Continuous and Proactive Software Architecture Evaluation: An IoT Case

DALIA SOBHY, Computer Engineering Department, Arab Academy of Science and Technology and Maritime Transport, Egypt
LEANDRO MINKU, University of Birmingham, UK
RAMI BAHSOON, FRSA and University of Birmingham, UK
RICK KAZMAN, SEI/CMU and University of Hawaii, USA

Design-time evaluation is essential to build the initial software architecture to be deployed. However, experts' assumptions made at design-time are unlikely to remain true indefinitely in systems that are characterized by scale, hyperconnectivity, dynamism and uncertainty in operations (e.g. IoT). Therefore, experts' design-time decisions can be challenged at run-time. A continuous architecture evaluation that systematically assesses and intertwines design-time and run-time decisions is thus necessary. This paper proposes the first proactive approach to continuous architecture evaluation of the system leveraging the support of simulation. The approach evaluates software architectures by not only tracking their performance over time, but also forecasting their likely future performance through machine learning of *simulated* instances of the architecture. This enables architects to make cost-effective informed decisions on potential changes to the architecture. We perform an IoT case study to show how machine learning on simulated instances of architecture can fundamentally guide the continuous evaluation process and influence the outcome of architecture decisions. A series of experiments is conducted to demonstrate the applicability and effectiveness of the approach. We also provide the architect with recommendations on how to best benefit from the approach through choice of learners and input parameters, grounded on experimentation and evidence.

CCS Concepts: • **Continuous Evaluation** → **Software Architecture Evaluation**; *Design-time, Reactive, Proactive*;

Additional Key Words and Phrases: Continuous evaluation, software architecture evaluation, time series forecasting, IoT

NOMENCLATURE

d_{ka}: Architecture decision for capability k implemented using component a
dao_i: i diversified architecture option
DAO: A set of diversified architecture options
$|DAO|$: Number of diversified architecture options
$q_{dao_i}(t)$: Quality of a *dao* varying over time
$q'_{dao_i}(t)$: Normalized Quality of a *dao* varying over time
$\hat{q}_{dao_i}(t)$: Forecast for Normalized Quality of a *dao* varying over time
$q'_{dao_i}(t+h)$: Normalized Quality of a *dao* varying over time for further ahead timesteps
$\hat{q}_{dao_i}(t+h)$: Forecast for Normalized Quality of a *dao* varying over time for further ahead timesteps
$|Q|$: Number of quality attributes
w_q: Weight of quality q
$B_{dao_i}(t)$: Benefit of a *dao* varying over time
$\mu_{dao_i}(t)$: Exponentially Smoothed Benefit of a *dao* varying over time
$c_{dao_i}(t)$: Cost of a *dao* varying over time
$c^v_{dao_i}(t)$: Costs of each variety v per *dao* (e.g. deployment cost, leasing cost, *etc*)

Authors' addresses: Dalia Sobhy, Computer Engineering Department, Arab Academy of Science and Technology and Maritime Transport, Alexandria, Egypt, dalia.sobhi@aast.edu; Leandro Minku, University of Birmingham, Birmingham, UK, l.l.minku@cs.bham.ac.uk; Rami Bahsoon, FRSA and University of Birmingham, Birmingham, UK, r.bahsoon@cs.bham.ac.uk; Rick Kazman, SEI/CMU and University of Hawaii, Hawaii, USA, kazman@hawaii.edu.

$\sigma^2_{dao_i}(t)$: Exponential Variance of a *dao* varying over time
$\sigma_{dao_i}(t)$: Exponential Standard Deviation of a *dao* varying over time
θ: Relative Importance of the past
α: Confidence level
$L_\lambda(dao_i, t)$: Loss Function shows how (un)desirable a *dao* is
$L'_\lambda(dao_i, t)$: Marginal loss of non-dominated *dao* varying over time
λ: A predefined parameter that controls the relative importance between $c_{dao_i}(t)$ and $\mu_{dao_i}(t)$
dao_{curr}: Current *dao*
ζ: Number of input attributes
h: Number of further ahead timesteps
η: Number of training examples
η_{min}: Minimum number of training examples for enabling forecasts
τ_q: Error threshold per quality attribute
$\hat{B}_{dao_i}$: Forecast benefit of a *dao* varying over time
$\hat{\mu}_{dao_i}(t)$: Forecast exponentially smoothed benefit of a *dao* varying over time
$\hat{\sigma}_{dao_i}(t)$: Standard deviation based on forecast exponentially smoothed benefit of a *dao* varying over time
$\hat{L'}_\lambda(dao_i, t)$: Forecast Marginal loss of non-dominated *dao* varying over time
$e_q(t)$: Forecast error per timestep
MAE: Mean Absolute Error
RMSE: Root Mean Square Error
RT: Response Time
NU: Network Usage
EC: Energy Consumption

1 INTRODUCTION

Design-time evaluation [19, 71, 73] is a necessary step to make fundamental architecture decisions when designing a software system. However, its effectiveness is highly influenced by the expertise of the evaluators, their knowledge of the domain and the usage contexts for which the architecture is conceived [20, 47, 89, 100]. Valuation and stakeholders' perception on cost and value of the candidate architecture solution can suffer from under- or over-estimations. They also rely on assumptions that are unlikely to hold true indefinitely in dynamic, scalable and uncertain environments, such as Internet of Things (IoT). Dynamicity, multi-tenancy, hyper connectivity, environmental changes and uncertainty are fundamental properties of such architectures, which are difficult to conceive and cater for purely via design-time evaluation.

In this context, design-time evaluation tends to be subjective and incomplete. Widely used architecture evaluation methods [71, 73, 100] fundamentally lack mechanisms for aligning operations with development when the architecture evaluation is conducted. Therefore, a more continuous approach to software architecture evaluation is necessary to *complement* design-time evaluation. We define *continuous* software architecture evaluation as *multiple evaluations of the software architecture that begin at the early stages of the development and can be periodically and repeatedly performed throughout the lifetime of the software system.* Continuous evaluation is composed of two phases: *design-time* and *run-time* evaluation. In particular, design-time evaluation supports the necessary initial system design and deployment. Run-time evaluation assesses the extent to which the architecture options conceived at design-time, as well as other potential architecture options, can perform well at run-time. This enables architects to make informed decisions on the potential changes to the architecture, so that its performance remains good over time.

Continuous architecture evaluation has been introduced and discussed as an integral phase within modern software development processes (e.g., agile [93], DevOps [14], and continuous delivery [64]). This is an enabler for rapid response to operational, environmental, and requirements changes. It has also been advocated as an essential appraisal and quality assurance practice that symbiotically aligns development and operations as a measure for responding to dynamism, unpredictability, and operational uncertainties. Despite the widespread acceptance of continuous architecture evaluation as a concept, findings from our systematic literature review on the topic ([102]), show that the literature requires further examples, frameworks and systematic methods on how continuous evaluation can actually be realized and conducted. There are few research efforts (e.g. [20, 50, 93]) which explicitly mention continuous architecting and assessment, while some others implicitly adopt it (e.g. [14]). Those approaches can benefit from further analysis in terms of dynamic tracking and forecasting of architecture decisions' performances, and automated management of cost-benefit trade-offs.

