).

ePICS
Engineering Proprioception
in Computing Systems

Contents

Chapter 1 Patterns for Self-aware Architecture Style 4
1.1 Definition of Self-awareness 6
1.1.1 Private and Public Self-awareness 6
1.1.2 Levels of Self-awareness 7
1.2 Definition of Self-expression 8
1.3 Basic Pattern 9
1.4 Basic Information Sharing Pattern 10
1.5 Coordinated Decision-making Pattern 12
1.6 Temporal Knowledge Sharing Pattern 14
1.7 Temporal Knowledge Aware Pattern 15
1.8 Goal Sharing Pattern 17
1.9 Temporal Goal Aware Pattern 20
1.10 Meta-self-awareness and Self-aware Patterns 22
Chapter 2 Architectural Primitives for Self-aware Systems 24
2.1 Architectural Primitives and Candidate Techniques 25
2.1.1 Capability 26
2.1.2 Behaviour 26
2.1.3 Interaction 26
2.1.4 Topology 26
2.2 The Dependency 30
Chapter 3 Pattern Driven Methodology for Engineering Self-aware and Self-expressive Systems 31
3.1 The Methodology Overview 31
3.1.1 Step 1 - Collect Requirements and Constraints 32
3.1.2 Step 2 - Propose Candidate Architecture 33
3.1.3 Step 3 - Select the Best Pattern(s) 33
3.1.4 Step 4 - Fit the Selected Pattern(s) 35
3.1.5 Step 5 - Determine the Important Primitives and the Possible Alternatives for Non-functional Requirements 35
3.1.6 Step 6 - Create Scenarios 36
3.1.7 Step 7 - Score the Alternative of Primitives Against each Non-functional Attribute using Analytical or Simulation Models 36
3.1.8 Step 8 - Find the Best Alternatives for the Final Architecture View 38

3.2 Qualitative and Quantitative Evaluation 38
  3.2.1 Cloud Autoscaling Case Study 38
  3.2.2 Smart Camera Networks Case Study 65

Chapter 1

Patterns for Self-aware Architecture Style

During previous Task 2.1, we have developed the notions of self-expression and different levels of computational self-awareness, inspired by corresponding psychological levels. In the context of architecture, we refer to the self-expression and different levels of computational self-awareness as **capability** of the systems to obtain and react upon certain knowledge. In this report, we study the categorisation of different capabilities from the architecture perspective; this could create the possibility to ensure that, when designing self-aware systems, only relevant capabilities are included, and their inclusion justified by identified benefits. There is no need for a system to become unnecessarily complex, learning and maintaining capabilities which do nothing to advance the outcomes for that system, generating only overhead. We have codified the knowledge about how to architecture self-aware applications in the form of architecture patterns, each contains different capabilities. In this task, an architecture pattern refers to an architectural problem-solution pair using the capabilities in a given context. We have elicited some patterns, where each pattern is decentralised by design. That is, structurally our self-aware patterns resemble a peer-to-peer network of interconnecting self-aware nodes, varying only in the number of the capabilities and the type of interconnection between them.

Until recently, architecture patterns for self-adaptive systems have received little attention [19]. Many existing patterns target specific application domains [15], limiting their reuse outside the domains where they were originally conceived. Weyns et al. [19] 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 self-aware capability and knowledge concerns in the architecture, their attention was about MAPE component interaction.

We adopt a pattern notation, similar to the one in [19] for describing our self-aware patterns. Firstly, Weyns's notation [19] 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. Existing work on architecture patterns focus on modelling the **components** and **connectors** of architecture; in such context, components are specialisations of modules in the architecture and therefore have attributes and operations, but are also associated with the *provide* and *required* interfaces; and connectors could be the assembly that connects the *required* interface of one component to the *provided* interface of the second, or they could be the delegation that links the ports of a component to its internal parts. In our self-aware patterns, instead of modelling components, we model the capabilities of self-awareness and self-expression (e.g., stimulus awareness) in the architecture. In this way, our patterns preserve the flexibility for the concrete architectural implementation; since whether two or more capabilities are combined and realised in one component; or one capability is implemented in separate components could be based on the context. On the other hand, the connectors in our patterns are based on the standard definition but they are associated with capability rather than component. Although the capabilities of patterns are designed in a flexible manner, it is important that the interactions amongst these capabilities should not be violated when realising the pattern. For instance, one should not realise a direct interaction between a sensor and an actuator if it is not presented in a pattern. The pattern notation is depicted in figure 1.1.

Two types of connectors are used to express the logical and physical interactions. The logical connector is used to express intra-capability interaction, which is applied to capabilities of the same type across different nodes. This connector is not required to have physical interaction directly. For instance, Self-expression might be logically required to reach consensus amongst different nodes, but such interaction is physically realised through Sensors and Actuators. Physical connector is used for inter-capability interaction, which is applied to the linkage between capabilities of different types. This kind of interaction does require to physically send/receive data or control flow. In particular, physical interactions between different levels of awareness are expressed by red arrows.

There are three types of multiplicity operators (mul_op). The multiplicity operator asterisk, *, expresses that the number of capability of the same type in the interaction is restricted to at least one; 1 indicates that one and only one capability of the same type is permitted; 0 indicates that zero, one or many of the type specified is permitted in the interaction. It is worth noting that when the operator is 0, it means that the associated interaction can be removed but does not represent that the corresponding capability can be eliminated. In case a capability is interact with itself, e.g., a * on both sides of the intra-capability arrow of a capability means that it can interact with the same capability implemented in other nodes. To better clarify the operators, suppose that there is a physical interaction between stimulus awareness and external sensors where the stimulus awareness is associated with 1 whereas the external sensors is associated *. This means that within the interaction, the stimulus awareness can

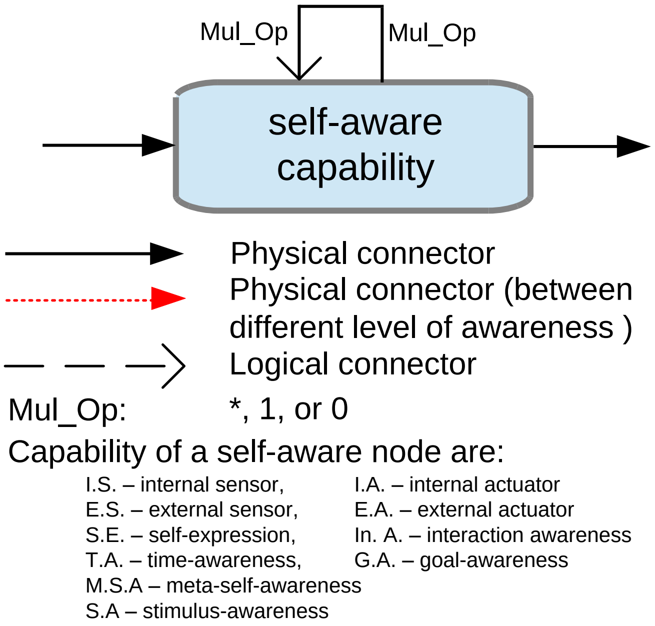


Figure 1.1: Notation for Describing Self-aware Architecture Pattern

only have one whereas the number of external sensors presented in the interaction needs to be one or many. Other multiplicity arrangements can be similarly interpreted. We document our patterns using standard pattern template [6] 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 definition of different self-awareness and self-expression capabilities.

1.1 Definition of Self-awareness

1.1.1 Private and Public Self-awareness

The sources of the relevant knowledge (i.e. internal or external sensors) for a node, underlie the notion of private and private self-awareness.

1. **Private self-awareness**: This concerns with a node possessing knowledge of and/or based on phenomena that are internal to itself.
2. **Public self-awareness**: This concerns with a node possessing knowledge of phenomena external to itself. Such knowledge grounds meaning (e.g. in the form of models) to the node's own perspective on phenomena external to itself, i.e. it is subjective to the node. This subjectivity is what underlies the notion of self.

1.1.2 Levels of Self-awareness

Described below are the levels of self-awareness, along with their relevance to either public or private self-awareness or both.

1. **Stimulus-aware**
   A node is stimulus-aware if it has knowledge of stimuli. The node is not able to distinguish between the sources of stimuli. It does not have knowledge of past/future stimuli. It enables the ability in a node to respond to events. It is a prerequisite for all other levels of self-awareness. Since stimuli may originate both internally and externally, stimulus-awareness can either be **private**, **public** or **both**.

2. **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 provoke, generate or cause specific reactions from the social or physical environment. It enables a node to distinguish between other nodes and environments. Simple interaction-awareness may just enable a node to reason about individual interactions. More advanced interaction-awareness may involve the node possessing knowledge of social structures such as communities or network topology. Interaction-awareness is typically based on external phenomena, whereby it is therefore a form of **public** self-awareness, however one can also envisage a system which learns about the effects of internal interactions with itself, which would constitute a form of **private** self-awareness.

3. **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. Since time-awareness can apply to both internal and external phenomena, it can either be **private**, **public** or **both**.

4. **Goal-aware**
   A node is goal-aware if it has knowledge of current goals, objectives, preferences and constraints. It is important to note that there is a difference between a goal existing implicitly in the design of a node, and the node having knowledge of that goal in such a way that it can reason about it. The former does not describe goal-awareness; the latter does. Example implementations of such knowledge in a node include state based goals (i.e. knowing what is a goal state and what is not) and utility based goals (i.e. having a utility or objective function). Goal-awareness permits acknowledgement of and adaptation to changes in goals. When coupled with interaction-awareness or time-awareness, goal-awareness permits the ability to reason about goals in relation to other nodes, or about likely future goals, respectively. Since goals may exist privately to the node, or collectively as a shared or externally imposed goal, goal-awareness can either be **private**, **public** or **both**.

5. **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. Such awareness permits a node to reason about the benefits and costs of maintaining a certain level of awareness (and degree of complexity with which it exercises this level). It further allows the node to adapt the way in which the level(s) of self-awareness are realised (e.g. by changing algorithms realising the level(s), thus changing the degree of complexity of realisation of the level(s)). As an example, this awareness may involve a node being able to dynamically select a particular technique out of a set of possibilities for realising one or more levels, in order to meet or manage trade-off between its goals or objectives. Since meta-self-awareness is concerned only with knowledge of internal processes, it is a form of **private** self-awareness.

1.2 Definition of Self-expression

The following ideas underpin what we mean by self-expression within a computing node.

- *A node exhibits self-expression if it is able to assert its behaviour upon either itself or other nodes.*
- *This behaviour is based upon the node's state, context, goals, values, objectives and constraints.*

Next, we present the eight self-aware patterns using the template described above. For our purposes, the state of the node comprises the self-aware capability and knowledge captured in its self-awareness processes. Thus, self-expression can be thought of as behaviour based on self-awareness.

1.3 Basic Pattern

Problem/Motivation. In some cases, a system may need to trigger some actions in order to cope with emergent events and stimuli. Such capacity could greatly help to manage system at runtime. As a result, there is an increasing need for system to react upon stimuli, based on either static or dynamic rules.

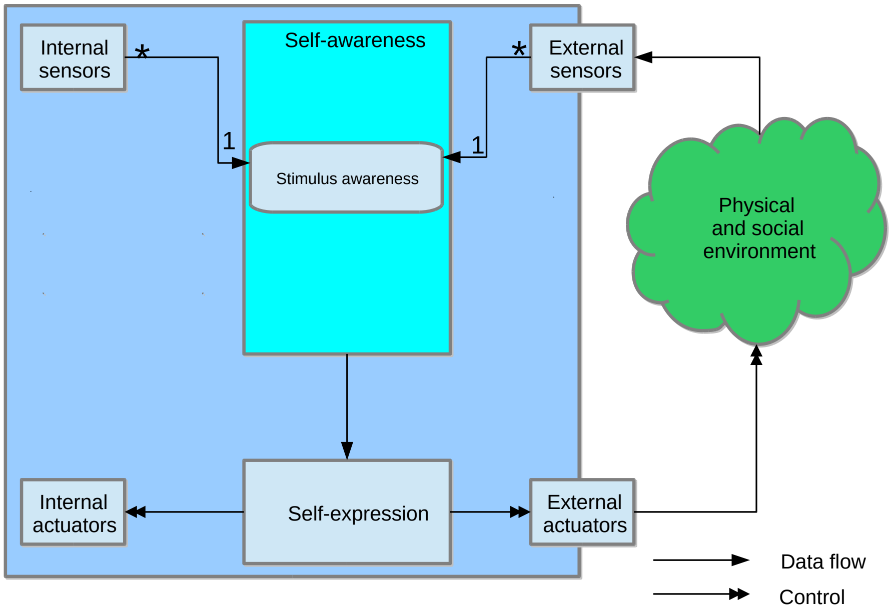


Figure 1.2: Basic Pattern

Solution. The simplest pattern to enable self-aware node is what we call Basic Pattern, as shown in figure 1.2. This pattern contains only stimulus awareness, which receiving data flow from sensors and actuators. Proper actions of self-expression could be triggered based on the type of stimulus detected. A concrete example has been shown in figure 1.3 where each node only aware of its own stimuli.

Consequences. A major limitation of this pattern is no information is shared amongst nodes, therefore the node is not aware of the environment and the other node. This could become a major problem in some cases (e.g., the smart camera case study) where there are intensive interaction and/or interference amongst nodes.

Examples. Consider the case of server farm or private cloud where the numbers of deployed applications/services are limited. The basic pattern could be realised in such context by defining *if-condition-then-action rules*, in which case the conditions could be various stimuli (e.g., QoS is low and utilisation is low); the

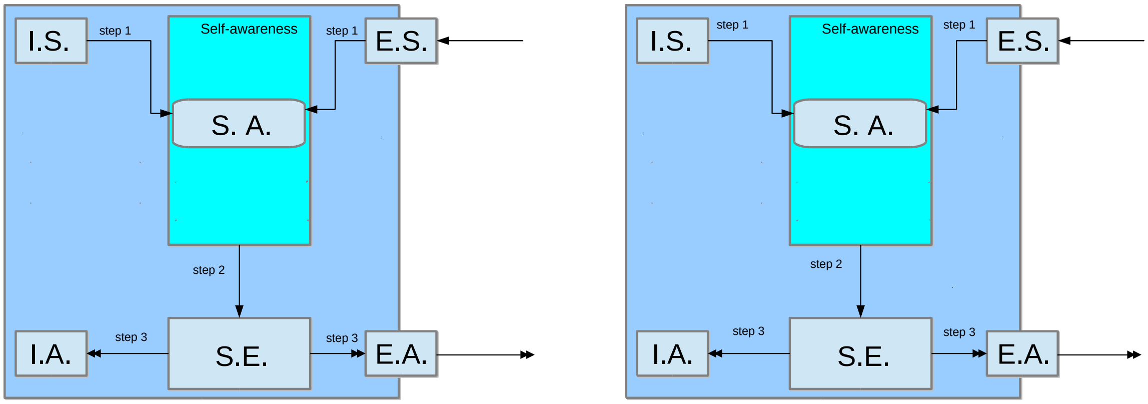


Figure 1.3: Concrete Instance of the Basic Pattern

action could be changing software configuration and/or resource provisioning.

1.4 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.
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 capability other than the stimulus-awareness. Interaction-awareness can be connected to one or more self-aware nodes as shown in figure 1.4. 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. It is important to note that nodes can not only interact with neighbours but also with their environment. For example, in the financial modelling application, interaction is all about communication between nodes and the market rather than amongst nodes themselves.

An example of the basic pattern where two nodes are connected via their interaction-awareness capabilities is shown in figure 1.5. Although only two nodes are shown in figure 1.5, 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

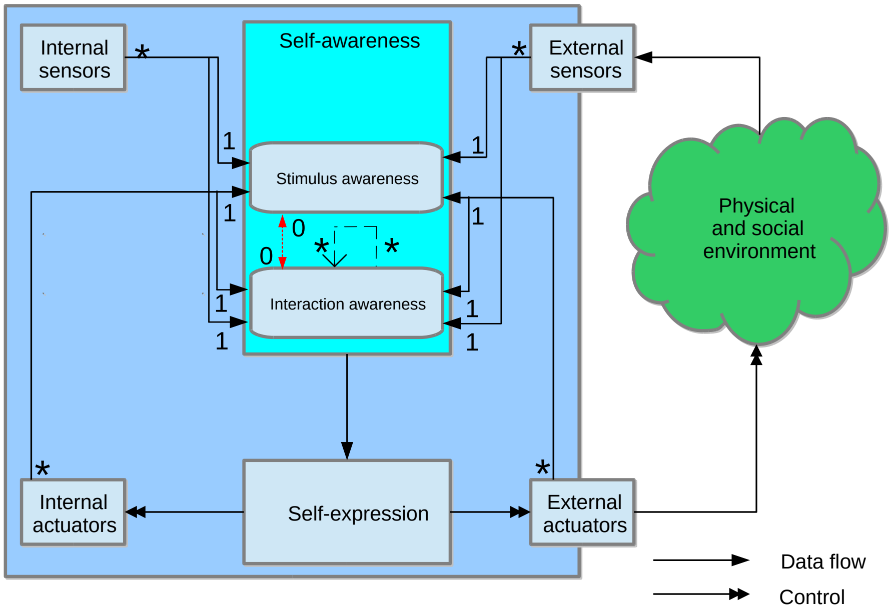


Figure 1.4: Basic Information Sharing Pattern

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.