Modern architecting practices have given rise to harvesting operational data (e.g. QoS data). The availability of such data has provided new opportunities for continuous architecture evaluation, whether the data are of real-time or simulated nature. Continuous evaluation leveraging data could potentially: (i) aid the architect in continuously learning about architecture decisions; (ii) complement design-time decisions; (iii) help in forecasting how well the architecture will behave in the future; (iv) enable proactively dealing with variability scenarios, which can be difficult to evaluate in the absence of data.

The state-of-art has provided well-established systematic methods for evaluating architecture design decisions and choices at design-time to replace ad hoc practices when adopting and deploying architectures (e.g.[71, 89, 96, 100]). However, as the environment is dynamic, value potentials of architecture design decisions and choices can fluctuate at run-time. In our previous work, [101], we have proposed a reactive approach for evaluating software architectures at run-time, using techniques inspired by reinforcement learning [109]. This approach monitors architecture design decisions with respect to some quality attributes of interest. If the adequacy of a given decision starts to deteriorate, the approach then reacts by suggesting possible refinements or changes of the decisions.

Though this approach [101] continuously helps the architect in understanding the past behaviour of the architecture decisions in question, it cannot proactively reason about their future potentials. Such proactive behaviour is important, because a decision that has worked well in the past may not necessarily work well in the future, due to the potential changes and uncertainty underlying the environment where the system is embedded. Conversely, a decision that may not seem attractive based on the past could have better future potentials. Discarding that decision could lead to poor future performance. Moreover, a current decision may seem poor, but it could become attractive in the near future. Relying on a reactive approach in this situation could trigger unnecessary architecture adaptations that would cause the system to be unstable. In summary, reactive approaches can lead to partially justified decisions, unnecessary adaptation, and increase development cost, while slowing down the operations. Therefore, existing work has only exploited data to provide benefit (i) mentioned previously, and partially provide benefit (ii). The full power of data has yet to be harnessed to provide benefits (i)-(iv), which are essential to better inform software architecture decisions at run-time.

This paper thus provides the first investigation of proactivity in continuous software architecture evaluation, when supported by simulated instances of the system-to-be. It exploits machine learning in the form of time series forecasting analytics to produce a more powerful continuous architecture evaluation approach, able to provide benefits (i)-(iv). Overall, this paper answers the following research questions:

- **RQ1:** *How proactivity in continuous evaluation, when supported by simulated instances of the system-to-be, can be realized and conducted? For instance, can continuous time series forecasting analytics, leveraging simulated instance of the system-to-be, enable us to predict the behaviour of software architectures over time? If so, how well?*
- **RQ2:** *How can proactive approaches complement reactive ones to provide a well-rounded and more effective continuous architecture evaluation, when supported by simulated instance of the architecture? What benefits can they bring to continuous software architecture evaluation and decision-making at run-time?*

To answer these RQs, this paper proposes a novel approach to continuous software architecture evaluation. It uses continuous time series forecasting [22, 46, 75] as a built-in mechanism to complement reactive run-time evaluation with proactive run-time evaluation. In this way, the proposed approach not only takes into account the past performance of architecture decisions, but also their future potentials, better informing run-time architecture decisions.

To enable proactive run-time evaluation, as a fundamental step in continuous evaluation, we envision several potential scenarios to evaluate the architecture of the system-to-be and capture its dynamic behavior:

- Simulation and simulated instances of the system-to-be is a commonly used engineering and scientific alternative to experiment, when the system-to-be is large scale and costly to implement. For the case of IoT as an example, the system can be composed of large numbers of heterogeneous and diverse things, including sensors and devices. As it would be prohibitively expensive to implement the system for testing and evaluation purposes, simulation can serve in prototype instance of the architecture and can assist in exhaustive and comprehensive in what-if analysis of the system to be, stress-testing of the architecture with inputs that can go beyond the ones encountered in normal operation and potential cost-effective scaling of the analysis.
- Run-time evaluation can also work if run-time data of a given configuration is available. Run-time data can be available through published benchmarks of functionally and behaviourally equivalent systems, cross companies data using similar systems and configurations (i.e. twin systems), the actual system itself if implementation is available, and/or data generated from throwaway prototypes or simulated instances of the system-to-be. Additionally, advances in simulation can symbiotically link a physical system to its simulated instances, under what so called to digital twins to enable continuous run-time monitoring, logging, and profiling of architecture design decisions and quality attributes of interest. This can be particularly useful for cases where the system is already deployed and further refinements are envisioned.
- A third scenario can be also possible, where the approach can be integrated in continuous development paradigms, such as DevOps, where run-time information from the operation side can provide feedback for development.

Each of the above mentioned scenarios can lead to distinct realisation of continuous architecture evaluation or stages for enabling such evaluation. The scope of this paper is concerned with the case of using simulation to enable proactive run-time evaluation of the architecture of the system-to-be. Nevertheless, the approach is fundamentally transferable and can be applicable to other cases, would the implemented system is available.

In particular, our proactive run-time evaluation adopts a proprietary simulation tool, *iFogSim* [60] for its fit to the case of IoT. *iFogSim* builds on Cloudsim [32]; it provides the architect with the freedom for hierarchically composing fog devices, clouds, and data streams to simulate the technical and value potentials of selected decisions, using normal usage and stress tests. The generated data of simulated eventualities of system operations, mode of interactions of things, requirements

and constraints related to the interaction, compile a "rich" set for enabling proactive evaluation. Proactivity can be, for example, useful in not only profiling and analyzing the likely technical and value potentials of the architecture decisions at run-time, but also in designing configuration policies, refinements and diversified options that can better cope with the predicted situation, should it be encountered.

A series of experiments were conducted to demonstrate the applicability and effectiveness of this approach using the case of architecting for IoT. The suitability of various *time series forecasting* algorithms [22, 46, 75] to realize proactivity is investigated under this case study (RQ1). The impact of proactivity on run-time decision-making is then evaluated through a comparative study against design-time and reactive continuous evaluation approaches (RQ2). Simulated run-time and operations data [60] were used to test the potentials of the approach on wider spectrum of scenarios and cases. These experiments also provide software architects with systematic guidance on how the approach can be realized in practice given a wide set of scenarios and data capabilities for forecasting (i.e. how to select forecasting algorithm, how to best benefit from forecasts, *etc*).

Our results show that the proactive approach was more advantageous than the design-time and reactive approaches in some important scenarios. In particular, we have set two scenarios with different QoS constraints (i.e. the target QoS should not exceed a particular threshold) based on the context and the stakeholders requirements. For normal constraint scenarios, the proactive approach provided similar overall behavior to the reactive approaches, but with improved system stability (i.e. smaller number of switches). It also contributed to a significant improvement in benefit: about 20-60% as compared to design-time approaches. For most of the scenarios where architects have set strict constraints, it also produced the best results (i.e. 40-70% better than reactive and baseline approaches).

Our novel contributions are:

- The first investigation of whether and how time series forecasting can be used to successfully forecast the future benefit of architecture decisions.
- The first proactive approach to continuous software architecture evaluation, when supported by simulated instances of the system-to-be. The approach uses the power of forecasting analytics to complement design-time decisions by taking into account not only the past, but also the future potentials of architecture decisions.
- A detailed analysis of when and why the proposed approach can help to better inform architecture decisions.
- Experimental guidelines aiding architects on how to tune the proposed approach.

The remainder of this paper is structured as follows: Section 2 discusses the necessary background to understand the proposed approach. Section 3 then presents the proposed approach. Section 4 introduces our motivating example in the context of IoT and its challenges. Section 5 provides a constructive and comparative evaluation. Section 6 provides a further discussion on the method, whereas Section 7 discusses threats to validity. Section 8 presents the related work. Section 9 concludes the work.

2 BACKGROUND

In this section, we will provide the background necessary to understand the proactive run-time evaluation approach.

2.1 Reactive Run-time Architecture Evaluation

Software architectures comprises a set of design decisions [27, 68, 107] that can relate to the architecture components, connectors defining the topology, and their relation to the environment.

The architects define the possible set of candidate architectures to serve a particular concern and then based on their experience and knowledge they choose the best candidate [35]. For example, in an IoT application, the design decisions for processing the data on the cloud rather than the fog devices to improve the energy consumption goes beyond networking as several qualities and their trade-offs need to be studied. The design decision could have a negative impact on the performance, calling for systematic software architecture evaluation.

In our previous work [101], we developed a systematic method for reactive evaluation. The reactive approach is a run-time evaluation approach inspired by self-adaptive systems. The approach is able to profile situations where options can be more effective and provide continuous updates on their value potentials. The reactive approach makes use of a simple reinforcement learning strategy that tracks the benefit of the architecture options over time using an exponential smoothing function that emphasizes the more recent benefit values over older ones. A change detection method is used to detect whether the time-decayed benefit is deteriorating significantly. If it is, reaction can be triggered under the assumption that such low benefit values will persist. In other words, it assumes that the best architecture option to switch to is the one which has recently been obtaining the best cost-benefit. This is effective in some contexts, e.g., when the deployed architecture option is not violating the QoS constraints that much, or when its performance level is not significantly changing over time. However, in other contexts, the reactive learning may suggest wrong decisions and recommend unnecessary switches due to the lack of knowledge about the future benefit of candidate architecture decisions. Taking into account the future potential of architecture options is important to provide a more informative evaluation. More advanced supervised learning approaches such as the ones used in the present paper have not been adopted. This is the motivation to the proposed approach, which harvests the power of predictive analytics to forecast the benefit of architecture decisions over time. In particular, the reactive approach is only able to track the value of the benefit over time, and is not able to predict future benefit values. Section 4.2 of [101] further clarifies how the reactive approach can be understood in the framework of reinforcement learning.

2.2 Forecasting Algorithms

A time series is a sequence of observations (e.g. response time, energy consumption, *etc*) measured/received sequentially over time [87]. Time series forecasting consists in forecasting the value of a future observation (output attribute), based on the values of a number of previous observations (input attributes) in the sequence. In this work, we will consider that, once the true value of the future observation becomes known, it can be used to create a training example (i.e., an example where both the input and output attributes are already known) for updating the current forecasting model. After that, this training example is discarded. The learning procedure where forecasting models are updated over time whenever a new observation arrives is called online learning, and is adequate when data arrive at high speed, as is the case of IoT. In this section, we will briefly explain two types of online learning algorithms, for stationary and non-stationary environments. More detailed explanation on how the forecasting model is built and trained is given in Section 3.2.

Learning models for stationary environments assume that the true function underlying the relationship between input attributes and forecast values does not change over time. The online learning models for stationary environments investigated in this work are: k-Nearest Neighbors (kNN) [111], Perceptron [61], and Stochastic Gradient Descent (SGD) Multiclass [28, 78, 98].

Learning models for non-stationary environments assume that the true function underlying the relationship between input attributes and forecast values may change over time, requiring specific mechanisms for models to update to such changes. The algorithms for non-stationary environments investigated in this work are: Fast Incremental Model Trees with Drift Detection (FIMTDD) [22, 65], Fading Target Mean (FTM) [22, 55] and Additive Experts (AddExp) [75].

3 PROACTIVE ARCHITECTURE EVALUATION APPROACH

The proposed continuous architecture evaluation approach is an approach that intertwines run-time evaluation with design-time evaluation for more informed decision-making. The steps of the approach are summarized in Figure 1. Here, the design-time evaluation is the one proposed in [100], which uses options theory (i.e. an economics-driven approach) [63] to evaluate the diversified architectural options and shortlist the initial diversified architecture options for run-time evaluation. The approach proposed in this paper builds on the reactive run-time approach (Section 2.1) by adopting forecasting analytics as depicted in the forecast and learn module from Figure 1. The use of forecasts makes the approach proactive, because it does not only rely on reacting to potential changes in the benefit of architecture decisions over time, but attempts to forecast them. In this way, architecture decisions could be made before negative changes in benefit become detrimental.

The approach provides a generic model that can be instantiated to address various quality of service concerns and their trade-offs. It is divided into two parts as shown in Figure 1: design-time evaluation It tracks and evaluates the actual benefit and cost using exponential decay factors (Section 3.1 – *Evaluate* in Figure 1). The benefit is then modelled based on quality attribute forecasting models (Section 3.2– *Forecast & Learn* in Figure 1). Then, instead of detecting significant deviations based only on previous benefit values, the proactive approach considers their *future* potentials (Section 3.3– *Detect* in Figure 1). The same applies to the selection of architecture options having the optimal cost-benefit trade-off, which is also based on their forecast benefits (Section 3.4– *Select* in Figure 1). This is done to avoid critical problems, such as reactively recommending an architecture design option which does not introduce added value over time, triggering unnecessary adaptations and hence leading to an unnecessary increase in costs. The future potentials are assessed based on *time-series forecasting approaches* [22, 46] (Section 2.2), which forecast the future benefits of diversified architecture options based on previous quality attribute values observed. Any time-series forecasting approach could potentially be used. In this context, our forecasting model relies on the availability of historic data (in numeric format) for the first few time steps to evaluate an architecture with respect to qualities of interest. After that, it can learn over time. Figure 2 illustrates the operational procedures performed by the proactive run-time evaluation approach. The steps of the proactive approach are explained below.

3.1 Evaluate

Design diversity [9, 10, 15, 62, 100] is used to design for dependability under uncertainty: the greater the uncertainty, the more diversity the architects may need to apply to improve performance. Our method investigates this phenomenon and formulates the problem of architecture diversity from run-time and economics-driven perspectives. We denote a software architecture which embeds diversity as diversified architecture option *dao* and the set of *dao* as *DAO*. A *dao* implements a set of diversified decisions to meet some quality goals and trade-offs. Consider a set of architecture decisions D, where a decision $d_{ka} \in D$; k denotes a particular capability, including connectivity, data collection, data management, *etc*; and a indicates the software architecture component and connection that implements this capability k. For example, in an IoT system, architecture decisions for the capability of data collection d_1 could be performed either through fixed d_{11}, mobile d_{12}, or fixed+mobile sensors d_{13}. Another example is data processing could be performed either in cloud d_{21}, or fog+cloud d_{22}. Therefore, dao_i could collect data from fixed and mobile sensors (d_{13}) and processes it in cloud-fog (d_{23}). Other examples of *DAO* are depicted in Section 4 through Table 3. In the IoT system case, the diversity in each *dao* can refer to using fixed and/or mobile fog devices for data collection capability; using different cloud providers and heterogeneous fog devices for data processing capability.