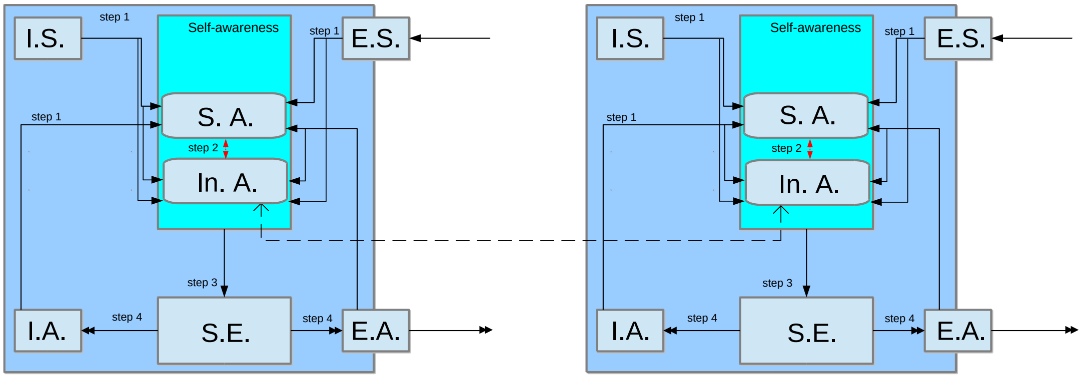


Figure 1.5: Concrete Instance of the Basic Information Sharing Pattern

Consequences. Self-aware nodes could use the interconnection between them to negotiate the protocol to use for communicating in a network. As observed in the smart camera case study, this pattern can be used to facilitate sharing information among nodes about neighbourhood relation in a network of smart camera.

Crucially, in this pattern each self-aware node maintains its autonomy about how to make adaptation decisions via its self-expression capability. 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 scenarios, 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 [20].

1.5 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 1.6 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. It is clear to see that the coordinated decision-making pattern is a related pattern to the basic information sharing pattern as they only differ on the self-expression capability. However, they are designed to aim for different problems and forces, therefore such separation of concepts paves a better way in pattern selection.

Consequences. Unlike the basic pattern, given the * to 0 multiplicity on the self-expression capability in figure 1.6, it is not mandatory for nodes to link their self-expression capabilities to each other. This makes it possible for nodes to form clusters, 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

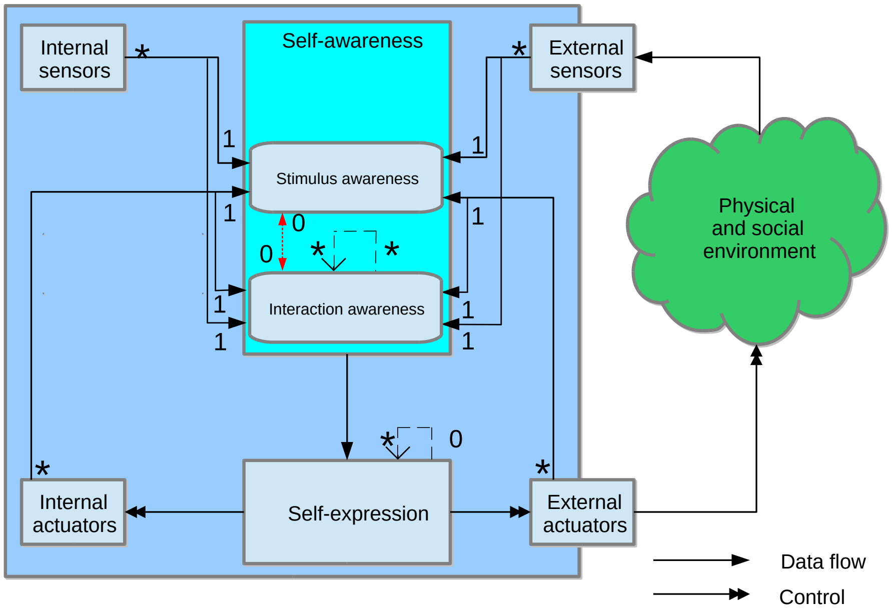


Figure 1.6: Coordinated Decision-making Pattern

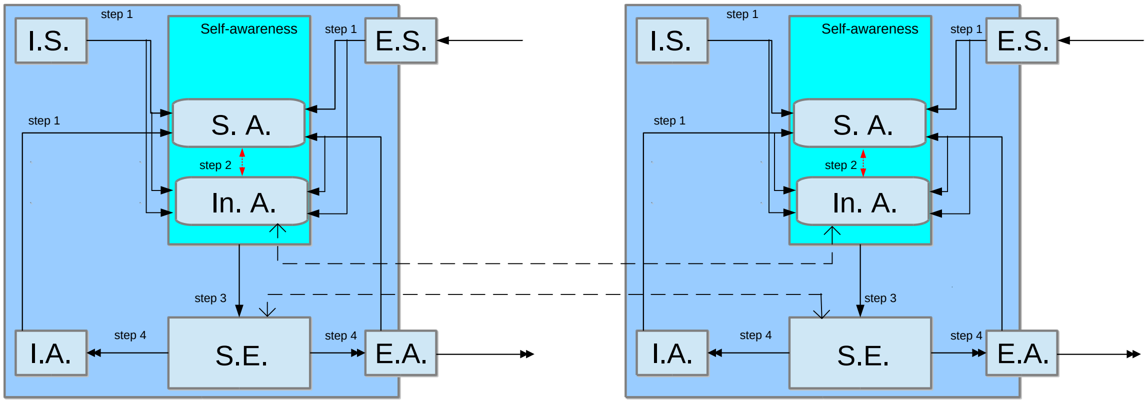


Figure 1.7: Concrete Instance of Coordinated Decision-making Pattern

parts. Figure 1.7 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 1.7 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 capabilities, upon agreement the self-expression capability of each cloud enforce the agreed policy within its (local) cloud.

1.6 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.

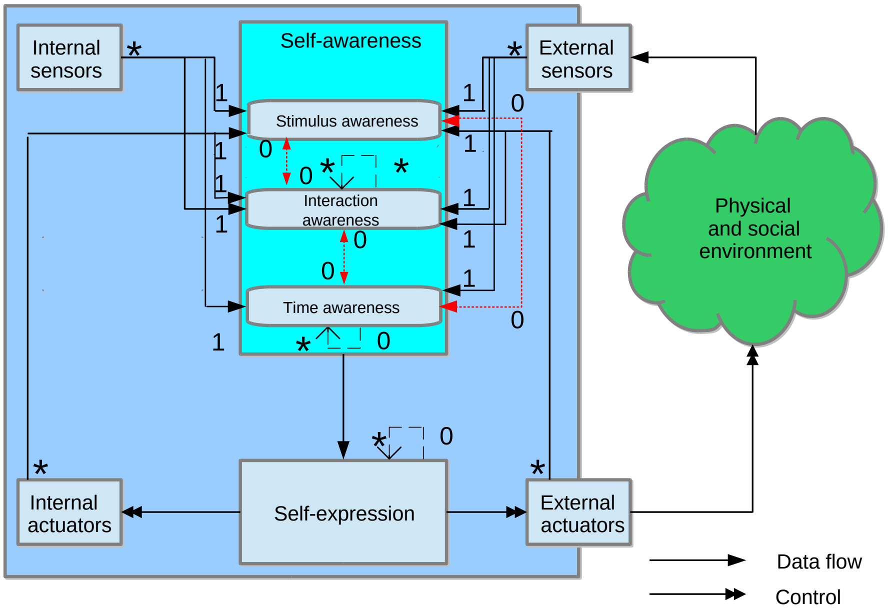


Figure 1.8: Temporal Knowledge Sharing Pattern

Solution. The temporal knowledge sharing pattern solves this problem by incorporating time-awareness capabilities into the coordinated decision-making pattern. As shown in figure 1.8, each self-aware node has a time-aware capability 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 1.9. This timing information can be exploited by the self-expression capability to manage the timing of adaptation actions across multiple nodes.

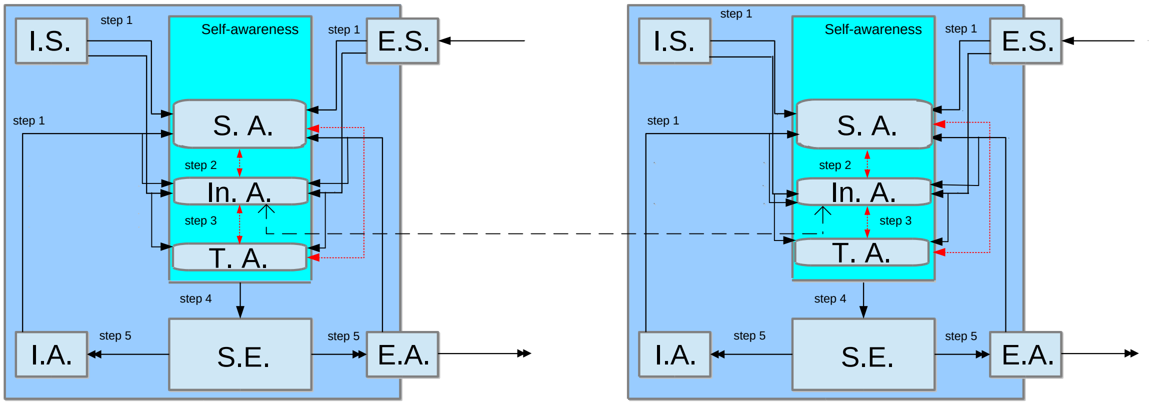


Figure 1.9: Concrete Instance of Temporal Knowledge Sharing Pattern

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.

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.

1.7 Temporal Knowledge Aware Pattern

Problem/Motivation. The knowledge of timing enables the capability of proactive adaptation and potentially, better adaptation quality. Within the previously mentioned pattern, Temporal Knowledge Sharing pattern is the only one that applies time awareness capability. However, a drawback of such pattern is that the interaction awareness capability might not be a unnecessarity, therefore it could affect the self-aware system as it is suffering unnecessary overhead.

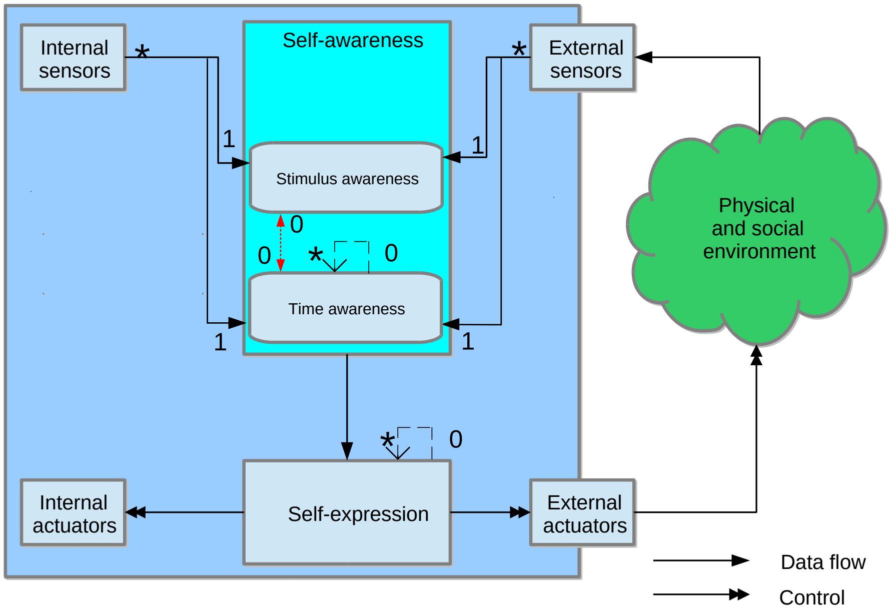


Figure 1.10: Temporal Knowledge Aware Pattern

Solution. As shown in figure 1.10, the temporal knowledge aware pattern solves this problem by incorporating only time awareness working in conjunction with stimulus awareness. Again, the time awareness capabilities of different node is logically linked together (optionally). This pattern allows the knowledge of timing to assist the self-expression capability and overhead the extra overhead produced by unneeded level of awareness. A concrete example has been shown in figure 1.11.

Consequences. There are scenarios where the software designer is uncertain about whether the lack of environmental information and information could affect the modelling of timing knowledge. This is highly depend on the concrete time-series prediction technique in the time awareness capability. An inappropriate use of certain time-series prediction technique could result in low accuracy, which eventually affect the quality of adaptation. As a result, the decision of which time-series prediction technique to be used is critical and the designers are recommended to consult experts of time-series modelling when applying this pattern.

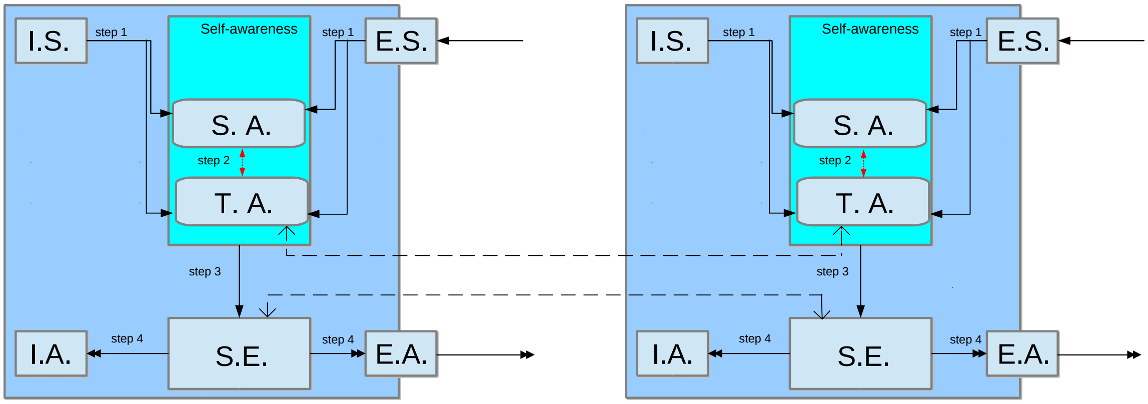


Figure 1.11: Concrete Instance of Temporal Knowledge Aware Pattern

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 Sharing Pattern* - will address the challenge of modifying or changing goal at run-time.

Examples. Cloud is an environment where resource is sharing via Virtual Machine (VM) on each node. In this context, by leveraging the historical usage of resources, time-series prediction would be able to predict the demand of VMs on a node for the nearly future, which assists proactive provisioning of resource and potentially, prevents SLA violation and/or resource exhaustion on a node.

1.8 Goal Sharing 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. In the smart camera case [12], a camera running low on battery may choose to bid for only the most valued objects within its field of view instead of aiming to track all objects in its vicinity. 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 1.12 shows the goal sharing pattern that address the concern of representing run-time goal. A goal-awareness capability represents knowledge about run-time goals, which can be changed as the system evolves. The

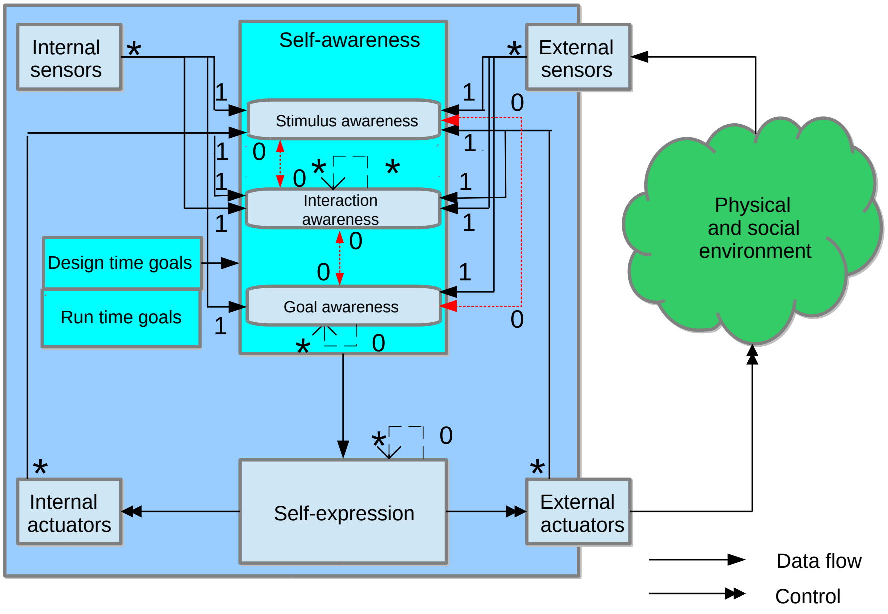


Figure 1.12: Goal Sharing Pattern

goal-awareness capability in a self-aware node can, optionally, share its state information with goal-awareness capabilities 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.

Consequences. As can be observed from figure 1.12, a time-awareness capability is not included in this pattern. This implies that time-awareness is not a necessary perquisite 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 different pattern that addresses this limitation (see figure 1.13). The pattern in figure 1.13 makes the inclusion of temporal knowledge explicit, making it suitable for application domains where changing goals and forecasting are required.