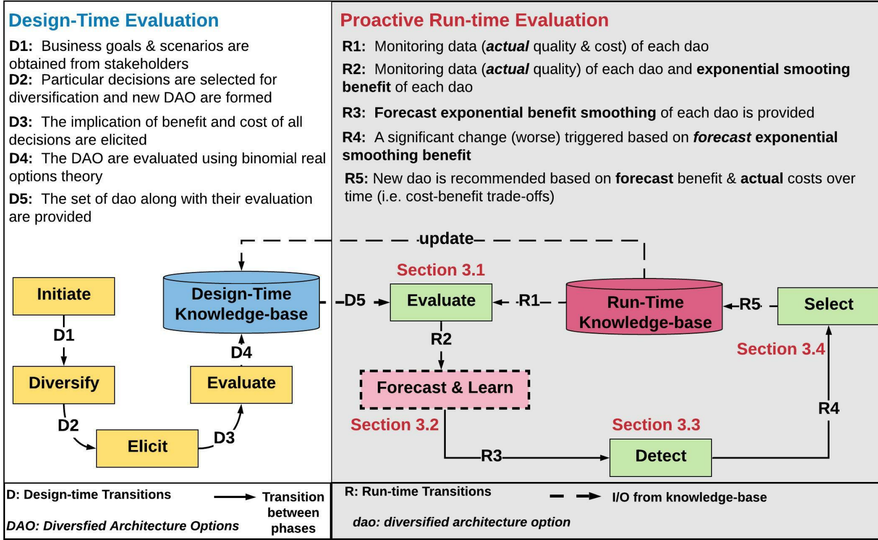


Fig. 1. Steps of the continuous evaluation approach, where the design-time evaluation [100] forms the initial design decisions and proactive run-time evaluation complements it by adopting time series forecasting.

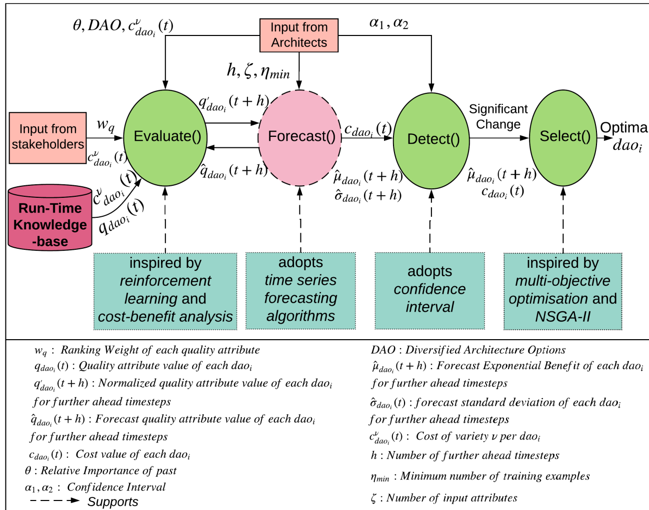


Fig. 2. An overview of the operational procedures of proactive run-time evaluation approach.

In the *Evaluate* step (Figure 1), we monitor the benefits of the diversified architecture options. The benefit $B_{dao_i}(t)$ of dao_i at timestamp t is a measure of the contribution of each quality attribute (R1 in Figure 1) to value creation, i.e., added value. Each $B_{dao_i}(t)$ (R2 in Figure 1) is a function of $q_{dao_i}(t), \forall q \in Q$, whereas each $q_{dao_i}(t)$ is a composite of the quality $q_{ka}(t)$ of each of its component architecture decisions $d_{ka} \in dao_i$. Each $q_{dao_i}(t)$ has one constraint, which follows one of the

following possible formats: $q_{dao_i}(t) \leqslant q^{max}$, $q_{dao_i}(t) \geqslant q^{min}$, $q^{min} \leqslant q_{dao_i}(t) \leqslant q^{max}$. If none of the benefit's quality attributes violates any constraint, then the benefit is computed using the following equation:

$$B_{dao_i}(t) = \sum_{q \in Q} w_q * q'_{dao_i}(t) \ . \tag{1}$$

If a *dao* at time t, violates any constraint, its benefit is set to zero. To compute $q_{dao_i}(t)$, the qualities of the architecture decisions need to be aggregated based on how they are connected to each other. Table 1 depicts some aggregate functions for quality of service. To place all quality attributes in the same scale, $q'_{dao_i}(t) = \frac{q_{dao_i}(t) - q^{min}}{q^{max} - q^{min}}$ could be used for scaling quality values that need to maximized (the higher the better, e.g. throughput), whereas $q'_{dao_i}(t) = \frac{q^{max} - q_{dao_i}(t)}{q^{max} - q^{min}}$ is for scaling quality values that need to minimized (the lower the better, e.g. response time), where $q^{max} - q^{min} \neq 0$. $q'_{dao_i}(t)$ denotes the normalized value of a given q of a particular dao_i at time t.

Table 1. Aggregate Functions. The Max, Min and $\sum$ operations are over all architecture decisions that are connected to each other in the specified way (parallel or sequence).

QoS Attribute	Parallel	Sequence
Response Time	$Max(q_{ka})$	$\sum q_{ka}$
Energy Consumption	$\sum q_{ka}$	$\sum q_{ka}$
Cost	$\sum c_{ka}$	$\sum c_{ka}$

We then monitor the costs over time (R1 in Figure 1). The cost associated with an architecture option at time t is denoted by $c_{dao_i}(t)$, which is computed using $c_{dao_i}(t) = \sum c^{v}_{dao_i}(t)$. Our consideration for the cost is situation dependent. As an example, the cost can relate to one or more dimensions of interest (i.e. v is variety of costs). This can include the cost of configuration, deployment, leasing, switching *etc*. These costs can be estimated using parametric models, reliant on experts (i.e. architects and other stakeholders), *etc* [26, 70], as well as run-time knowledge (i.e. monitoring tools).

In non-dynamic environments, tracking the benefit based on a simple average of its value at each time t could be sufficient. However, in dynamic environments, simple average may take a long time to reflect changes [109] in benefit. In this context, we employ a time-decayed average function $\mu_{dao_i}(t)$ of a *dao* (R2 in Figure 1) for evaluating the benefit inspired by reinforcement learning [109]. In particular, a decay function is adopted that continually learns and updates the aggregated benefit of monitored quality values forming the exponentially smoothed benefit $\mu_{dao_i}(t)$, which is computed as follows:

$$\mu_{dao_i}(t) = \theta \mu_{dao_i}(t-1) + (1-\theta) B_{dao_i}(t) \ . \tag{2}$$

How much emphasis is given to the present/past is controlled by a predefined parameter θ, which affects the system's stability. In particular, the relative importance of the present is denoted by $1 - \theta$, whereas of the past by θ, where $0 \leq \theta < 1$.