In both patterns, self-expression capability makes use of the goal-awareness capability to make strategic decisions in line with the system's current goal. Figure 1.14 shows an instance of the pattern (without time-awareness), while

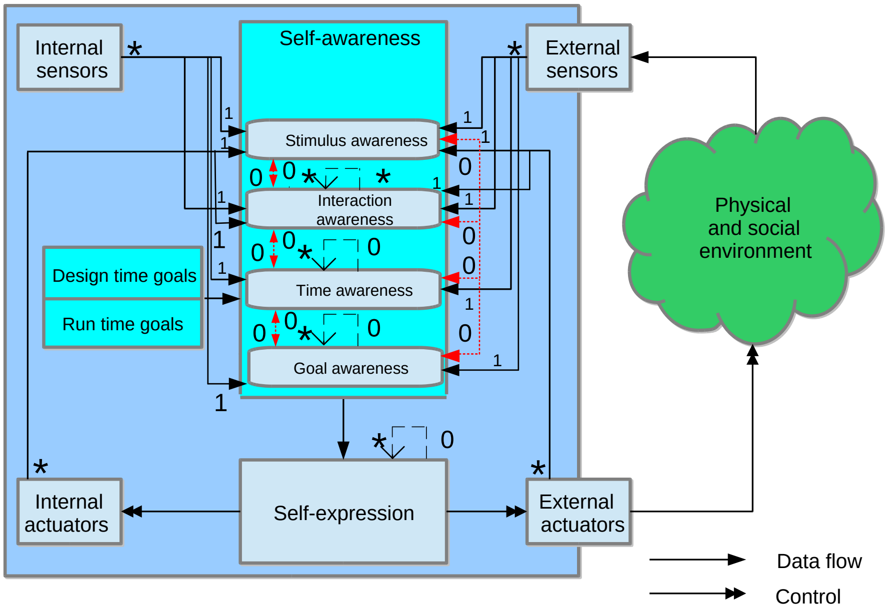


Figure 1.13: Goal Sharing Pattern (with time-awareness capability)

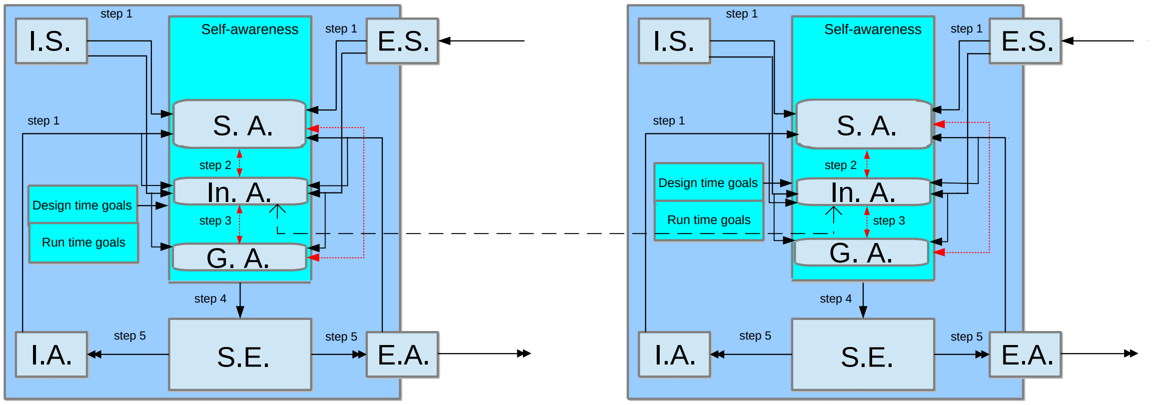


Figure 1.14: Concrete Instance of Goal Sharing Pattern (without time-awareness capability)

figure 1.15 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.

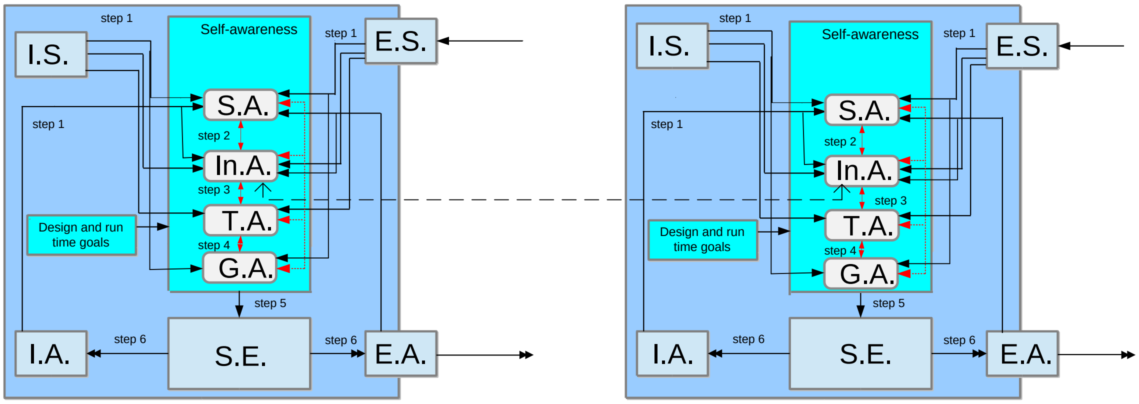


Figure 1.15: Concrete Instance of Goal Sharing Pattern (with time-awareness capability)

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 goal-sharing pattern with time-awareness capability (see figure 1.13) 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 capability 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.

1.9 Temporal Goal Aware Pattern

Problem/Motivation. The knowledges of goals and time might not necessarily to be shared amongst nodes, especially in cases where the optimisation of local goals could lead to acceptable global optimun. As a result, the presence of interaction awareness capability could cause extra overhead on the system.
Solution. As shown in figure 1.16, the temporal goal aware pattern solves this problem by removing the interaction awareness capability. In this pattern, there is no notion of 'sharing' as the nodes are not aware of any interactions and therefore not aware of the presence of the other nodes. It is worth noting that the absence of interaction awareness does not mean there is no interaction - nodes and the environment could still interact with each other, but the nodes

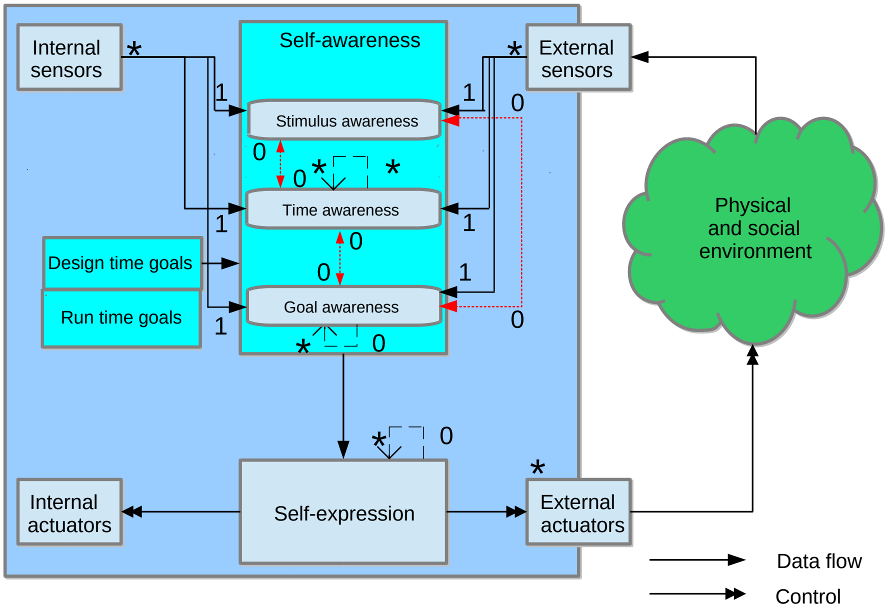


Figure 1.16: Temporal Goal Aware Pattern

are not aware of it. A concrete example has been shown in figure 1.17.

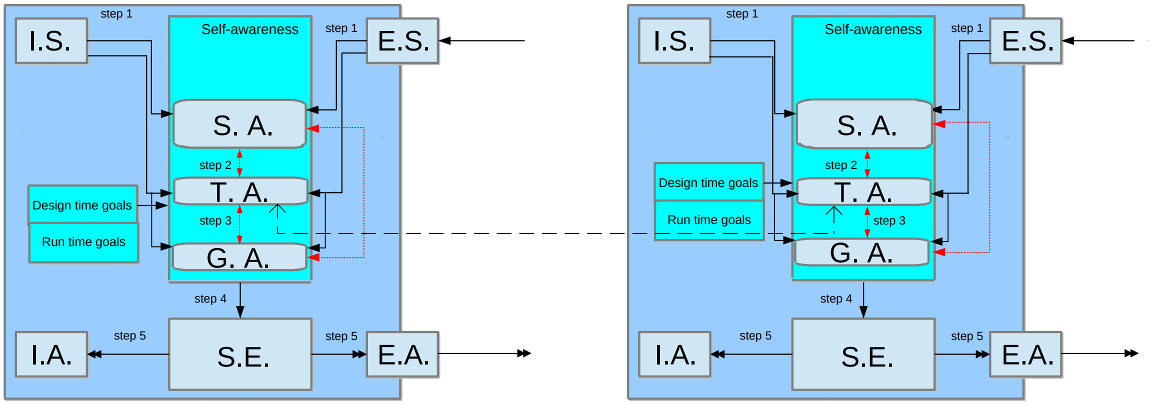


Figure 1.17: Concrete Instance of Temporal Goal Aware Pattern

Consequences. The removal of interaction awareness implies that the nodes could be in inconsistent state. The designer should carefully verify that such situation would not result in violations of system requirements. In addition, the self-expression capability could not use any information from other nodes when making decisions.

Examples. Orchestrate fully decentralised harmonic synchronisation amongst different mobile devices requires each node to aware of stimulus, time and goal but not necessarily interaction. In such case, each node receives phase and frequency updates from the other nodes or the environment, and reacts upon based on its own time and goal information. This is a typical example where there are occurrences of interaction, but no occurrences of interaction awareness; because the nodes only aware of the incoming phase and frequency updates but it has no knowledge of where they come from; the sources could be other nodes, the environment or even some unexpected noise.

1.10 Meta-self-awareness and Self-aware Patterns

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 [10]. 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 capability.

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 capability depending on the complexity to be managed and expertise of the application designer. Figure 1.18 shows the goal-sharing pattern with time-awareness capability where a meta-self-aware pattern is present to manage trade-off between goal-, interaction-, and time-awareness capabilities. Presumably, the presence of meta-self-awareness capability could help to switch between different pattern at runtime, which could be a very interesting direction for future work.

There are also other examples of meta-self-awareness capability in EPiCS. For instance, in the smart camera demonstrator [12], the meta-self-awareness

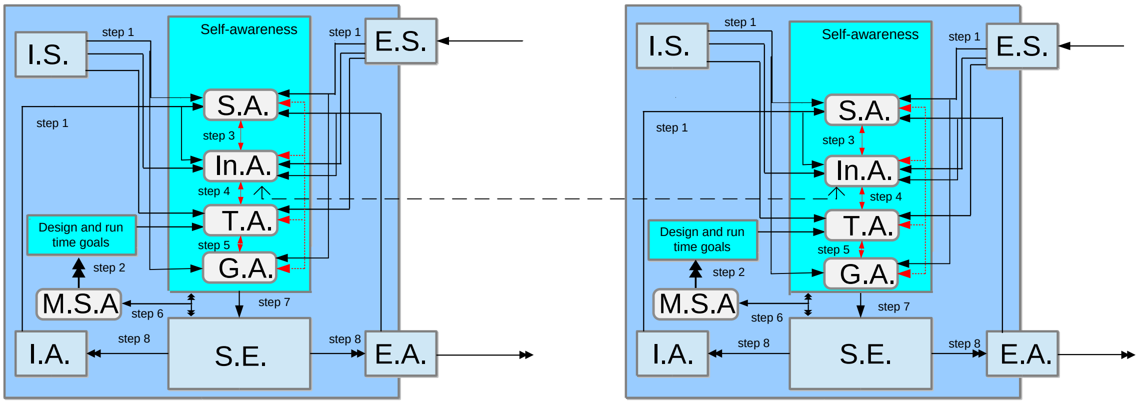


Figure 1.18: Concrete Instance of Goal Sharing Pattern (including meta-self-awareness capability)

is used to switch between different behavioural strategies in the interaction awareness and self-expression capabilities; In the hyper music demonstrator, the meta-self-awareness can help to control the degree of stimulus awareness based on confidence measure.

Questions & Answers of the Patterns

- Q: Can the nodes be heterogeneous (i.e., different nodes implementing different patterns)?
  A: Yes, different pattern can be chosen to realise different node if it is required.
- Q: Can we realise different capabilities using the same technique?
  A: Yes, the notions of capability itself is flexible.
- Q: Can we have logical connections between different capabilities (e.g., goal and time awareness)?
  A: Yes, logical connections is not restricted as physical ones.

Chapter 2

Architectural Primitives for Self-aware Systems

These architectural patterns provide proven solution to recurring design problems that arise in a context of self-aware and self-expressive system. However, the pattern itself is an abstract form of the architecture. Often, the abstract patterns can be used as the first step for architecting and engineering self-aware systems by mapping them with an existing/candidate architecture instance, which is application specific. We have organised a workshop that allows the partners to present their results in using the patterns. After the workshop, we have observed that the partners could use the same pattern in many different ways depending on the problem context and their concrete architecture instance (e.g., whether to use meta-self-awareness; whether to use a particular technique to realise a capability; whether to realise two capabilities in the same component); however, the concrete applications of patterns in different contexts still share many underlying concepts. These shared concepts lead to the notion of **architectural primitive**, which refers to the common concepts that can be constructively used to form a concrete architecture instance of a pattern. The architecture instances of a pattern differs in terms of the candidate techniques and/or attributes (i.e., a particular form of an architectural primitive) that used to realised each architectural primitive. These differences amongst architecture instances of a pattern are referred to as **pattern variability**; similarly, the alternative architecture instances of a pattern are the **variants** of this pattern.

Architectural primitives and pattern variability are long-studied problems [14]. However, unlike traditional work that aim for generic system architectures, we are specifically interested in how these concepts can be used in self-aware systems. In particular, we aim to link these well-studied concepts to the self-awareness principles [5], which are all about different levels of knowledge awareness. In this report, we have documented the architectural primitives in four categories (i.e., Capability, Behaviour, Interaction and Topology), each of which cover an aspect of a self-aware system. We anticipate that a benefit from this will be a reduction in the chances to introduce faults, and easier fault detection, when using the proposed patterns during design. We show the potential dependency between these primitives. These architecture primitives and candidate techniques and/or attributes have been used in EPiCS to form different variants of a pattern.

2.1 Architectural Primitives and Candidate Techniques

To better understand the orthogonal aspects of self-aware systems, we have used four categories. This is motivated by our investigation about the critical aspects that could affect the functional and/or non-functional requirements of a self-aware system. These categories are shown as below:

- **Capability**: The capability of the system to obtain certain knowledge or to react based upon the knowledge. e.g., the levels of awareness and expression.
- **Behaviour**: The process of a capability regarding how input data is consumed and output data is produced.
- **Interaction**: The relationship between the capabilities or of a capability itself as expressed by the multiplicity operators.
- **Topology**: The deployment about how capabilities are distributed to the components in the architecture instance.

We have structured the architectural primitives to express: (i) the characteristic of a components or connector; and (ii) the function of a components or connector. The architectural primitives with respect to the four categories are shown as below, the terms in the bracket are the attributes of an architectural primitive:

- **Capability**: stimulus-awareness, interaction-awareness, time-awareness, goal-awareness, self-expression, meta-self-awareness.
- **Behaviour**: send (synchronous or asynchronous), handle (sequential or parallel), state (proactive or reactive), transit.
- **Interaction**: link (one-to-many, many-to-many, one-to-one or none).
- **Topology**: structure (combine, separate or compact), existence (exist or non-exist).

The use of these primitives is application specific and depends on whether they would affect the functional and/or non-functional requirement of the self-aware systems.

2.1.1 Capability

The capable primitives consists of the 4 level of knowledge in self-awareness principles, which can be realised in one or more components in an architecture instance. The *self-expression* and *meta-self-awareness* capabilities are also belong to this category. These primitives belong to capability can be realized by using different candidate techniques.

2.1.2 Behaviour

In the behavioural primitives, *send* is used to describe the output process of a sender. In particular, it has two attributes: *synchronous* and *asynchronous*. *Synchronous* refer to the scenarios where after the sender sends a request, it needs to wait for a reply from the receiver before transit to the next actions. On the other hand, *asynchronous* refer to the case that the sender does not require such blocked communication. *Handle* is used to express how the receiver process inputs. *Sequential* handler refer to the cases where incoming data are processed by specific sequence using a queue. On the other hand, *parallel* handler simply process all incoming data in parallel upon its arrival. Both *send* and *handle* primitives can be realised using different candidate techniques. *State* primitive is used to express the behaviour of a capability, it depends on the candidate techniques that realise such capability. In particular, In the *reactive* state, the capability responds when a change has already happened, while in the *proactive* state, the capability predicts and reasons about when the change is going to occur, it then act upon [9]. It is also possible that a capability is realised using both *proactive* and *reactive* attributes. *Transit* is a specific behaviour of meta-self-awareness capability; it aims to reason about and switch on/off one or more other capabilities. Similarity, it can be realised using different candidate techniques. The main differences between the candidate techniques of *meta-self-awareness* and those of *transit* is that the former one focus on how to improve and optimise other capabilities; whereas the later on focus on the question of whether we should keep or shutdown certain capabilities? However, we have not seen any application of techniques belong to *transit* primitive in EPiCS.

2.1.3 Interaction

Interactive primitive, *link*, is used to describe the physical and logical relationship between the capabilities as expressed by the multiplicity operators. It has four attributes: *one-to-many*, *many-to-many*, *one-to-one* or *none*. The use of this primitive is constrained by the selected pattern.

2.1.4 Topology

Finally, topological primitives are design for expressing how the capabilities are mapped to the components of an architectural instance. *Structure* primitive has three attributes: *separate* or *compact*. *Combine* is used to describe the cases where a capability is realised in conjunction with other capabilities within a single component; conversely, the *separate* refers to a single capability is realised using separate components; *compact* refers to a capability is realised using exactly one component. In particular, it is possible to have both *separate* and *compact* for a capability. *Existence* primitive is simply used to show whether a capability is needed, as certain pattern allows optional capabilities.

A detailed summary of the architectural primitives and a list of possible candidate techniques surveyed from the EPiCS projects has been shown in Table 2.1. Please note that the list of candidate techniques here is not exhaustive, therefore more exampled candidate techniques can be added when appropriate. We omitted some primitives (e.g., the *link*) here as they do not associated with any concrete techniques.

Table 2.1: Architectural primitives and their candidate technique

Architectural Primitives	Candidate Techniques/Options			
	Technique	**Demonstrator**	**Approach Details**	**Nature**
stimulus-awareness	Simple Update	Financial Application [2]	1. Read from either human inputs or files	Stimulus: market data (price, interest rate, trade), parallelism, precision, IO bandwidth
	Auction Invitation/Bidding	Smart Camera [5]	1. Perform background subtraction (private) 2. Bound boxes candidates (public) 3. Perform valuation of bids based on tracking results (private)	Stimulus: images, auctions, bids, handover, and object tracking results.
	Background Substitution			
	Bounding Boxes Generation			
	Threshold-based Algorithm	Dynamic Protocol Stack [7]	1. Simple mapping: map most-used functional block to HW 2. Smart mapping: considers processing time per block	Stimulus: Workload, core utilization, performance, Network condition, link quality, signal-to-noise ratio, bandwidth
	Conditions-Actions Rule	HW/SW Platform [6]	1. Trigger performance management / thermal management	Stimulus: performance counter, local temperature, input data, thermal diode
	Simple Update Function	Rhythmic Music Application [3]	1. Trigger phase and frequency updates upon input from other nodes.	Stimulus: Sound input from other nodes
interaction-awareness	Market-based Mechanism	Smart Camera [5]	1. Create model of objects to be tracked (public) 2. Hand-over process using broadcast 3. React to auctions, bids and handover (public) 4. Define neighbourhood based on auctions (private)	Interaction: Participation in auction & vision graph, Model of objects
	Object modelling and matching			
	Threshold-based Algorithm	Dynamic Protocol Stack [7]	1. Packet processing engine (stack builder) 2. Source node proposes a set of protocol stacks 3. Meet application / node goals, network conditions 4. Destination node selects one and informs source 5. Both nodes can suggest run-time adaptations	Interaction: Source and destination nodes negotiate protocol stack.
time-awareness	Ant Inspired Artificial Pheromones	Smart Camera [5]	1. Calculate pheromones in vision graph (private)	Time: Evaporation of pheromones
	Partial Autocorrelation	Dynamic Protocol Stack [7]	1. Perform proactive reconfiguration of HW/SW mapping 2. Create fitness function for HW/SW mapping	Time: Historical traffic patterns
	2-Layer Resistor-Capacitor (RC) Network	HW/SW Platform [6]	1. Learn thermal model of chip	Time: Historical temperature data of the chip
	Randomized Hill Climbing			
	Median Filter with Moving Average	Rhythmic Music Application [3]	1. Filter the errors in frequency	Time: Historical frequency errors.
	Hawkes Point Process Estimation	Financial Application [2]	1. Perform point process 2. Correlate time series	Time: Historical financial trends
	HAC estimation and verification			
goal-awareness	Error/Confidence Measure	Rhythmic Music Application [3]	1. Learn the tempo change and how it affects the goals.	Goal: minimizing error of producing music
self-expression	Hand-over Mechanism	Smart Camera [5]	1. Send out auction invitations and bids (public)	Expression: Pan-tilt-zoom (internal), decision to participate in

			2. Automate load balancing	auction (internal), communication channel – send bids/initiate auctions (external)
	Static Mapping	Dynamic Protocol Stack [7]	1. Update the protocol stack mapping 2. Minimize CPU utilization	Expression: Send normal packets, send messages for negotiation, setup protocol stacks, migrate functional blocks between HW/SW
	Brute Force Optimization	HW/SW Platform [6]	1. Find the optimal thermal mappings	Expression: Start/stop/migrate threads, generate heat on the chip
	Simple Update	HW/SW Platform [6]	1. A confident node will make sound at a steady tempo. 2. An insecure node will constantly try to adapt its tempo to the other nodes.	Expression: Send tones and make sound.
	Gaussian Process	Financial Application [2]	1. Optimize decision values and decisions for financial trading. 2. Improve design fitness by exploring / tuning several design parameters	Expression: Design configurations, trend prediction decisions
	Support Vector Machine			
	Reinforcement Learning			
meta-self-awareness	Hand-over Mechanism	Smart Camera [5]	1. Perform hand-over process using vision graph information (private/public) 2. Bandit solvers introduce very simple meta-self-awareness (private)	Meta-self-awareness: choose the cameras for the auction, start auction, hand-over, update vision graph
	Bandit Solver			
	Quality- driven and Threshold-based Algorithm	HW/SW Platform [6]	1. Select best strategy to meet goals in current situation	Meta-self-awareness: Switch between different scheduling strategies for thermal management
		Rhythmic Music Application [3]	1. Change scaling of frequency updates based on confidence, i.e. managing the degree of stimulus awareness	Meta-self-awareness: Change the degree of adapting frequency, which is directly controlled by confidence measure
send	synchronous function call	Dynamic Protocol Stack [7]	Applied to every capability except self-expression	
		HW/SW Platform [6]	Applied to every capability except self-expression	
		Rhythmic Music Application [3]	Applied to every capability except self-expression	
		Financial Application [2]	Applied to every capability except time-awareness	
	synchronous multicast	HW/SW Platform [6]	Applied to self-expression capability	
	asynchronous function call	Financial Application [2]	Applied to time-awareness capability	
	asynchronous multicast	Dynamic Protocol Stack [7]	Applied to self-expression capability	
		HW/SW Platform [6]	Applied to self-expression capability	
		Smart Camera [5]	Applied to every capability	
	asynchronous broadcast	HW/SW Platform [6]	Applied to self-expression capability	
	asynchronous stigmergy	Rhythmic Music Application [3]	Applied to self-expression capability	
handle	First-Come-First-Serve (sequential)	Dynamic Protocol Stack [7]	Applied to every capability except self-awareness to handle data from external sensors	
		HW/SW Platform [6]	Applied to every capability except self-awareness to handle data from external sensors	
	multi-threading (parallel)	Smart Camera [5]	Applied to every capability	
		Rhythmic Music Application [3]	Applied to every capability	
		Financial Application [2]	Applied to every capability	
		Dynamic Protocol Stack [7]	Applied to self-awareness to handle data from external sensors	
		HW/SW Platform [6]	Applied to self-awareness to handle data from external sensors	

It is worth noting that it is possible to select more than one candidate techniques for a primitives, each work on a particular aspect of a capability. In addition, we do not provide candidate techniques for sensor and actuator as they are highly application specific and in some cases, they are uncontrollable.

2.2 The Dependency

It can be clearly seen that dependency exist amongst some of the aforementioned architectural primitives, that is, certain primitives can not be used if another primitive or attribute has not be considered. The dependency has been illustrated in Figure 2.1. Note that the arrow here means 'or' relationship, e.g., *state* primitive can be considered if any capability primitive exists.

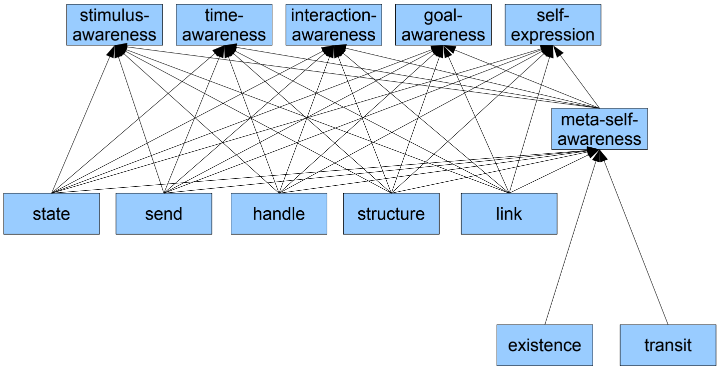


Figure 2.1: Dependency amongst primitives

Chapter 3

Pattern Driven Methodology for Engineering Self-aware and Self-expressive Systems

Engineering self-aware and self-expressive systems is a widely important and complex activity. This is because the process involves many decision makings on the possible alternatives (e.g., what technique/attribute should one apply in order to realise certain level of awareness?) even in the early stage of development. In addition, it is difficult to throughly reason about the consequences of different design alternatives to the functional and non-functional requirements of the systems. Research for engineering methodology has been widely conducted on the area of conventional software architecture [14, 17, 16, 4], however a systematic approach to the design and engineering of self-aware and self-expressive systems is still in its infancy. In this report, we present a pattern driven methodology to this engineering problem by leveraging on previously proposed patterns and architectural primitives. The methodology contains detailed guidance to make decisions with respect to the possible design alternatives. We evaluate the approach in two aspects: (i) a qualitative evaluation using two case studies: the smart camera networking problem within EPiCS [11, 12] and the elastic cloud autoscaling problem [7, 8], which is an example outside EPiCS; and (ii) a quantitative assessment by comparing the resulted self-aware and self-expressive system to a conventional and non-self-aware system.

3.1 The Methodology Overview

To facilitate a systematic way of building self-aware and self-expressive systems, we proposed a pattern driven methodology leverage on the 8 proposed patterns and the defined architectural primitives. This methodology is a variation of ATAM [17] extended by applying patterns and quality-values analysis in relation

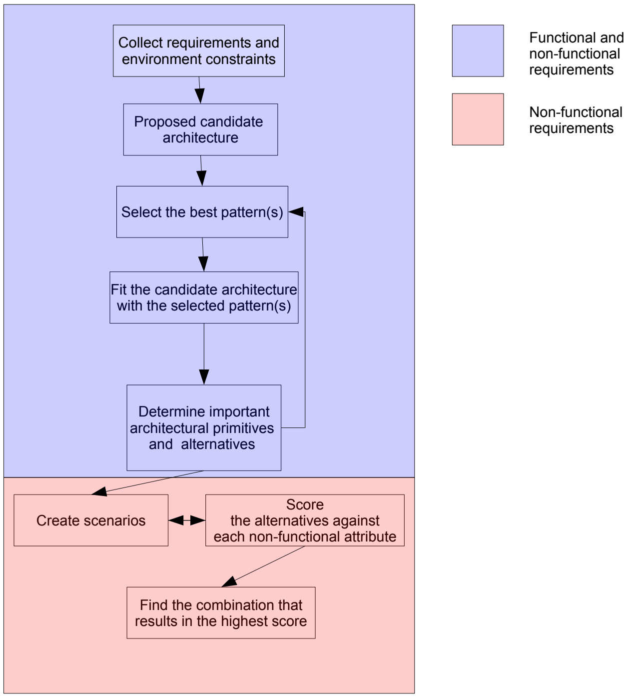


Figure 3.1: Overview of the pattern driven methodology

to the self-awareness principles. In particular, the aim of such methodology is two-folds: (i) select the right pattern(s); and (ii) select the right variation of the chosen pattern(s). As shown in Figure 3.1, we can see that the initial four steps, responsible for selecting the right pattern, are tightly related to the functional and possibly the non-functional requirements of systems. After the selection of pattern, there is an intermediate stage (step 5) aims to determine the important design decisions for selecting variant with respect to the selected pattern. The last three steps, on the other hand, are used to select the right variation of the chosen pattern; and they are associated with the non-functional aspects of the systems. We argue that the separate consideration of requirements for selection of pattern and pattern variation promote a concise and precise design of self-aware and self-expressive systems. In the following, we will see each of the steps in details.

3.1.1 Step 1 - Collect Requirements and Constraints

The first step in this method involves collecting requirements and constraints from the stakeholders and environment for engineering self-aware and self-expressive systems. Similar to the step in ATAM [17], the purpose of this step is to gain depth knowledges about the problem context; to operationalise both functional and non-functional requirements, to facilitate communication between stakeholders, and to develop a common vision of the important activities the system should support.

The requirements could be either functional or non-functional, for instance, *the system should be able to record historical data for analysing its behaviour* is clearly a functional requirement. On the other hand, *the adaptation delay in the system should be no more than 1 minute* is an example of non-functional requirement. Constraints are also another important factor to be considered, in particular, they could come from the stakeholders, e.g., *the hourly cost of infrastructure for running the system should not be more than 10 dollars*; or from the environment, e.g., *the topology of nodes in the context is dynamic as nodes can join/leave on demand*.

3.1.2 Step 2 - Propose Candidate Architecture

Our methodology assumes that there is an existing candidate architecture of the system. This is because our self-aware patterns are generic and do not rely on any assumptions about the application domain. Therefore, in order to apply the pattern in practise, it is essential to obtain certain knowledge about the architecture related to the context of given application domain. Often, this knowledge is represented as a collection of components and connectors, which we call them existing candidate architecture. By doing so, we could obtain two major benefits: (i) it could provide a clear view about what is really needed by the domain from an architecture perspective, and thus assist the engineer to reason about and select the right pattern; (ii) it is possible to refine the existing candidate architecture when mapping the components and connectors to the capabilities of chosen pattern(s). We believe this is a rational assumption as once the important requirements and constraints have been determined, the engineer should be able to propose a candidate architecture based on the obtained information. In addition, design almost never starts from a clean slate: legacy systems, interoperability, and the successes/failures of previous projects all constrain the space of architectures.

The candidate architecture must be described in terms of architectural components/elements. In particular, the architecture should express the module/component view [?] of the system, which is usually used to reason about work assignments and information hiding. In this work, we do not assume any standard notations (e.g., UML) therefore the engineer could use either the standard ones or invited his/her own notations.

3.1.3 Step 3 - Select the Best Pattern(s)

This is the first critical step in our methodology, which is about selecting the suitable pattern(s) based on the functional and non-functional requirements as well as the constraints. By doing so, the pattern could help the engineer to think of the domain specific architecture problem in a self-aware computing sense. The patterns differ mainly in terms of the self-awareness and self-expression capabilities, which play an integral role in satisfying the functional requirements, therefore the selection of pattern is equivalent to the selection of the right set of self-awareness and self-expression capabilities.

To promote a systematic approach for pattern selection, we design certain questions for each self-awareness and self-expression capability and ask the software engineer to consider these questions. The suitable pattern(s) could be determined based on these answers. Specifically, one should answer the following questions:

- What does the capability mean in your problem context?
- What are the functional requirements that affected by this capability?
- What are the non-functional requirements that affected by this capability?
- What are the constraints that could affect this capability?
- Whether this capability is necessary or beneficial?

These questions could lead to the conclusion about whether a capability should be included in the design. It is also possible to answer the questions in more than one different sets depends on how many patterns would be used. for example, if the case requires two pattern, then there could be two sets of answers for each self-aware capability; one set for each pattern. Finally, Table 3.1 could be used to select the suitable pattern(s).

Table 3.1: The patterns with respect to the principle of self-awareness. (y means the capability is included in the pattern; o denotes that this capability is optional)

	Basic Pattern	Basic Information Sharing Pattern	Coordinated Decision Making Pattern	Temporal Knowledges Sharing Pattern	Temporal Knowledges Aware Pattern	Goal Sharing Pattern	Goal Sharing Pattern (with time)	Temporal Goal Aware Pattern
Stimulus-awareness	y	y	y	y	y	y	y	y
Time-awareness				y	y		y	y
Interaction-awareness		y	y	y		y	y	
Goal-awareness						y	y	y
Meta-self-awareness	o	o	o	o	o	o	o	o

3.1.4 Step 4 - Fit the Selected Pattern(s)

Once the best pattern(s) has been determined, we can now fit the components/elements from the candidate architecture to the capabilities described in the pattern(s). It is worth noting that, our architectural patterns preserve the flexibility for the concrete architecture; since whether two or more capabilities are combined and fit in one component; or one capability is fit in separate components could be based on the requirements and constraints. It is also an opportunity to refine/improve the candidate architecture udring the mappings.

3.1.5 Step 5 - Determine the Important Primitives and the Possible Alternatives for Non-functional Requirements

The next step is to determine the important architectural primitives and the relevant alternatives (certain form of techniques) for the given problem context. Particularly, we should consider the non-functional requirements here and link them to these architectural primitives, their attributes and the relevant techniques. Those primitives, their attributes and techniques that could influence the non-functional requirements would be extensively modelled and examined for selecting the right ones during next steps. In addition, this is also an opportunity to eliminate the primitives and techniques, which could be easily selected or are trivial to be considered. For example, if performance requirement is much more important than the others and the environment conditions are dynamic, then the behaviour primitives (e.g., the primitives send and handle) could be eliminated as it is almost certain that only multicast and parallel interactions are feasible here.

Previously, we have reported on a list of techniques and attributes for each architectural primitives. But not all of them are feasible depends on the problem context. The most important task is to determine the alternatives out of the candidate techniques and attributes. Here, the software engineer could also eliminate the techniques/primitives that could only affect functional requirements or that are useless due to some constraints of the environment. For instance, a supervised learning algorithm would not be useful in the case where only unlabelled data is available. In addition, more appropriate techniques that have not been reported could be added in to the consideration. The common practise of identifying alternative is to consider each technique as independent alternative. However, it is possible to create ensemble of techniques and attributes, therefore we consider such ensemble and any of this combination as independent alternative.