After that we determine the variance $\sigma^2_{dao_i}(t)$ and standard deviation $\sigma_{dao_i}(t)$ of the benefit of a *dao* over time; $\sigma^2_{dao_i}(t) = \theta \sigma^2_{dao_i}(t-1) + (1-\theta) * (B_{dao_i}(t) - \mu_{dao_i}(t))^2$, $\sigma_{dao_i}(t) = \sqrt{\sigma^2_{dao_i}(t)}$.

3.2 Forecast and Learn

The forecasting model learns a function that receives as input previous quality attribute values monitored by the Evaluate step, and outputs a forecast of the future value of the quality attribute.

Since there are different types of quality attributes monitored over time $q_{dao_i}(t)$: positive (e.g. Response Time) and negative (e.g. Throughput). The current step uses the normalised ones $q'_{dao_i}(t)$ as generated by the Evaluate step (Section 3.1).

One forecasting model is produced for each normalised quality attribute. We refer to a point in time when a new quality attribute value has been observed as a timestep. At each timestep t, the following two main operations are run, for each quality attribute:

1. **Learn:** a new training example is produced if enough quality attribute values have been observed to compose it. This training example is used to train (update) the forecasting model corresponding to this quality attribute. A training example is composed of ζ input attributes, where ζ is a predefined parameter, and one output attribute. The input attributes are past quality values $(q'_{dao_i}(t'-h-\zeta-1), q'_{dao_i}(t'-h-\zeta), \cdots, q'_{dao_i}(t'-h-1), q'_{dao_i}(t'-h))$, where h represents the number of timesteps in the future for which we wish a prediction to be made. The output attribute is the quality value that we want to learn how to predict $(q'_{dao_i}(t'))$. Therefore, a training example can be seen as a tuple $<q'_{dao_i}(t'-h-\zeta-1), q'_{dao_i}(t'-h-\zeta), \cdots, q'_{dao_i}(t'-h-1), q'_{dao_i}(t'-h), q'_{dao_i}(t')>$
2. **Forecast:** a forecast $\hat{q}_{dao_i}(t+h)$ is provided based on the forecasting model corresponding to the quality attribute. The forecast is made by feeding the past ζ observed quality attributes $(q'_{dao_i}(t-\zeta-1), q'_{dao_i}(t-\zeta), \cdots,, q'_{dao_i}(t-1), q'_{dao_i}(t))$ as inputs to the forecasting model.

Table 2 illustrates the training examples and forecasts produced at each timestep for a given quality attribute. In this illustrative example, the quality value observed in a given timestep t has the same numeric value as the timestep itself, so that the example is easier to follow. However, in real scenarios, the quality value received at a given timestep t does not necessary have the value t.

Suppose that the number of input attributes ζ used for forecasting is 4 and that we wish to forecast $h = 3$ timesteps in the future. Therefore, the algorithm will require 4 past quality values $(q'_{dao_i}(t-3), q'_{dao_i}(t-2), q'_{dao_i}(t-1), q'_{dao_i}(t))$ as input attributes to provide a forecast $\hat{q}_{dao_i}(t+3)$.

A given training example $< q'_{dao_i}(t-h-\zeta-1), q'_{dao_i}(t-h-\zeta), \cdots, q'_{dao_i}(t-h-1), q'_{dao_i}(t-h), q'_{dao_i}(t) >$ can only be produced at a given timestep t if all its input and output attribute values have already been observed. Therefore, the first training example used to train the forecasting model can only be produced at timestep $t = 7$. This also means that the first forecast by the model can only be provided at timestep $t = 7$. Before that, even though new quality values were being observed and stored to compose training examples, no complete training example had been produced yet. So, the forecasting model could not be built. From timestep $t = 7$ onwards, the number of training examples produced by the approach increases by 1 at each timestep. Each new training example produced by the approach can be used to train (update) the forecasting model, so that it will produce improved forecasts over time. Even though it is possible to provide forecasts from timestep $t = 7$ onwards, it may be desirable to wait to start providing forecasts only after a predefined number of training examples η_{min} has been produced. This is because a forecasting model trained on too few examples may not perform well, and we would not have enough training examples to estimate its forecasting ability either.

Each forecasting model is built and updated based on a given machine learning algorithm. This algorithm may or may not be the same for different forecasting models. The choice of which machine learning algorithm to use for each quality attribute depends on which algorithm is able to provide better forecasts for that quality attribute. Our work investigates the use of two types of machine learning algorithms: (i) online learning models for stationary environments; and (ii) online learning models for non-stationary environments [46]). Online learning means that the models are updated over time based on new training examples produced over time. Learning algorithms for

stationary environments assume that the true function underlying the relationship between input attributes and forecast values does not change over time. Learning algorithms for non-stationary environments assume that this function may change, therefore adopting specific mechanisms to swiftly update forecasting models to such changes. More details on how that is achieved can be found in Section 2.2. We investigate the suitability of different learning algorithms to forecast quality attributes in Section 5.

A more detailed explanation of the steps followed by the proposed approach is provided below with respect to Algorithm 1. This algorithm is run for each quality attribute q.

Table 2. An Illustration of the Process of Creating Training Examples to Train the Forecasting Models. The table illustrates what training example is produced at each timestep, and what forecast can be provided at each timestep, when $\eta_{min} = 1$. The value of "-" in a cell represents the absence of a certain input attribute or forecast at a given timestep. A training example can only be created when the true value of all of of its input and output attributes has already been observed.

Current Timestep (t)	# Training Examples	Training Example Generated					Timestep Being Forecast
		Input attributes				Output Attribute	
		$q'_{dao_i}(t-6)$	$q'_{dao_i}(t-5)$	$q'_{dao_i}(t-4)$	$q'_{dao_i}(t-3)$	$q'_{dao_i}(t)$	$t+3$
1	0	-	-	-	-	1	-
2	0	-	-	-	-	2	-
3	0	-	-	-	-	3	-
4	0	-	-	-	1	4	-
5	0	-	-	1	2	5	-
6	0	-	1	2	3	6	-
7	1	1	2	3	4	7	10
8	2	2	3	4	5	8	11
9	3	3	4	5	6	9	12
10	4	4	5	6	7	10	13

1. **Provide the input parameters:** The architect will enter the following parameters to initialise the forecasting procedures: q (quality value q for which the forecasting model is being used), $q'_{dao_i}(t)$ (normalized quality values for each *dao* over time), ζ (number of input attributes required for forecasting), h (number of further ahead timesteps), η_{min} (minimum number of training examples to start using the forecasting models), and τ_q (error threshold for using the model). The two latter parameters are used to prevent using poor forecasting models. In particular, if the number of training examples used to build the forecasting models is too small, these models are likely to perform poor forecasts, and the estimation of their error will not be reliable either. Therefore, none of the forecasting models is used before being trained on η_{min} examples. In this case, the approach will behave reactively, as in Section 2.1. Once η_{min} training examples have become available, if the estimated error of a given forecasting model for quality attribute q is above τ_q, this model is not used at the current timestep and the approach behaves reactivily for this quality attribute. The architect has the flexibility to adjust the prior parameters. The impact of such parameters will be investigated in Section 5.
2. **Create training example:** If enough past quality attributes have been stored in the queue (line 6), a new training example will be created (line 7).
3. **Evaluate the forecasting model:** The current training example is used to update the estimate of the predictive performance of the forecasting model. Therefore, the forecasting model is first requested to provide a forecast for the current value of the quality attribute (line 10). As the current value of the quality attribute is known, we can check how large the

error of the forecast is by computing $e = \hat{q}_{dao_i}(t) - q'_{dao_i}(t)$ (line 11). Therefore, we can use this error to update the error average incurred by the forecasting model, based on some error metric (line 12). This is going to be used later on to decide whether to adopt the forecasts provided by this model.

The error metric used in this work is the Root Mean Squared Error, shown in equation 3.

$$e_q(t) = \sqrt{\frac{\sum_{t'=1}^{t} (\hat{q}_{dao_i}(t') - q'_{dao_i}(t'))^2}{\eta}} \quad (3)$$

where η is the number of training examples received so far. The error on the current training example (line 11) can be used to update $e_q(t)$ incrementally if desired. In this way, there is no need for storing all previous errors.

4. **Train the forecasting model:** The new training example $\{q'_{dao_i}(t-h-\zeta-1), q'_{dao_i}(t-h-\zeta), \cdots, q'_{dao_i}(t-h-1), q'_{dao_i}(t-h), q'_{dao_i}(t)\}$ is used to train its corresponding forecasting model (line 13).
5. **Provide a forecast:** If the error average incurred by the forecasting model is below the threshold τ_q and the number of training examples used to train the forecasting model is larger than η_{min} (line 14), the forecasting model is used to produce a forecast $\hat{q}_{dao_i}(t+h)$ (line 15-16). Otherwise, the algorithm returns *null* (line 17).

   When a forecast is produced, it is used to update the estimated benefit $(\hat{B}_{dao_i}(t))$ – (R3 in Figure 1), which is computed as in equation 4. In this context, $q^*_{dao_i}(t)$ is equal to $\hat{q}_{dao_i}(t+h)$. Otherwise, if a forecast is not produced, the approach behaves reactively and uses the actual normalized quality value instead (i.e. $q^*_{dao_i}(t) = q'_{dao_i}(t)$).

$$\hat{B}_{dao_i}(t) = \sum_{q \in Q} w_q * q^*_{dao_i}(t) \quad (4)$$

   w_q: is a ranking score set by stakeholders for each quality attribute q.

   The approach also allows the architect to use the average forecasts of $t+h$ instead of using a single value of h. For instance, if $h \in \{1, 5, 8, 15\}$, then $q^*_{dao_i}(t)$ will be equal to the mean of $\hat{q}_{dao_i}(t+h)$ for varying h values. The use of average forecasts may provide more realistic results than single h, as it considers a window of time in the future, rather than a specific point in the future. This potential advantage of using the average of the forecasts will be investigated in Section 5.2.

   Further, the approach provides the ability to check quality constraints violation. If $q^*_{dao_i}(t)$ violates the constraint, $\hat{B}_{dao_i}(t)$ is set to zero, otherwise equation 4 will be used to compute estimated benefit $\hat{B}_{dao_i}(t)$. In other words, if one or more of the constraints are violated, then the benefit is set to zero. The constraint violation is based on the type of quality attribute, for example if it is response time, then it may not exceed a particular value. If it exceeds, the dao is considered invalid and thus has a benefit of zero.

3.3 Detect Based on Forecast and Learn

At every timestep t, the detect module (Figure 1) triggers an alert if there is a significant change for the worse on the *forecast* benefit $\hat{B}_{dao_i}(t)$. Here, the change is detected with respect to the *forecast* exponentially smoothed benefits (R4 in Figure 1) rather than the *actual* ones [101]. We use the confidence interval of the maximum *forecast* exponentially smoothed benefit seen so far and its corresponding *forecast* exponential standard deviation; $[\hat{\mu}^{max}_{dao_i} - \alpha\sigma^{max}_{dao_i}, \hat{\mu}^{max}_{dao_i} + \alpha\hat{\sigma}^{max}_{dao_i}]$. $\hat{\mu}^{max}_{dao_i}$ is the maximum *forecast* exponentially smoothed benefit seen so far, $\hat{\sigma}^{max}_{dao_i}$ is its corresponding standard

ALGORITHM 1: ForecastAndLearn()

Input: quality attribute q for which to forecasting model is being trained, normalized quality value $q'_{dao_i}(t)$ observed at time t, number of input attributes ζ, number of further ahead timesteps h, minimum number of training examples for enabling forecasts η_{min}, error threshold per quality attribute τ_q

Output: forecast quality $\hat{q}_{dao_i}(t)$

```
1  if this is the first call to ForecastAndLearn then
       {queue stores past observations of quality values and η keeps track of the number of examples
        produced by the algorithm. Their values persist over future calls of ForecastAndLearn. }
2      Initialize Model[q].queue to empty.
3      Initialize Model[q].η = 0.
4      Initialize Model[q].error = 0
   end
5  Add q'_{dao_i}(t) to Model[q].queue.
6  if Size(Model[q].queue) == ζ then
7      Create new training example
        < q'_{dao_i}(t − h − ζ − 1), q'_{dao_i}(t − h − ζ), ··· , q'_{dao_i}(t − h − 1), q'_{dao_i}(t − h), q'_{dao_i}(t) >.
8      Model[q].η = Model[q].η + 1
9      Delete first element of Model[q].queue.
       {Evaluate the forecasting model by forecasting the output of the training example }
10     q̂_{dao_i}(t) = Model[q].Forecast(h, {q'_{dao_i}(t−h−ζ−1), q'_{dao_i}(t−h−ζ), ··· , q'_{dao_i}(t−h−1), q'_{dao_i}(t−h)})

11     e = q̂_{dao_i}(t) − q'_{dao_i}(t)
12     Use e to update the root mean squared error Model[q].error (Equation 3).
       {Train the forecasting model }
13     Model[q].Train(< q'_{dao_i}(t−h−ζ−1), q'_{dao_i}(t−h−ζ), ··· , q'_{dao_i}(t−h−1), q'_{dao_i}(t−h), q'_{dao_i}(t) >)
   end
   {Provide a forecast }
14 if (Model[q].error ≤ τ_q & Model[q].η ⩾ η_min ) then
15     q̂_{dao_i}(t + h) = Model[q].Forecast(h, {q'_{dao_i}(t − ζ), q'_{dao_i}(t − ζ + 1), ··· , q'_{dao_i}(t − 2), q'_{dao_i}(t − 1)})
16     Return q̂_{dao_i}(t + h)
   else
17     Return null.
   end
```

deviation. Whenever a new reading arrives at time t, those values are updated if $\hat{\mu}_{dao_i}(t) > \hat{\mu}^{max}_{dao_i}$. α is a parameter that affects the confidence level [54]. In particular, confidence levels 95% and 99% correspond to $\alpha = 1.96$ and 2.58, respectively.