From this step forward, our methodology would start focusing on the non-functional requirements because they are more difficult to be satisfied due to their compliances are often related to runtime uncertainty. In addition, the non-functional attributes are usually highly sensitive to the variants of patterns. The problem become even more complex when the non-functional requirements are conflict, e.g., accuracy vs overhead.

Up to this step, it is possible to go back to Step 2 if one feels that the chosen pattern(s) is inappropriate. This iterative process can continue till the final suitable pattern(s) is determined.

3.1.6 Step 6 - Create Scenarios

At this stage, we create the important scenarios that could influence each non-functional attribute and likely to occur at runtime. There is no restriction on the number or granularity of scenario per non-functional attribute, for instance, one could have a faulty scenario for a system running in stable state in order to examine the influence to availability attribute; in addition, he/she could also create a scenario under the unstable state (e.g., when new nodes join in or existing nodes leave). It is worth noting that the defined scenarios do not need to be exhaustive as using these scenarios do not imply the developed system is only able to cope with theses scenarios. Selecting pattern variants based on these scenarios could provide confidence about how the self-aware system would perform under the likely scenarios.

It is worth noting that there are cases where it is very difficult if not impossible to design scenarios, especially when these scenarios can not be foreseen and can only be dealt with in real time. In these case, the selection of pattern variants (step 6-8) can be skipped as it is impossible to assess and compare alternatives. Instead, the proper variants can be determined based on the knowledge of domain experts, e.g., previous publications, experiments, implementations etc.

3.1.7 Step 7 - Score the Alternative of Primitives Against each Non-functional Attribute using Analytical or Simulation Models

In this step, we need to determine the score for each alternative of a primitive against each non-functional attribute under the considered scenarios. The purpose of this step is to assess each alternative with respect to the non-functional attributes. We assume that there is no or limited dependency between the architectural primitives in terms of how they affect the non-functional attributes, therefore we assess each primitive in isolation. In particular, we aim to gain relative score for every alternative in the context of each architectural primitive. Scoring an alternative can be achieved by using either analytical model or simulation where the former one refers to the empirical analysis of a particular alternative against a non-functional attribute, for examples the complexity analysis. The later one uses the quantitative results form some simulations on that said alternative under the considered scenarios. In addition, in cases where the techniques of a primitive is difficult to be assessed in neither ways, then the software engineer could score the alternative using empirical knowledges [4] by assigning weights based on experience. It is worth noting that in cases where the non-functional attributes can only be assessed using multiple primitives, then when scoring the alternatives of a primitive, it is important to ensure that the uses of alternative for other primitives are equivalent. For example, when scoring the alternatives of stimulus-awareness in terms of system's overall expression quality, then the chosen alternative for self-expression should be consistent. Precisely, the scoring process has three phases:

Firstly, weight the relative importance of different non-functional attributes after negotiation amongst the stakeholders. In particular, this can be achieved using the pair-wise comparison in AHP [18]. This alternative can be used to measured how much importance of attributes Q_a is over the attribute Q_b based on some scale.

Secondly, score each alternative against each non-functional attribute under every defined scenarios. Having this done, we then have a matrix of n times m for the hth primitive under a given non-functional attribute k, as shown below:

$$P_h = \begin{array}{cc} & SC_1, ..., SC_m \\ \begin{array}{c} A_1 \\ ... \\ A_n \end{array} & \begin{pmatrix} S^k_{11} & ... & S^k_1 m \\ ... & ... & ... \\ S^k_{n1} & ... & S^k_{nm} \end{pmatrix} \end{array} \tag{3.1}$$

where n is the total number alternatives of the *h-th* primitive P_h and m denotes the total number of considered scenarios for non-functional attribute k. SC_m denote the *m-th* scenario; A_n mean the *nth* alternative and S^k_{nm} denote the score of the *n-th* alternative under the *m-th* scenario. In cases where more than one scenario for a non-functional attribute, the score of a alternative would the aggregative result of the scores under all scenarios. Therefore, we calculate the total score of an alternative for the *h-th* primitive under non-functional attribute k by aggregating its score of all scenarios. The matrix can be reduced to a vector as shown below:

$$P_h = \begin{array}{c} A_1 \\ ... \\ A_n \end{array} \begin{pmatrix} S^k_1 \\ ... \\ S^k_n \end{pmatrix} \quad s.t. \;\; S^k_n = \sum_{x=1}^{m} S^k_{nx} \tag{3.2}$$

where S^k_n is the aggregative score for the *nth* alternative of the *hth* primitive under non-functional attribute k.

Thirdly, we normalise the raw score for the alternatives in each architectural primitive against a non-functional attribute k. To achieve this, we use the following formula:

$$the\, A_a\, of\, primitive\, P_h = \begin{cases} NS^k_{ah} & if\, max Q_k \\ 1 - NS^k_{ah} & if\, min Q_k \end{cases} \quad s.t. \;\; NS^k_{ah} = \frac{S^k_a}{\sum_{x=1}^{p} S^k_x} \tag{3.3}$$

where is NS^k_{ah} the normalised score for the *a-th* alternative of the *h-th* primitive; p denotes the total number of alternatives for the *h-th* primitive. By doing so, we can normalise the score scaling from 0 to 1. In case where the attribute is to be minimised, the final normalised score would be calculated as $1 - NS^k_{ah}$.

3.1.8 Step 8 - Find the Best Alternatives for the Final Architecture View

Once all the scores of alternatives have been obtained, the final task to do is to find out the best combination that produces the highest total score. Specifically, we need to maximizing the formula below:

$$argmax \sum_{k=1}^{x}(w_k \times \sum_{h=1}^{y} NS_{sh}^{k}) \tag{3.4}$$

where NS_{sh}^{k} is the score of the *s-th* selected alternatives for the *h-th* primitive; w_k is the weight for the *k-th* non-functional attribute. x and y are the total number of non-functional attributes and architectural primitives respectively. It is worth noting that the solution needs to refers to the dependency of primitives, e.g., one can not select a technique for meta-self-awareness if it has been decided not to use this capability. Equation 3.4 can be solved by any optimisation algorithms, and finally our methodology provides a pattern based architecture with the best selected alternatives for all architectural primitives.

3.2 Qualitative and Quantitative Evaluation

We evaluate the methodology qualitatively using two scenarios: a cloud autoscaling case study and a smart camera networks case study. We also quantitatively assess the resulted system by comparing against a non-self-aware system for both case studies.

3.2.1 Cloud Autoscaling Case Study

Introduction and Background

In cloud computing paradigm, the cloud-based services are deployed as Software as-a-Service (SaaS) and are typically supported by the software stack in the Platform as-a-Service (PaaS) layer [2]. They are also supported with Virtual Machines (VM) and hardware within the Infrastructure as-a-Service (IaaS) layer [1]. Under changing environmental conditions (e.g., workload, size of incoming job etc.), it is important to manage and control the Quality of Service (QoS) of cloud-based services. By QoS, we refer to the non-functional attributes (e.g., throughput) experienced by the end-users who use these services.In particular, the QoS can be managed by various control knobs, which include software (e.g., threads) and hardware resources (e.g., CPU) in a shared infrastructure. Hohowever, inappropriate use of software and hardware resources could result in large rental cost to the service. In this report, we refer to these control knobs and environmental conditions in the cloud as primitives.

With the context in mind, the term elasticity [13] in cloud refers to the ability to adaptively scale control knobs to match the demand of cloud-based services at runtime. Given the uncertainty and dynamics of QoS, there is an increasing demand on cloud where the realisation of elasticity can be managed without human intervention. In particular, for all cloud-based services, the cloud should dynamically and continuously select an elastic strategy, which is the combinatorial decision of configurations for various control knobs; this process is called autoscaling.

Autoscaling is an important mechanism to realise elasticity in the cloud. We argue that the autoscaling should be cost and QoS optimised; more precisely, upon each provisioning and de-provisioning process, our aim is to design an autoscaling mechanism that adaptively optimise QoS attributes and cost (i.e., cost of software and hardware resources) for **all** cloud-based applications and services at runtime by autoscaling to the best combinatorial values of control knobs. Due to the on-demand and dynamic nature of cloud, human intervention and traditional analytical approaches are limited to achieve this goal. Therefore, we intend to build a self-aware and self-expressive system to perform efficient and intelligent autoscaling. To this end, this system should dynamically model QoS, which could use the primitives as inputs and predict the likely QoS value as an output. These models can better express QoS sensitivity. By sensitivity, we are interested in *which* (e.g., are throughput and CPU correlated?), *when* (i.e., at which point in time they are correlated?) and *how* (i.e., the magnitude of primitives in correlation) the primitives correlate with QoS. In particular, the system mustconsider dynamics and uncertainty caused by workload and QoS interference (both service-level and VM-level). By QoS interference, we refer to scenarios where a service suffers wide disparity in its QoS depends on the fluctuated primitives of co-located services on the same VM and co-hosted VMs on the same Physical Machine (PM). This is a typical consequence of resources contention. In addition, the system shall also take objective-dependency into account as the objectives of a cloud-based service could be either conflicted or harmonic with the objectives form the same service (intra-service dependency) or the other co-located services (inter-service dependency).

Terminology

We advocate a fine-grained approach to the modelling and analysis of QoS. To achieve this, we decompose the notion of primitives into two major categories: these are **Environmental Primitives (EP)** and **Control Primitives (CP).** We posit that CP can be either software or hardware, which could be managed by cloud providers to support QoS provisioning. In particular, software control primitives are software tactics and configurations; such as the number of threads in thread pool and its life time, the number of connections in database connection pool, security and load balancing policies etc. Whereas, hardware control primitives are computational resources provisioning, such as CPU, memory and bandwidth. Software and hardware control primitives rely on the PaaS and IaaS layers respectively. In particular, it is a non- trivial task to consider software control primitives when QoS modelling in the cloud as they tend to influence QoSs significantly. On the other hand, we look at environmental primitives in the context of highly dynamic scenarios, which reflect the cloud setting. The environmental primitives can significantly influence the QoS. The providers often can not predict and fully control their behaviour. Examples include unbounded workload and unpredictable bound received data etc. If the provider would be able to predict and control the presence of these scenarios, these can be then considered as control primitives.

We assume that cloud-based applications are composed of one or more services, each with its QoS requirements and can experience different environmental changes (e.g., changes in workload). These services are deployed on a cloud software stack, which can be setup using various configurations and tactics. In addition, they are hosted on the cloud infrastructure, where resources are shared via VMs. As a result, the control knobs and environmental conditions could significantly influences their QoSs. In distributed environment like cloud, each tier in a multi-tiers application, composed of concrete services $\{S_1, S_2, \ldots S_i\}$ may have multiple replicas deployed on different VMs. The replica of a tier running on a VM is assumed to have the replicas of its services running on the same VM. In this work, we refer to the replicas of concrete services as service-instances: the *j-th* service-instance of the *i-th* concrete service is denoted by S_{ij}. Unlike existing work, which focus on realising elasticity at the application and VM level, we aim to adaptively optimise the QoS attributes and rental cost of utilising control knobs for each individual service-instance, considering the QoS interferences caused by the co-located service-instances on a VM and the co-hosted VMs on a PM.

In addition, we do not consider global resources contention caused by shortage in cloud capacity; our architecture works for cases where software and hardware resources tend to be available, which is normal in a cloud environment. Henceforth, we assume that the maximum demand of software and hardware resources for all cloud service-instances (e.g., according to their budget) should be satisfied by the capability of the cloud provider. Under such assumption, we eliminate extreme cases where the capacity of cloud provider reaches its limits causing likely global resources contention. This is because the increasing demand of each service-instance would eventually be satisfied by scale up/out as long as the cost does not exceed the budget. We believe this is a reasonable assumption as in realistic scenarios, proper admission control can be applied to restrict the number of cloud-based service-instances. Moreover, in case where the cloud provider actually encounters capacity shortage, the unsatisfied services can be switched to an alternative provider via a cloud selection mechanism, which presumably hold our assumption. However, the design of admission control and selection mechanism is outside the scope of this work.

Objective and Models

We formulate an "online" QoS model, which captures both dynamic sensitivity and interference with respect to the selectedprimitives over time. The model at given sampling interval t is formally expressed as:

$$QoS_k^{ij}(t) = f(SP_k^{ij}(t), \delta) \tag{3.5}$$

where $QoS^{ij}_{k}(t)$ is the average value of *k-th* QoS of S_{ij} at interval *t*. *f* is the QoS function, which dynamically changes at runtime. δ refers to any other inputs that are required by the algorithm to train f apart from the primitives. Examples of other inputs may include historical QoS values and tuning variables. To handle QoS interferences, we denote the input $SP^{ij}_{k}(t)$ of Eq. 3.5 as the selectedprimitives matrix of $QoS^{ij}_{k}(t)$ at interval *t*. This matrix contains the selectedprimitive inputs of $QoS^{ij}_{k}(t)$ and it is updated online.

In the context of cloud, utilizing control primitives may be subject to certain monetary cost to the service owners, therefore the total costs model for S_{ij} can be represented as:

$$Cost^{ij} = \sum_{a=1}^{n} g(CP^{ij}_{a}(t), P_a) \tag{3.6}$$

where g is the fixed and unified cost function for each type of control primitives, and n is the total number of control primitive type that used by service-instance S_{ij} to supports its QoS attributes. $CP^{ij}_{a}(t)$ is the amount of the *a-th*control primitiveprovision for S_{ij} at interval t. P_a denotes the corresponding price per unit of the *a-th* control primitive. In this work, we assume that the price of each control primitivetype is fixed for all service providers and their service-instances. It is worth noting that the hardware control primitives (e.g., CPU and memory) can be only provisioned for each VM whereas the cost model is per-service based, thus the price of a hardware control primitiveshould be equally proportioned to each of the service-instances deployed on the provisioned VM.

To achieve globally-optimal QoS and cost in elastic cloud via autoscaling, we aim at adaptively and dynamically determine and scale to the control primitive configurations, which supports the best of all QoS attributes (Eq. 3.5) with minimal costs (Eq. 3.6) for all service-instances in the cloud. In this work, we apply a linear weighted-sum aggregation to express the global result for QoS attributes and costs of different service-instances in the cloud. Formally, at any given interval t, we aim to optimise the global objective by maximising the function in Eq. 3.7.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} w'_{ij} \cdot (\sum_{a}^{l} w_a \cdot QoS^{ij}_{a}(t) - \sum_{b}^{r} w_b \cdot QoS^{ij}_{b}(t) - w_{(l+r+1)} \cdot Cost^{ij}) \tag{3.7}$$

where *n*and *m*are the total number of services and their instances in the cloud; w'_{ij} is the weight for each service-instance. Because the global objective is to maximise Eq. 3.7, we need to carefully place the maximised QoS (e.g., throughput) and the minimised ones (e.g., response time); thus *l*and *r*are the total number of the maximised and minimised QoS for S_{ij} respectively; w_a, w_b and $w_{(l+r+1)}$are refer to the corresponding weight of the QoS and cost for S_{ij}. In addition, the optimisation of Eq.3.7 should be subject to the constraint of budget and SLA

In the following, we qualitatively evaluate the proposed methodology by showing how it can be applied to engineer self-aware and self-expressive system in the cloud case study. We also show the experiments that compare the resulted system with a non-self-aware system. To simplify the explanation, we only consider the throughput and cost of cloud-based services as the objectives that need to be maintained.

Step 1 - Collect Requirements and Constraints

After negotiated with the stakeholders and analysed the environment, the requirements and constraints are presented as the following Table:

Table 3.2: The functional, non-functional requirements and constraints for the cloud case study.

Functional Requirements	**Explanation**
The system must record historical data for analysing the behaviours of cloud-based services.	The data could be logs or real time measurement collected by the sensors.
The system must able to control both software and hardware control primitives.	Both types of control knobs could influence QoS significantly.
The system must be aware of the changes in workload and deployment.	This is the cause of dynamic and uncertainty in cloud
The system should be able to aware of QoS interference.	This could influence the autoscaling decisions.
The system should support both vertical and horizontal scaling.	Both actions could help to improve QoS.
The system must cope with the conflicted objectives.	This could influence the autoscaling decisions.
The system must aware of the functional dependency between cloud-based services across nodes.	This could influence the autoscaling decisions.
The system should be able to cope with any given QoS attributes and cost objective of cloud-based services.	
The system should be able to cope with any runtime changes of QoS and cost objectives made by the service providers.	This is something could occur in the cloud.
Non-functional Requirements	
Accuracy: the accuracy of QoS modelling should be no less than 75% to the actual QoS value.	This could influence the effectiveness of autoscaling decisions.
Adaptation Quality: the QoS of the cloud-based services that being managed should not be worse than 20% of the threshold in SLA for more than 5 mins.	This is only applicable in case where the budget is allowed.
Overhead: the overhead for making autoscaling decision should be less than 200 seconds.	This could influence the effectiveness of autoscaling decisions.
Reliability: to what extent the designer believe that the alternative would work at runtime when there is an emergent scenario occurs?	This can assess the confidence of simulation under unknown events at runtime (e.g., different workload)
Constraints	
VM can be added or removed	Due to the dynamic cloud
Service can be added or removed	Due to the dynamic cloud
Workload for each cloud-based service is fluctuated.	Due to the dynamic cloud
The cost of the cloud-based services that being managed should not exceed its defined budget.	Should not let the cloud consumer pay more than they would like, even with the cost of worse QoS.
QoS interference occurs once there are contentions, in which case the QoS could be negatively affected.	Could caused by workload or improper configuration of control primitives.