Therefore, they are computed as follows:

$$\hat{\mu}_{dao_i}(t) = \theta\hat{\mu}_{dao_i}(t-1) + (1-\theta)\hat{B}_{dao_i}(t) \tag{5}$$

$$\hat{\sigma}^2_{dao_i}(t) = \theta\hat{\sigma}^2_{dao_i}(t-1) + (1-\theta) * (\hat{B}_{dao_i}(t) - \hat{\mu}_{dao_i}(t))^2 \tag{6}$$

$$\hat{\sigma}_{dao_i}(t) = \sqrt{\hat{\sigma}^2_{dao_i}(t)} \tag{7}$$

The architect also has the flexibility to adjust θ, α_1 and α_2 (Figure 2) with respect to required accuracy and stability. Algorithm 2 depicts the steps of change detection with respect to forecast

values. At each timestep, the approach updates the maximum forecast exponentially smoothed benefit $\hat{\mu}^{max}_{dao_i}$ (line 2-3) and its corresponding exponential standard deviation $\hat{\sigma}^{max}_{dao_i}$ (line 4). After that, the approach checks if a change/warning is detected. If the approach detects a significant change (line 5-6), the maximum forecast exponentially smoothed benefit and its corresponding standard deviation are reset (line 7). In this context, the select function will then search for an optimal *dao* based on the forecast exponentially smoothed benefit (Section 3.4). The approach also has the ability to just trigger warning to the architect based on the α value chosen (line 8-9). Otherwise, the approach indicates that neither a change nor a warning were detected (line 10).

ALGORITHM 2: DetectBasedOnForecastAndLearn()

Input: confidence intervals (α_1, α_2), forecast exponentially smoothed benefit $\hat{\mu}_{dao_i}(t)$, standard deviation $\hat{\sigma}_{dao_i}(t)$ based on forecast benefit
Output: change/warning/no change is detected
The red parameters denote the differences with respect to the reactive approach

{Initialize the maximum forecast exponentially smoothed benefit and its corresponding standard deviation}
1 $\hat{\mu}^{max}_{dao_i} = \hat{\mu}_{dao_i}(0)$, $\hat{\sigma}^{max}_{dao_i} = \hat{\sigma}_{dao_i}(0)$
{Update the maximum forecast exponentially smoothed benefit and its corresponding forecast exponential standard deviation}
2 **if** $\hat{\mu}_{dao_i}(t) > \hat{\mu}^{max}_{dao_i}$ **then**
3 $\quad\hat{\mu}^{max}_{dao_i} = \hat{\mu}_{dao_i}(t)$
4 $\quad\hat{\sigma}^{max}_{dao_i} = \hat{\sigma}_{dao_i}(t)$
end
{Check if a change is detected}
5 **if** $\hat{\mu}_{dao_i}(t) - \hat{\sigma}_{dao_i}(t) \leq \hat{\mu}^{max}_{dao_i} - \alpha_2 * \hat{\sigma}^{max}_{dao_i}$ **then**
6 a change is confirmed
7 reset $\hat{\mu}^{max}_{dao_i}$ and $\hat{\sigma}^{max}_{dao_i}$
{Check if a warning is triggered}
8 **else if** $\hat{\mu}_{dao_i}(t) - \hat{\sigma}_{dao_i}(t) \leq \hat{\mu}^{max}_{dao_i} - \alpha_1 * \hat{\sigma}^{max}_{dao_i}$ **then**
9 a warning is triggered
else
10 no change/warning is detected

3.4 Select based on Forecast and Learn

If a significant change is detected (based on its forecast exponentially smoothed benefit), then an optimal *dao* is selected based on its forecast quality values (R5 in Figure 1). In this context, it follows the same procedures of [101], but using the forecast quality values (Figure 2) rather than the observed ones. Algorithm 3 summarizes the steps and their differences with respect to reactive approach [101].

Step 1: *Determining which DAO are non-dominated by any other dao* [29, 44] (line 2-9). A given dao_i dominates dao_j iff:($c_{dao_i}(t) \leq c_{dao_j}(t)$ and $\hat{\mu}_{dao_i}(t) \geq \hat{\mu}_{dao_j}(t)$) and ($c_{dao_i}(t) < c_{dao_j}(t)$ or $\hat{\mu}_{dao_i}(t) > \hat{\mu}_{dao_j}(t)$). **Step 2:** *Calculating the marginal loss of the Non-Dominated DAO over time*, which is used for the purpose of computing the *expected* marginal loss (Step 3). In particular, we use a loss function $\hat{L}_\lambda(dao_i, t)$ that aggregates cost and *forecast* exponentially smoothed benefit into a single value (line 11), where $\lambda \in [0, 1]$ is a predefined parameter that controls the relative importance between cost and exponentially smoothed benefit. The loss describes how (un)desirable

ALGORITHM 3: SelectBasedOnForecastAndLearn()

Input: change detected in dao_{curr}, a predefined parameter λ, number of λ ($|\lambda|$), number of DAO ($|DAO|$), forecast exponentially smoothed benefit $\hat{\mu}_{dao_i}(t)$, cost $c_{dao_i}(t)$

Output: optimal dao_i

The red parameters denote the differences with respect to the reactive approach

{A change is detected in dao_{curr}, then do:}
{Initialization:}
1 $\lambda : \{0.1, 0.2, \cdots, 0.9\}$, $|\lambda| = 9$
{Determine the Non-Dominated DAO:}
2 **for** *i = 1 to* $|DAO|$ **do**
3 $\quad dominant = 0$
4 $\quad$ **for** *j = 1 to* $|DAO|$ **do**
5 $\quad\quad$ **if** $(c_{dao_i}(t) \leq c_{dao_j}(t) \& \hat{\mu}_{dao_i}(t) \geq \hat{\mu}_{dao_j}(t)) \& (c_{dao_i}(t) < c_{dao_j}(t) | \hat{\mu}_{dao_i}(t) > \hat{\mu}_{dao_j}(t))$ **then**
6 $\quad\quad\quad$ dao_i dominates dao_j
7 $\quad\quad\quad$ $dominant = 1$
$\quad\quad$ **end**
$\quad$ **end**
8 $\quad$ **if** $dominant = 0$ **then**
9 $\quad\quad$ Add dao_i to list of non-dominant DAO
$\quad$ **end**
end
{Calculate marginal loss for the non-dominant DAO:}
10 **for** *i = 1 to Length (list of non-dominant* $|DAO|$*)* **do**
11 $\quad$ **foreach** $\lambda \in \{0.1, 0.2, \cdots, 0.9\}$ **do**
$\quad\quad$ $\hat{L}_\lambda(dao_i, t) = \lambda c_{dao_i}(t) - (1 - \lambda)\hat{\mu}_{dao_i}(t)$
12 $\quad\quad$ **if** $i = argmin_i \hat{L}_\lambda(dao_{i'}, t)$ **then**
13 $\quad\quad\quad$ $\hat{L'}_\lambda(dao_i, t) = min_{(i \neq ii)} \hat{L}_\lambda(dao_{ii}, t) - \hat{L}_\lambda(dao_i, t)$
$\quad\quad$ **else**
14 $\quad\quad\quad$ $\hat{L'}_\lambda(dao_i, t) = 0$
$\quad\quad$ **end**
$\quad$ **end**
end
{Determine the expected marginal loss for the non-dominant DAO:}
15 **for** *i=1 to Length (list of non-dominant* $|DAO|$*)* **do**
16 $\quad$ $approxM[\hat{L'}_\lambda(dao_i, t)] = \frac{\sum_{\lambda \in \{0.1, 0.2, \cdots, 0.9\}} \hat{L'}_\lambda(dao_i, t)}{|\lambda|}$
end
17 Return optimal $dao_i = dao_i$ with $\max(approxM[\hat{L'}_\lambda(dao_i, t)])$