Step 2 - Propose Candidate Architecture

Based on the aforementioned requirements and constraint, we scratch the an initial version of architectural view, as illustrated in the Figure below:

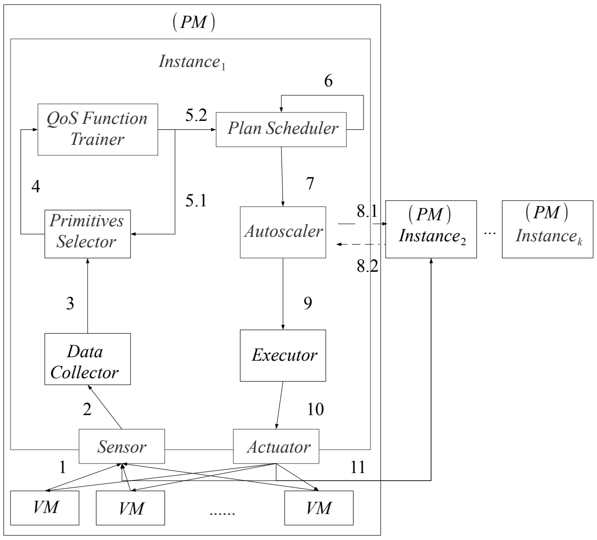


Figure 3.2: The proposed architecture

As we can see that the architecture is deployed as distributed instances, each of which running on a separate VM (e.g., *Dom0* on Xen [3]) on every PM (node) in the cloud. The workflow of the proposed architecture has been shown in Figure 3.2. More precisely, the *sensor* on each PM collects the data (e.g., QoS values, CP usages and EP values) from the underlying VMs and service-instances; and possibly from other PMs due to functional dependency (step 1). In addition, the sensor could sense deployment changes and QoS sensitivity changes from other PMs. Next in step 2, the sensor passes raw information it received to the *Data Collector* for normalising the data. At step 3, the *Primitives Selector* receives both current and historical data after normalisation, which would be used to determine the inputs of models. The *QoS Function Trainer* would be used to train the function (step 4-5.1). Once QoS models have been generated, the propagation goes to step 5.2. In particular, the adaptation can be triggered if one or more of the following symptoms is detected:

- Symptoms 1: Proactively detect if the QoS of a service- instance is likely to violate SLA constraint for k intervals by using the QoS models.

- Symptoms 2: Reactively detect if the QoS of a service- instance has violated its SLA constraint for k intervals and/or if the utilisation of a CP has violated the constraint for k intervals.
- Symptoms 3: occurrence of QoS sensitivity changes and deployment changes.

All symptoms are handled by the *Plan Scheduler* component, which would be responsible for deciding whether to trigger the Autoscaling decision making process or the area of effect caused by the violations (step 6). The *Autoscaler* component is designed to dynamically search the best adaptation strategies toward the optimal result, using the QoS and cost models (step 7). In particular, the *Autoscaler* of each node is triggered independently and asynchronously. There are cases where the optimisation for a better autoscaling decision need to communicate (for obtaining external QoS models) with other nodes because of the functional dependency between services. In addition, it is critical to ensure the same objective is not optimised simultaneously on more than one nodes. These processes are expressed as step 8.1 and 8.2.

Once the elastic strategy is determined, the process proceeds to the *Executor* via step 9. In particular, The *Executor* is responsible for determining which concrete actions (e.g., scale up/down, in/out and/or VM migration and replication etc) need to be taken in order to fulfil the elastic strategy. In this work, we consider both vertical and horizontal scaling and apply a simple solution to determine the actions, this is: we always try vertical scaling (i.e., scale up/down) first before horizontal scaling (i.e., scale out/in). This is because horizontal scaling is usually more expensive than vertical scaling. As for the VM migration/ replication decision, we always choose the one that result in smaller overhead based on a predefined VM profiling pattern. Finally, the actions are taken by the *Actuator* via step 10 and 11.

Step 3 - Select the Best Pattern(s)

We now select the pattern using the questions presented previously for each self-awareness and self-expression capability:

Table 3.3: Questions and answers for deciding whether to include stimulus-awareness.

Stimulus-awareness	
What does the capability mean in your problem context.	The ability to aware of newly-measured data (data for environmental primitives and control primitives), violations of QoS and utilisation thresholds, QoS sensitivity changes and deployment changes.
What are the functional requirements that affected by this capability?	1. The system must be aware of the changes in workload and deployment. 2. The system should be able to aware of QoS interference. 3. The system must aware of the functional dependency between cloud-based services.
What are the non-functional requirements that affected by this capability?	1. Accuracy 2. Overhead 3. Adaptation Quality
What are the constraints that could affect this capability?	1. VM can be added or removed. 2. Service can be added or removed. 3. Workload for each cloud-based service is fluctuated. 4. QoS interference occurs once there are contentions, in which case the QoS could be negatively affected.
Whether this capability is necessary or beneficial?	Yes

Table 3.4: Questions and answers for deciding whether to include time-awareness.

Time-awareness	
What does the capability mean in your problem context.	The ability to aware of historical behaviours of cloud-based services, the continuous consequences of autoscaling decisions and the emergent events that occurred in the past.
What are the functional requirements that affected by this capability?	1. The system must record historical data for analysing the behaviours of cloud-based services.
What are the non-functional requirements that affected by this capability?	1. Accuracy 2. Adaptation Quality 3. Overhead
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

Table 3.5: Questions and answers for deciding whether to include interaction-awareness.

Interaction-awareness	
What does the capability mean in your problem context.	The ability to aware of the state (e.g., QoS models) of other nodes because of functional dependency; and also the possible internal interactions of local services.
What are the functional requirements that affected by this capability?	1. The system must aware of the functional dependency between cloud-based services across nodes.
What are the non-functional requirements that affected by this capability?	1. Accuracy 2. Adaptation Quality 3. Overhead
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

Table 3.6: Questions and answers for deciding whether to include goal-awareness.

Goal-awareness	
What does the capability mean in your problem context.	The ability to aware of the QoS and cost objectives of cloud-based services. In addition, it should also aware of the changes.
What are the functional requirements that affected by this capability?	1. The system must aware of the functional dependency between cloud-based services. 2. The system should be able to cope with any given QoS attributes and cost objective of cloud-based services. 3. The system should be able to cope with any runtime changes of QoS and cost objectives made by the service providers.
What are the non-functional requirements that affected by this capability?	1. Adaptation Quality 2. Overhead
What are the constraints that could affect this capability?	1. The cost of the cloud-based services that being managed should not exceed its defined budget.
Whether this capability is necessary or beneficial?	Yes

Table 3.7: Questions and answers for deciding whether to include self-expression.

Self-expression	
What does the capability mean in your problem context.	The ability to change the configurations, tactics and resource provisioning of a node.
What are the functional requirements that affected by this capability?	1. The system must able to control both software and hardware control primitives. 2. The system should support both vertical and horizontal scaling.
What are the non-functional requirements that affected by this capability?	1. Adaptation Quality 2. Overhead
What are the constraints that could affect this capability?	1. VM can be added or removed. 2. Service can be added or removed.
Whether this capability is necessary or beneficial?	Yes

Table 3.8: Questions and answers for deciding whether to include met-self-awareness.

Meta-self-awareness	
What does the capability mean in your problem context.	The ability to improve the goal-awareness by either dynamically finding the best technique or using ensemble methods.
What are the functional requirements that affected by this capability?	1. The system must be aware of the changes in workload and deployment. 2. The system should be able to aware of QoS interference. 3. The system must aware of the functional dependency between cloud-based services.
What are the non-functional requirements that affected by this capability?	1. Accuracy 2. Reliability 3. Adaptation Quality 4. Overhead
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

According to the above analysis and the aforementioned Table 3.1, the selected pattern for this cloud case study is the *Goal Sharing with time-awareness capability Pattern.*

Step 4 - Fit the Selected Pattern(s)

We now fit the proposed architecture to the selected pattern, as shown in the Figure 3.3 below.

Step 5 - Determine the Important Primitives and the Possible Alternatives for Non-functional Requirements

It is worth noting that certain primitives are eliminated form consideration as they are trivial in this problem context, these are: *transit, link* and*structure. Transit*is eliminated because all the capabilities are associated with at least one functional requirements, therefore we do not need to consider the possibility of switch on/off certain capabilities at runtime in this case. As for *link*, we consider that the topology of components is constrained by the environment and functional requirements, thus it does not significantly influence the non-functional requirements. Similarity, the *structure* primitive is eliminated because how the capabilities are distributed into components is not significantly associated with the non-functional requirements in our case.

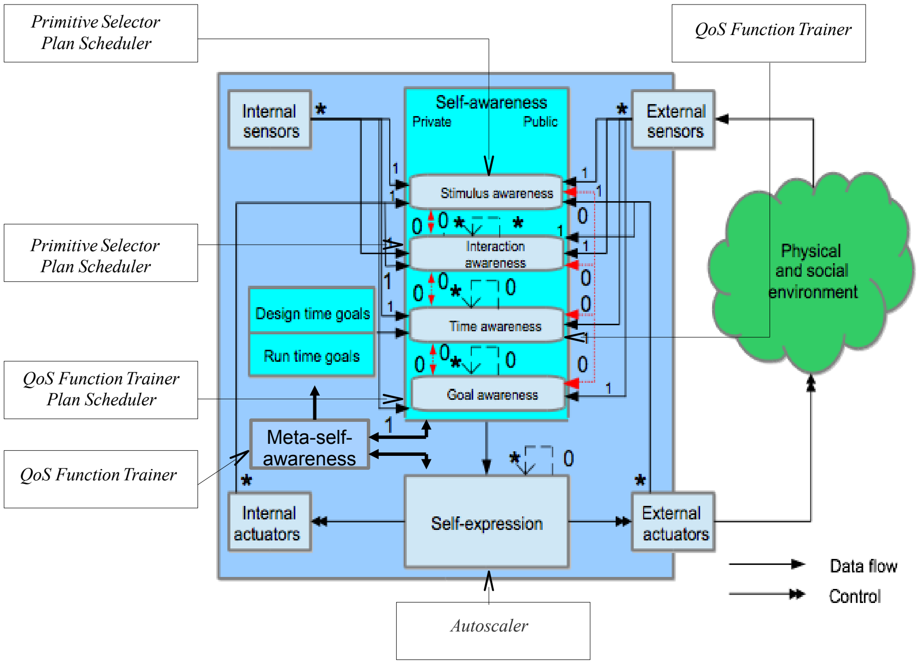


Figure 3.3: Fit the proposed architecture to Goal Sharing with time-awareness capability Pattern

In addition, we eliminated some techniques as they are fundamentally not applicable in our case due to the constraints and non-functional requirements. Precisely, the Table below list the rest architectural primitives and the possible alternatives for this case study:

Table 3.9: The chosen architectural primitives and their alternatives for the cloud case study.

Architectural Primitives	**Alternatives**
stimulus-awareness	Symmetric Uncertainty Measurement, Queuing Network, Queuing Network +Simple Update Function, Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement, Symmetric Uncertainty Measurement+ Conditions-Actions Rule, Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning
interaction-awareness	Threshold-based Algorithm, Symmetric Uncertainty Measurement, Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement
time-awareness	Linear ARMAX, Neural Network, Regression Tree, Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)
goal-awareness	Linear ARMAX, Neural Network, Regression Tree, Sensitivity and Region-based Partitioning+Linear ARMAX, Sensitivity and Region-based Partitioning+Neural Network, Sensitivity and Region-based Partitioning+Regression Tree, Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)
self-expression	Random Optimisation, Static Mapping, Brute Force Optimization,
meta-self-awareness	Bucket of Models, Ensemble Method
send	synchronous function call+asynchronous multicast, asynchronous function call+asynchronous multicast
handle	First-Come-First-Serve (FCFS), multi-threading, First-Come-First-Serve (FCFS)+multi-threading
state	proactive Goal-awareness, reactive Goal-awareness, proactive+reactive Goal-awareness
existence	exist, non-exist

Step 6 - Create Scenarios

We now defined some scenarios for each non-functional attribute. In particular, the accuracy analysis uses 3 scenarios represents the most common facts in the cloud:

- The cloud-based services are under different level of burst workload.
- QoS interference occurs due to contention.
- There are VM migration/replication taken place due to actuations.

For adaptation quality analysis, we use 3 scenarios to assess how the self-aware and self-expressive system behaves.

- There are some amount of conflicted and harmonic objectives of different cloud-based services.

- There are some amount of conflicted and harmonic objectives of different cloud-based services.
- The cloud-based services are under different level of burst workload.

For overhead analysis, we assume 2 scenarios, representing the anticipated way in which the system could suffer from overhead.

- More than one cloud-based services located on each VM.
- More than one VMs hosted on each node.

For the reliability analysis, we only use one scenario. In particular, this attribute is measured by empirical method [4] instead of simulation models as it involved unknown workload.

- The cloud-based services are under unknown workload and/or events

Step 7 - Score the Alternative of Primitives Against each Non-functional Attribute using Analytical or Simulation Models

According to the steps mentioned previously, we first weight the relative importance of different non-functional attributes after negotiation amongst the stackholders. The results are shown as below:

Table 3.10: The weights of different non-functional attributes for the cloud case study.

Attribute	**Weight**
Accuracy	0.1
Adaptation Quality	0.75
Overhead	0.05
Reliability at runtime	0.1

Secondly, we score each alternative of all primitives against each non-functional attribute under every defined scenarios. In particular, we run simulation for each alternative under the aforementioned scenarios. The results are shown as below (the score of 0 means they have no or limited impact to the non-functional attribute):

Table 3.11: The scores of different alternatives for accuracy.

Accuracy (%)			
Alternative	**Scenario 1**	**Scenario 2**	**Scenario 3**

stimulus-awareness	Symmetric Uncertainty Measurement	84.1	84.5	85.7
	Threshold-based Algorithm	54.3	39.5	55.4
	Simple Update Function	56.4	40.7	56.2
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	84.1	84.5	85.7
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule	84.1	84.5	85.7
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	84.1	84.5	85.7
interaction-awareness	Conditions-Actions Rule	0	0	0
	Symmetric Uncertainty Measurement	84.1	84.5	85.7
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	84.1	84.5	85.7
time-awareness	Linear ARMAX	83.3	82.1	80.2
	Neural Network	85.4	85	86.1
	Regression Tree	75.4	70.2	73.3
	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	84.1	84.5	85.7
goal-awareness	Linear ARMAX	83.3	82.1	80.2
	Neural Network	85.4	85	86.1
	Regression Tree	75.4	70.2	73.3
	Sensitivity and Region-based Partitioning+Linear ARMAX	83.3	82.1	80.2
	Sensitivity and Region-based Partitioning+Neural Network	85.4	85	86.1
	Sensitivity and Region-based Partitioning+Regression Tree	75.4	70.2	73.3
	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	84.1	84.5	85.7
self-expression	Random Optimization	0	0	0
	Static Mapping	0	0	0
	Brute Force Optimization	0	0	0

meta-self-awareness	Bucket of Models	84.1	84.5	85.7
	Ensemble Method	78.6	81.1	80.5
send	synchronous function call+asynchronous multicast	0	0	0
	asynchronous function call+asynchronous multicast	0	0	0
handle	First-Come-First-Serve (FCFS)	0	0	0
	multi-threading	0	0	0
	First-Come-First-Serve (FCFS)+multi-threading	0	0	0
state	proactive Goal-awareness	0	0	0
	reactive Goal-awareness	0	0	0
	proactive+reactive Goal-awareness	0	0	0
existence	exist	84.1	84.5	85.7
	non-exist	80.3	76.1	79.2

Table 3.12: The scores of different alternatives for adaptation quality.

Adaptation Quality (calculated by Eq 7.)				
Alternative		**Scenario 1**	**Scenario 2**	**Scenario 3**
stimulus-awareness	Symmetric Uncertainty Measurement	5.7	4.2	4.7
	Threshold-based Algorithm	4.8	4.1	4.5
	Simple Update Function	5.2	4.7	3.9
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	5.8	4.4	4.8
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule	5.8	4.5	4.7
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	6.2	5.6	4.9
interaction-awareness	Conditions-Actions Rule	4.3	4	3.5
	Symmetric Uncertainty Measurement	5.7	4.2	4.7
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	5.9	5.2	4.9

time-awareness	Linear ARMAX	4.3	4.5	3.9
	Neural Network	6	5.1	5.2
	Regression Tree	4	4.4	4.5
	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	5.8	4.8	5.1
goal-awareness	Linear ARMAX	4.3	4.5	3.9
	Neural Network	6.1	5.2	5
	Regression Tree	4	4.4	4.5
	Sensitivity and Region-based Partitioning+Linear ARMAX	4.4	4.8	4.1
	Sensitivity and Region-based Partitioning+Neural Network	5.9	5.3	5.2
	Sensitivity and Region-based Partitioning+Regression Tree	4.5	4.8	4.6
	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	5.8	4.8	5.1
self-expression	Random Optimization	4.8	4.9	4.8
	Static Mapping	4.7	4.6	4.9
	Brute Force Optimization	5	5.2	5.1
meta-self-awareness	Bucket of Models	5.5	5.4	5.3
	Ensemble Method	5.9	5.1	4.9
send	synchronous function call+asynchronous multicast	0	0	0
	asynchronous function call+asynchronous multicast	0	0	0
handle	First-Come-First-Serve (FCFS)	0	0	0
	multi-threading	0	0	0
	First-Come-First-Serve (FCFS)+multi-threading	0	0	0
state	proactive Goal-awareness	4.6	4.5	4.1
	reactive Goal-awareness	4.8	4.9	3.9
	proactive+reactive Goal-awareness	5	5.1	5
existence	exist	5.8	5.3	5.1
	non-exist	4.1	4.5	4.1

Table 3.13: The scores of different alternatives for overhead.

Overhead (s) (instead of measure the overhead of the whole system, we measure the overhead of each alternative)			
Alternative		**Scenario 1**	**Scenario 2**
stimulus-awareness	Symmetric Uncertainty Measurement	1.2	1.3
	Threshold-based Algorithm	0.7	0.6
	Simple Update Function	0.7	0.6
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	1.7	1.5
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule	1.2	1.3
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	1.7	1.5
interaction-awareness	Conditions-Actions Rule	0.8	0.8
	Symmetric Uncertainty Measurement	1.2	1.3
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	1.7	1.4
time-awareness	Linear ARMAX	1.2	1.2
	Neural Network	35.4	27.9
	Regression Tree	5.1	3.6
	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	41.2	33.3
goal-awareness	Linear ARMAX	1.2	1.2
	Neural Network	35.4	27.9
	Regression Tree	5.1	3.6
	Sensitivity and Region-based Partitioning+Linear ARMAX	2.7	2.3
	Sensitivity and Region-based Partitioning+Neural Network	36.9	29
	Sensitivity and Region-based Partitioning+Regression Tree	6.6	4.7
	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	42.7	34.4

self-expression	Random Optimization	55.4	49.3
	Static Mapping	135.3	118.2
	Brute Force Optimization	154.3	144.2
meta-self-awareness	Bucket of Models	5.5	5
	Ensemble Method	7.8	6.3
send	synchronous function call+asynchronous multicast	37.3	35.3
	asynchronous function call+asynchronous multicast	11.4	13.2
handle	First-Come-First-Serve (FCFS)	35.5	37
	multi-threading	12.3	10
	First-Come-First-Serve (FCFS)+multi-threading	18.8	23.7
state	proactive Goal-awareness	0	0
	reactive Goal-awareness	0	0
	proactive+reactive Goal-awareness	0	0
existence	exist	41.2	33.3
	non-exist	35.4	27.9

Table 3.14: The scores of different alternatives for reliability.

Reliability (relative weights)		
Alternative		**Scenario 1**
stimulus-awareness	Symmetric Uncertainty Measurement	5
	Threshold-based Algorithm	1
	Simple Update Function	1
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	5
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule	7
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	7
interaction-awareness	Conditions-Actions Rule	1
	Symmetric Uncertainty Measurement	7

	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	7
time-awareness	Linear ARMAX	1
	Neural Network	1
	Regression Tree	1
	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	7
goal-awareness	Linear ARMAX	1
	Neural Network	1
	Regression Tree	1
	Sensitivity and Region-based Partitioning+Linear ARMAX	1
	Sensitivity and Region-based Partitioning+Neural Network	1
	Sensitivity and Region-based Partitioning+Regression Tree	1
	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	7
self-expression	Random Optimization	5
	Static Mapping	1
	Brute Force Optimization	5
meta-self-awareness	Bucket of Models	1
	Ensemble Method	1
send	synchronous function call+asynchronous multicast	1
	asynchronous function call+asynchronous multicast	1
handle	First-Come-First-Serve (FCFS)	1
	multi-threading	1
	First-Come-First-Serve (FCFS)+multi-threading	1
state	proactive Goal-awareness	1
	reactive Goal-awareness	1
	proactive+reactive Goal-awareness	1
existence	exist	1
	non-exist	1

Once we obtain all the scores and calculate the total score for all scenarios, we then normalised the scores using Eq. 3.3, the results are shown in Table 3.15.

Table 3.15: The normalised scores of different alternatives for all non-functional attributes.

Alternative		**Accuracy**	**Adaptation Quality**	**Overhead**	**Reliability**
stimulus-awareness	Symmetric Uncertainty Measurement	0.192	0.165	0.821	0.19
	Threshold-based Algorithm	0.112	0.151	0.907	0.04
	Simple Update Function	0.116	0.156	0.907	0.04
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	0.192	0.169	0.771	0.19
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule	0.192	0.169	0.821	0.27
	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	0.2	0.189	0.771	0.27
interaction-awareness	Conditions-Actions Rule	0	0.278	0.777	0.07
	Symmetric Uncertainty Measurement	0.5	0.344	0.653	0.47
	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	0.5	0.377	0.569	0.47
time-awareness	Linear ARMAX	0.252	0.22	0.984	0.1
	Neural Network	0.263	0.283	0.575	0.1
	Regression Tree	0.224	0.224	0.942	0.1
	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	0.261	0.273	0.5	0.7
goal-awareness	Linear ARMAX	0.145	0.125	0.99	0.08
	Neural Network	0.151	0.161	0.729	0.08
	Regression Tree	0.129	0.127	0.963	0.08
	Sensitivity and Region-based Partitioning+Linear ARMAX	0.145	0.131	0.979	0.08
	Sensitivity and Region-based Partitioning+Neural Network	0.151	0.162	0.718	0.08
	Sensitivity and Region-based Partitioning+Regression Tree	0.129	0.137	0.952	0.08
	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	0.15	0.155	0.670	0.54
self-expression	Random Optimization	0	0.33	0.841	0.45
	Static Mapping	0	0.323	0.614	0.09
	Brute Force Optimization	0	0.348	0.545	0.45
meta-self-awareness	Bucket of Models	0.514	0.509	0.573	0.5
	Ensemble Method	0.486	0.491	0.427	0.5
send	synchronous function call+asynchronous multicast	0	0	0.253	0.5
	asynchronous function call+asynchronous multicast	0	0	0.747	0.5
handle	First-Come-First-Serve (FCFS)	0	0	0.472	0.33

Step 8 - Find the Best Alternatives for the Final Architecture View

Finally, we search for the alternative that resulting the highest value using Eq. 3.4. The final output of the selected alternatives is list as in Table 3.16.

Table 3.16: The normalised scores of selected alternatives for all non-functional attributes.

Selected Alternative		**Accuracy**	**Adaptation Quality**	**Overhead**	**Reliability**
stimulus-awareness	Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	0.2	0.189	0.771	0.27
interaction-awareness	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	0.5	0.377	0.569	0.47
time-awareness	Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	0.261	0.273	0.5	0.7
goal-awareness	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree (need meta-self-awareness)	0.15	0.155	0.670	0.54
self-expression	Random Optimization	0	0.33	0.841	0.45
meta-self-awareness	Bucket of Models	0.514	0.509	0.573	0.5
send	asynchronous function call+asynchronous multicast	0	0	0.747	0.5
handle	multi-threading	0	0	0.838	0.33
state	proactive+reactive Goal-awareness	0	0.360	0	0.33
existence	exist	0.519	0.541	0.46	0.5

This selection gives us the highest score of 15.437 according to Eq 4. Once we combine the results with those primitives, which were eliminated at the beginning of this step, the detailed variation of our architectural instance based on the Goal Sharing Pattern with time-awareness is shown in the Table 3.17.

Table 3.17: The selected alternatives for the cloud case study.

		stimulus-awareness	interaction-awareness	time-awareness	goal-awareness	self-expression	meta-self-awareness	sensor	actuator
Selected alternative(s)		Symmetric Uncertainty Measurement+ Conditions-Actions Rule+ Sensitivity and Region-based Partitioning	Sensitivity and Region-based Partitioning+Symmetric Uncertainty Measurement	Linear ARMAX+Neural Network+ Regression Tree	Sensitivity and Region-based Partitioning+Linear ARMAX+Neural Network+ Regression Tree	Random Optimization	Bucket of Models		
send	synchronous	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	asynchronous	function call	function call	function call	function call	function call	function call	function call	multicast
handle	sequential	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	parallel	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading
state		reactive	reactive	reactive	proactive and reactive	proactive	proactive		
transit							N/A		
link		one-to-one	many-to-many, one-to-one	one-to-one	one-to-one	one-to-many	one-to-many		
structure		combine+ separate	combine+ separate	combine	combine+ separate	compact	combine	compact	compact
existence							exist		

Quantitative Experiments

In this section, we conduct quantitative evaluation by experimenting our self-aware and self-expressive system against a non self-aware system, which adapts simple rule-based policies. We primary assess the adaptation quality for cloud-based services under the management of these two systems. The observed adaptation quality is measured by score, which is the average result calculated by Eq. 3.7 for the interval after a previous elasticity decision point and before the next one. Each of these intervals is referred to as effect point.

To evaluate global benefit of the elastic strategies produced by our architecture and the overhead for reaching these strategies, we have conducted an experimental evaluation. In particular, we have implemented the architecture prototype using Java JDK1.6, and we assessed the elastic scaling of 8 hypothetical cloud-based service-instances under the control of our prototype. In the experiment setup, each service-instance was deployed on software stack including Apache, Tomcat and MySQL. We simulate a synthetical workload to each service-instance. The workload has been designed in a way that the intensity was sufficient for causing QoS interference on the co-located services and co-hosted VMs. The testbed is a private cloud, where PMs are connected by Gigabit Ethernet and a switch. Xen [3] is used as the underlying hypervisor. The initial deployment and the considered CP/EP of our experiments are shown on Table 3.18. The scale of each CP and their corresponding prices are specified in Table 3.19.

For simplicity, we assume that the service-instances and their QoS/cost are equivalently important and thus all weights in the global objective function (Eq. 3.7) are set to 1. In addition, we consider both vertical and horizontal scaling; and apply a simple solution to determine the actions, this is: we always try vertical scaling (i.e., scale up/down) first before horizontal scaling (i.e., scale out/in). This is because horizontal scaling is usually more expensive than vertical scaling.

Table 3.18: Initial deployments and the examined objectives/primitives

PM	**VM**	**Service-instance**	**Objectives**	**Software CP**	**Hardware CP**	**EP**
PM1	VM	S_{11}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{21}	Throughput and cost	The max threads		workload
	VM	S_{31}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{41}	Throughput and cost	The max threads		workload
PM2	VM	S_{12}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{51}	Throughput and cost	The max threads		workload
PM3	VM	S_{32}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{61}	Throughput and cost	The max threads		workload

Table 3.19: Scaling options and price of control primitives

CP	**Optional Values**	**Unit**	**Price**
Max Threads	5,10,15,20,25,30,35,40,45,50	Thread count	$0.8 for each 5 unit per hr
CPU	1, 2,3, 4,5,6, 7, 8	Compute Unit	$2.5 for each 1 unit per hr
Memory	0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2,1.3,1.4, 1.5,1.6,1.7,1.8,1.9,2	GB	$1.5 for each 0.1 unit per hr

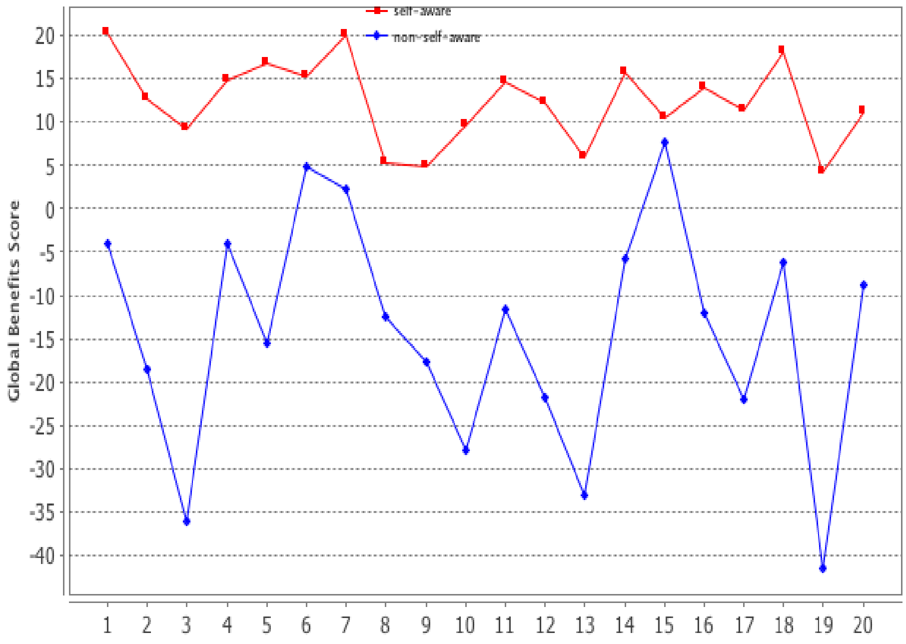


Figure 3.4: The global adaptation quality with respect to effect points

Figure 3.4 illustrate the results of the score (y- axis) in relation to each effect point (x-axis). we can clearly see that the self-aware system perform much better than the non-self-aware one along the entire time series. This is due to the fact that the non-self-aware system ignores the sensitivity caused by QoS interferences on co-located services and co-hosted VMs, which are significant in our experiments.

3.2.2 Smart Camera Networks Case Study

In the following, we qualitatively evaluate the proposed methodology by showing how it can be applied in the smart camera networks case study. We also show the experiments that compare the resulted system with a non-self-aware system.

Step 1 - Collect Requirements and Constraints

The requirements and constraints of the smart camera networks context as shown in the Table below:

Table 3.20: The functional, non-functional requirements and constraints for the smart camera networks case study.

Functional Requirements
The system should continuously track objects while they are visible to at least one camera of the network.
The system has to coordinate tracking of objects within a network of smart cameras via handover.
Each camera of the system has to be able to track objects autonomously within its own FOV.
The system has to be able to re-identify objects reliably within various cameras with different viewpoints.
Each camera has to be able to record information about its local handover behaviour.
The system should notice disappeared objects.
The system should be robust to node failures.
The system should be extensible (add new cameras during runtime).
The system should minimise communication effort while maximising tracking responsibility.
Non-functional Requirements
Maximise tracking utility.
Minimise the number of exchange messages in the network.
Constraints
Cameras can be added or removed during runtime.
Tracker can fail during execution.
Resources on each camera may not be exceeded.
the observed area has to be illuminated.

Step 2 - Propose Candidate Architecture

The presented architecture is implemented on distributed smart cameras. The workflow of the proposed architecture has been shown in Figure 3.5 . The sensor on each camera senses the current environment. The image acquisition collects the image data (step 1) from the sensor and transmits it to the tracking algorithm (step 2). The tracking algorithm detects and identifies the objects of interest. In case the object moves out of the scope of the current camera, the

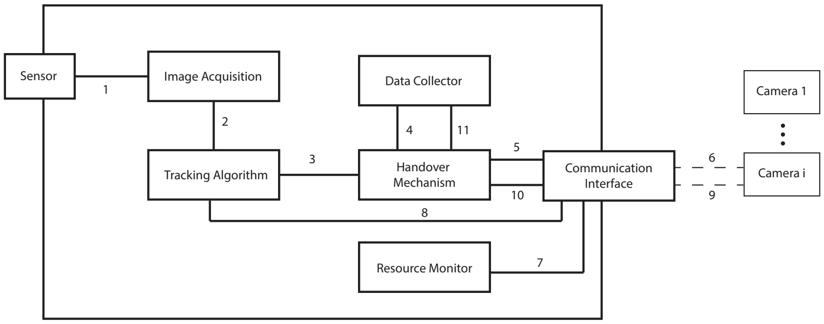


Figure 3.5: The proposed architecture

tracking algorithm initialises the handover mechanism (step 3). The handover mechanism requests available cameras to continue tracking from the data collector (step 4). Afterwards, the handover mechanism notifies those cameras and requests their tracking capabilities via the communication interface (step 5 and 6). Upon receiving such a request, the camera analyses its available resources (step 7) and tries to detect the object of interest (step 8). If the camera is able to track the object, it notifies the initial camera (step 9). When the initial camera received a reply from all contacted cameras, the next camera can be selected via the handover mechanism (step 10). Information about the cameras able to track the object are stored in the data collector and serve as a reference for future coordination (step 11).

Step 3 - Select the Best Pattern(s)

We now select the pattern using the questions presented previously for each self-awareness and self-expression capability:

Table 3.21: Questions and answers for deciding whether to include stimulus-awareness.

Stimulus-awareness	
What does the capability mean in your problem context.	Locate and value objects within own field of view of a camera.
What are the functional requirements that affected by this capability?	1. The system should continuously track objects while they are visible to at least one camera of the network. 2. The system has to coordinate tracking of objects within a network of smart cameras via handover. 3. Each camera of the system has to be able to track objects autonomously within its own FOV. 4. The system has to be able to re-identify objects reliably within various cameras with different viewpoints. 5. The system should notice disappeared objects.
What are the non-functional requirements that affected by this capability?	1. Maximise tracking utility.
What are the constraints that could affect this capability?	1. Cameras can be added or removed during runtime. 2. Tracker can fail during execution. 3. Resources on each camera may not be exceeded. 4. The observed area has to be illuminated.
Whether this capability is necessary or beneficial?	Yes

Table 3.22: Questions and answers for deciding whether to include time-awareness.

Time-awareness	
What does the capability mean in your problem context.	Unlearn previously learnt information in case something changes. Explore and exploit behavioural strategies.
What are the functional requirements that affected by this capability?	1. The system has to coordinate tracking of objects within a network of smart cameras via handover. 2. Each camera has to be able to record information about its local handover behaviour. 3. The system should minimise communication effort while maximising tracking responsibility.
What are the non-functional requirements that affected by this capability?	1. Maximise tracking utility. 2. Minimise the number of exchange messages in the network.
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

Table 3.23: Questions and answers for deciding whether to include interaction-awareness.

Interaction-awareness	
What does the capability mean in your problem context.	Reaction to auctions, bids and handover. Definition of neighbourhood based on auctions.
What are the functional requirements that affected by this capability?	1. The system has to coordinate tracking of objects within a network of smart cameras via handover. 2. The system has to be able to re-identify objects reliably within various cameras with different viewpoints. 3. The system should be robust to node failures. 4. The system should be extensible (add new cameras during runtime).
What are the non-functional requirements that affected by this capability?	1. Minimise the number of exchange messages in the network.
What are the constraints that could affect this capability?	1. Cameras can be added or removed during runtime. 2. Tracker can fail during execution. 3. Resources on each camera may not be exceeded. 4. The observed area has to be illuminated.
Whether this capability is necessary or beneficial?	Yes

Table 3.24: Questions and answers for deciding whether to include goal-awareness.

Goal-awareness	
What does the capability mean in your problem context.	Utility function for objects to be tracked. Performance measurement of different strategies.
What are the functional requirements that affected by this capability?	1. Each camera of the system has to be able to track objects autonomously within its own FOV. 2. The system has to be able to re-identify objects reliably within various cameras with different viewpoints. 3. The system should minimise communication effort while maximising tracking responsibility.
What are the non-functional requirements that affected by this capability?	1. Maximise tracking utility. 2. Minimise the number of exchange messages in the network.
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

Table 3.25: Questions and answers for deciding whether to include self-expression.

Self-expression	
What does the capability mean in your problem context.	Sending out auction invitations and bids.
What are the functional requirements that affected by this capability?	1. The system has to coordinate tracking of objects within a network of smart cameras via handover. 2. Each camera has to be able to record information about its local handover behaviour. 3. The system should be robust to node failures. 4. The system should be extensible (add new cameras during runtime).
What are the non-functional requirements that affected by this capability?	1. Minimise the number of exchange messages in the network.
What are the constraints that could affect this capability?	1. Cameras can be added or removed during runtime. 2. Tracker can fail during execution. 3. Resources on each camera may not be exceeded.
Whether this capability is necessary or beneficial?	Yes

Table 3.26: Questions and answers for deciding whether to include meta-self-awareness.

Meta-self-awareness	
What does the capability mean in your problem context.	Bandit solvers.
What are the functional requirements that affected by this capability?	1. The system should notice disappeared objects.
What are the non-functional requirements that affected by this capability?	1. Maximise tracking utility. 2. Minimise the number of exchange messages in the network.
What are the constraints that could affect this capability?	
Whether this capability is necessary or beneficial?	Yes

In summary, we again select the *Goal Sharing with time-awareness capability pattern including meta-self-awareness capabilities.* This selection is based on the aforementioned Table **??**.

Step 4 - Fit the Selected Pattern(s)

We now fit the proposed architecture to the selected pattern, as shown in the Figure 3.6 below.

Step 6 to Step 8

In smart camera systems, any alternative mechanisms apply either a central component (server) or introduce a priori knowledge about the scenario to coordinate tracking responsibilities. In the EPiCS smart camera demonstrator we do not have both assumptions and deploy our system without any knowledge of the scenario and without any central coordination. This allows a quick deployment of such a system in a highly dynamic environment. Due to the lack of applicable alternatives, we are not able to compare our approach directly. However, the benefits of our socio-economic approach in comparison to a prior knowledge and fixed communication partners is presented in the quantitative experiments.

The details of the final architecture are shown in the Table 3.27 below:

Table 3.27: The selected alternatives for the smart-camera case study.

		stimulus-awareness	interaction-awareness	time-awareness	goal-awareness	self-expression	meta-self-awareness	sensor	actuator
Selected alternative(s)		Object detection and tracking	Reaction to auctions, bids and handover	Artificial pheromones in vision graph	Utility function and local performance measurement	Communication policies and auction schedules	Multi-armed bandit problem solvers		
send	synchronous	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	asynchronous	function call	function call	function call	function call	function call	function call	function call	multicast
handle	sequential	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	parallel	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading	multi-threading
state		reactive	proactive and reactive	proactive and reactive	reactive	proactive and reactive	proactive and reactive		
transit							N/A		
link		one-to-one	many-to-many, one-to-one	one-to-one	one-to-one	one-to-many	one-to-many		
structure		compact	combine+separate	compact	combine+separate	combine+separate	compact	compact	compact
existence							exist		

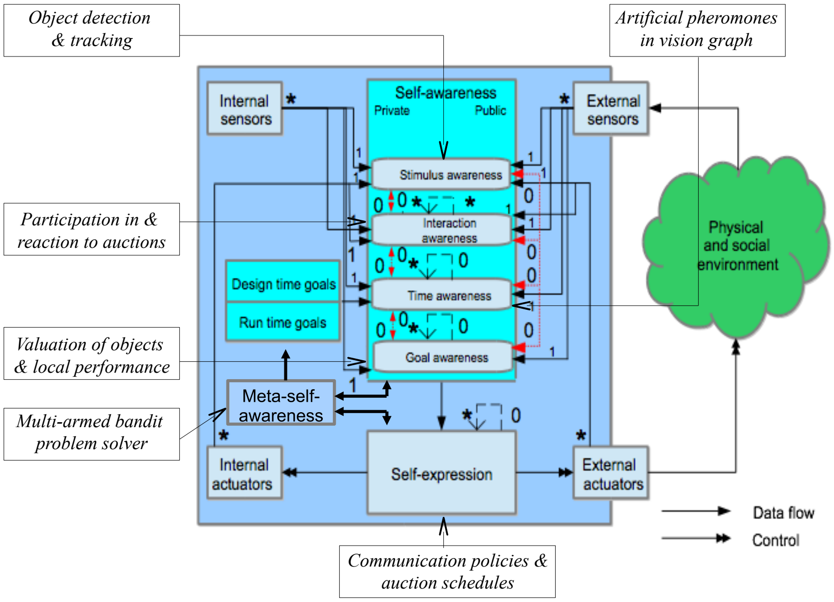
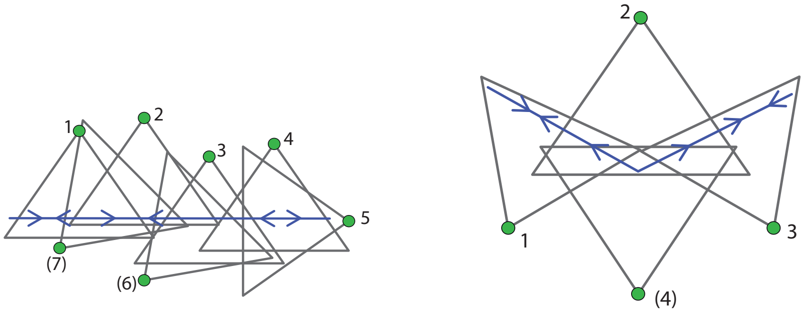


Figure 3.6: Fit the proposed architecture to Goal Sharing with time-awareness capability Pattern including meta-self-awareness capabilities.

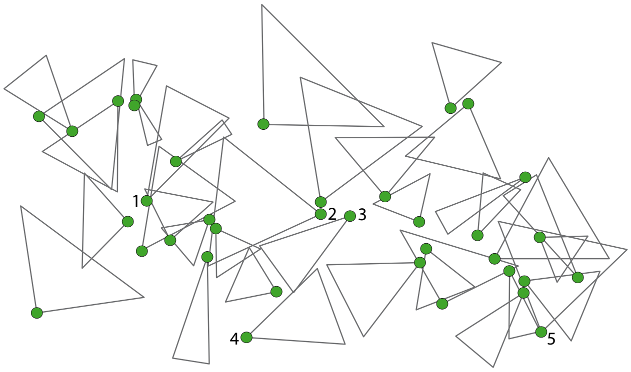
Quantitative Experiments

We conduct experiments with our smart camera case study using our self-aware, socio-economic approach and compare the results with a non-self-aware approach where each camera only communicates with its direct neighbours. In the self-aware approach, these neighbourhood relations are learnt online while in the non-self-aware approach the neighbourhood relationships are defined a priori and are not adapted during runtime. We simulate different scenarios and change the network of cameras during runtime. These so-called uncertainties affect the camera network only in the form of adding new cameras, remove existing cameras for a certain time or change the location and/or orientation of a camera. We measure the generated utility during runtime and show the accumulated utility for the entire network over time for all self-aware as well as the non-self-aware approach.

Since we are interested in performing repeatable experiments to investigate adaptivity and robustness issues, we used the simulation environment CamSim [11] with different scripted experimental setups of smart-camera networks. In the following subsection the different experimental scenarios will be described. For our experiments we considered three general scenarios and executed these scenarios with a variety of objects, paths and events. The different scenarios are illustrated in Figure 3.7.



(a) Scenario 1 (b) Scenario 2



(c) Scenario 6

Figure 3.7: Three qualitative different scenarios with various uncertainties. Green dots represent cameras while grey triangles indicate the corresponding field of view

For the first and second scenarios we defined paths for the object to traverse along. These paths are illustrated as blue lines. For scenario three, the objects move in a straight line in a random direction. For each scenario we defined different experiments using our events. For our three distinctive scenarios we conducted experiments where we added a camera during runtime, removed an camera from the test environment and changed the extrinsic parameters of a camera. An overview is given in the following Table:

Table 3.28: The experiments configurations.

Experiment No.	**Scenario**	**Action**
1	Scenario 1	Add Camera (6)
2	Scenario 1	Remove Camera 3
3	Scenario 1	Change Position 3 to (7)
4	Scenario 2	Add Camera (4)
5	Scenario 2	Remove Camera 2
6	Scenario 2	Change Orientation of Camera 2 by -55 degree
7	Scenario 3	Remove Cameras 1, 2, 3, 4, 5

For the non-self-aware approach applied in the first and second scenario, neighbourhood relations are defined between cameras only when they have overlapping FOVs. For scenario three, neighbourhood relations are defined whenever an object can traverse from the FOV of one camera to another in a straight line without appearing in the FOV of any other camera.

Figure 3.8 shows results for experiment number 3 employing our active approach. We changed the position of a single camera within the environment to show the ability of our approach to deal with changes of the extrinsic parameters of cameras. The vertical line shows the time at which the event happened. The drop in utility gain for the static approach after the event occurred is apparent, demonstrating its inability to adapt to the change. While the static approach loses overall utility, the SMOOTH and STEP policies are able to keep a high utility after the event, indicating their robustness to change.

The results of experiment 4 are shown in Figure 3.9 where we added a new camera during runtime. The occurrence of the event is indicated with a red vertical line again at time step 518. The increased accumulated utility using the active SMOOTH and STEP approach is apparent. Since the camera was placed at a location which was already covered by a different camera, the improvement was rather small.

Figure 3.10 illustrates the results of scenario 2 with a camera failure event (experiment 5), when passive approaches were used. Here the drop of the accumulated utility is obvious for the static approach, while the socio-economic approaches are able to relearn the vision graph online and continue tracking the

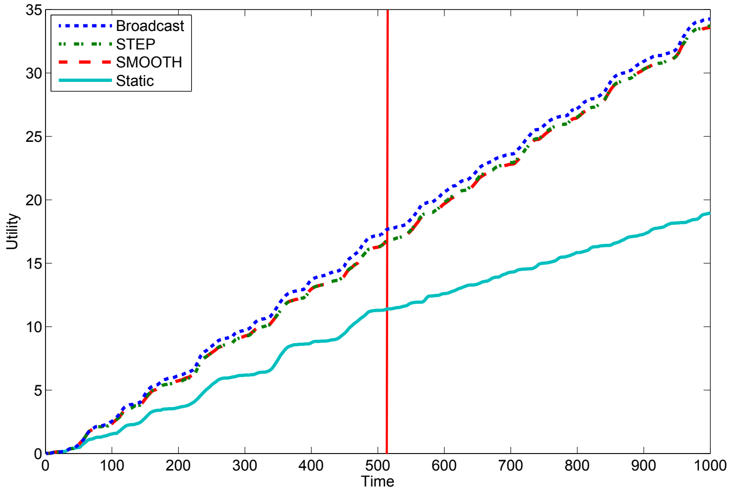


Figure 3.8: Cumulative sum of the entire network utility over time for a typical simulation run of experiment 3 (Scenario 1 with change event) and using our passive approaches. The vertical line indicates the timestep when the event occurred. The simulation ran for 1000 timesteps. We changed the position of a single camera within the environment to show the ability of our approach to deal with changes of the extrinsic parameters of cameras.

object within the entire network.

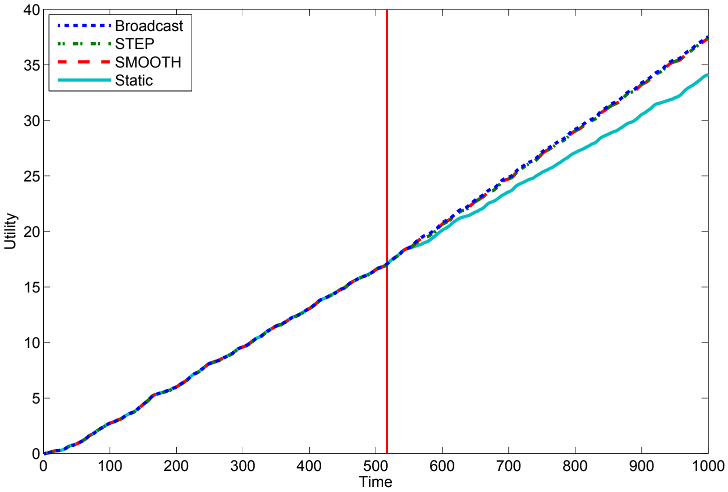


Figure 3.9: Cumulative sum of the entire network utility over time for a typical simulation run of experiment 4 comparing our active socio-economic approaches with a static handover. The red vertical line indicates the timestep when the event occurred. The simulation ran for 1000 timesteps.

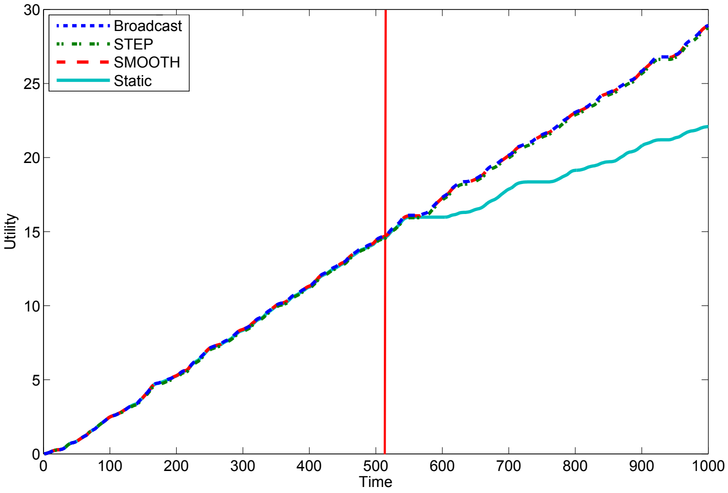


Figure 3.10: Cumulative sum of the entire network utility over time for a typical simulation run of Scenario 2 with an error event (experiment 5) and using our active approaches. The red vertical line indicates the timestep when the event occurred. The simulation lasted for 1000 timesteps.

Bibliography

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Google app engine. http://code.google.com/appengine/.

[3] Xen: a virtual machine monitor. http://xen.xensource.com/.

[4] Tariq Al-Naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, and Boualem Benatallah. A quality-driven systematic approach for architecting distributed software applications. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 244–253, New York, NY, USA, 2005. ACM.

[5] Tobias Becker, Andreas Agne, Peter R. Lewis, Rami Bahsoon, Funmilade Faniyi, Lukas Esterle, Ariane Keller, Arjun Chandra, Alexander Refsum Jensenius, and Stephan C. Stilkerich. Epics: Engineering proprioception in computing systems. In *Proc. Int. Conf. on Computational Science and Engineering (CSE)*, pages 353–360. IEEE Computer Society, dec 2012.

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

[7] Tao Chen and Rami Bahsoon. Self-adaptive and sensitivity-aware qos modeling for the cloud. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 43–52, Piscataway, NJ, USA, 2013. IEEE Press.

[8] 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.

[9] Rogério de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap.

[10] 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.

[11] Lukas Esterle, Peter R. Lewis, Horatio Caine, Xin Yao, and Bernhard Rinner. Camsim: A distributed smart camera network simulator. In *Proceedings of the 2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, SASOW '13, pages 19–20, Washington, DC, USA, 2013. IEEE Computer Society.

[12] 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.

[13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX.

[14] Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ESEC / SIGSOFT FSE*, pages 347–350. ACM, 2003.

[15] 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.

[16] Clements Paul, Len Bass, and Rick Kazman. *Evaluating Software Architectures: Methods and Case Studies.* MA: Addison-Wesley, 1998.

[17] Clements Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, 2002.

[18] Thomas L. Saaty. *The Analytical Hierarchical Process.* McGraw-Hill, 1980.

[19] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and KarlM. Göschka. On patterns for decentralized control in self-adaptive systems. In Rogério Lemos, Holger Giese, HausiA. Müller, 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.

[20] 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.