a certain *dao* is. After that, we use the loss function to compute the marginal loss $\hat{L'}_\lambda(dao_i, t)$, which represents how much worse the loss would be if dao_i was not available and we had to use the one with the second optimal trade-off, given a certain λ [29] (line 12-14). **Step 3:** *Determining the expected marginal loss of DAO over time.* The *dao* with the optimal trade-off between cost and benefit at time t is the *dao* with the maximum expected marginal loss at time t. This option can be suggested to the software architect as the optimal *dao* to be adopted at time t. This *dao* may or may not be the same as current option dao_{curr}. As in [29], we use an approximation of the expected marginal loss rather than the true marginal loss, to facilitate the choice of which solution from the

Pareto front to adopt in practice. In contrast, a single-objective problem would use a single fixed value for λ. However, we need to compute the marginal loss with a sample of different λ values. In our work, we used equally spaced values λ: $\{0.1, 0.2, \cdots, 0.9\}$, where $|\lambda| = 9$ is the number of λ values used. The expected marginal loss ($approxM[\hat{L'}_\lambda(dao_i, t)]$) can be approximated by taking the average of the marginal losses computed using different sampled values for λ [29] (line 15-16). The optimal dao_i is the one with maximum expected marginal loss (line 17).

Concerning the possibility to use several *dao* in parallel. In particular, our approach can work in several contexts:

1. When the architecture options are all simulated [1]. Since the real implementation is an expensive exercise and evaluation is pre-requisite to the implementation.
2. When the current architecture option is implemented and the other architecture options are simulated in parallel, to reduce deployment costs.
3. When the architecture option is implemented/supported but not always instantiated/active at run-time, continuous monitoring becomes expensive and instead we opt for monitoring every T' timesteps rather than every timestep, inline with our work [101] (also mentioned in Section 7). Alternatively, the architect may choose a "slice" of the architecture for further monitoring. This, for example, can take the form of execution traces, including components and connectors that may require further runtime evaluation.

When using simulation, the inputs of the simulator typically include information about the current state of the run-time environment, which enables our framework to take into account the environment where the system is deployed in order to provide an appropriate run-time evaluation. In our evaluation (Sections 4 and 5), we have simulated all the diversified architecture options and how they will behave in different environmental conditions. We then forecast their future values. This is used as an example, to show how our approach can apply Cases 1 and 2. Section 4.5 of [101] also explains the ways in which the exponentially smoothed benefit and standard deviation of the *DAO* can be obtained, including those not currently in use by the system.

4 IOT CASE STUDY DESIGN

In this section, we discuss the IoT case and its challenges, and how diversity can be embedded in the architecture.

4.1 IoT Case

We consider the case of architecting for IoT to demonstrate (i) how continuous evaluation, leveraging proactive approaches (e.g. forecasting analytics), can be realized and conducted for handling IoT challenges; (ii) what benefits proactive approaches can render to IoT architecture evaluation as opposed to reactive approaches.

Architecture evaluation for IoT is a particularly complex and interesting candidate for applying continuous evaluation. Let us focus the discussion on the following challenges ([42, 58, 66, 85]):

- *Heterogeneity*: These architectures are heterogeneous in nature; they consist of different types of things, such as static sensors, mobile crowdsensing (e.g. cellular-based, or vehicle-based sensors), virtual services (e.g. web services), and social sensing (e.g. share data across social networks like Facebook);

[1] We also recognise that there is a threat to validity in that the real environment may differ from the simulated data due to uncertainties at run-time. This threat was considered when we investigated the robustness of our approach to noise [101], to mimic variations and fluctuations in real settings and to check how well the approach can handle this issue. Our proposed proactive approach was superior to the existing reactive approach in most cases, demonstrating a good behaviour on this issue.

- *High dynamism*: Due to the presence of mobile things and uncertainty of their resource demand and QoS provision over time (i.e. service level objectives), varying energy consumption per thing and their varied availability;
- *Scale*: is a another challenge, where their ubiquitous, light, and mobile nature has led to the presence of many millions of things.

Our IoT application extends Gupta et al. [60]'s application – an urban traffic monitoring system, named *iTransport*. Gupta et al. [60] used a video surveillance application of non-trivial scale to demonstrate the usefulness of their proposed cloud/fog simulator tool *iFogSim*. However, in their case study, the context of run-time architecture evaluation and adaptability under time-varying environment have not been considered, even though iFogSim is capable of simulating the dynamics and uncertainty of cloud/fog environments. This has motivated us to extend their case study. In a nutshell, iFogSim is a cloud/fog simulation environment; it can aid developers to simulate the impact of their application on qualities of interest. It forms the basis in our work to mimic the dynamics and uncertainty of cloud/fog environments, and their impact on qualities of interest in our case study. Further explanation related to our use of iFogSim, which differs from that of Gupta et al., can be found in Section 4.4. In addition, Gupta et al. [60] assume the presence of one application architecture (i.e. configuration). We have designed the multiple configurations with respect to the common architecture decisions of a video surveillance application

iTransport uses smart cameras, which are either fixed or mobile (attached to vehicles and bikes) to capture the traffic for accident avoidance and traffic management. The application has 6 modules as seen in Figure 3: camera, motion detector, object detector, object tracker, accident storage, and emergency control. Smart cameras transmit raw video streams to the motion detector module, which then forwards the video in which motion was detected to the object detector module. The object detector module analyzes the objects and detects any abnormal actions (i.e. car accidents). If it observes an accident, the emergency control searches for a nearby ambulance for notification. The data is then sent to the accident storage cloud to profile the accidents with respect to areas.

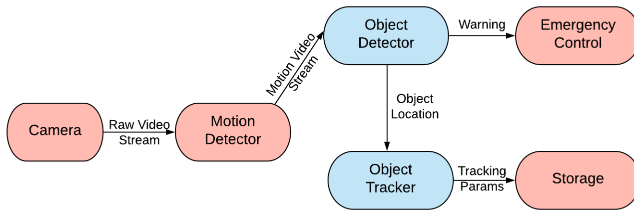


Fig. 3. Data flow diagram of the *iTransport* application [24].

4.2 Diversified Architecture Options In The Context Of IoT Case

When architecting the *iTransport* application, the architects must address uncertainties due to heterogeneity of the things; the dynamicity of the things' behaviors and the dynamism of their composition. The architects have employed design diversification strategies [9, 10, 15, 100] to respond to the challenges and to handle uncertainties: the greater the uncertainty, the more diversity the architects have applied to attempt to improve performance. Since there are many ways to diversify, each diversified architecture can be treated as a candidate option, which we denote by *dao* [100]. A *dao* implements a set of diversified decisions to meet some quality goals and trade-offs. Given a set of architecture decisions D, where a decision $d_{ka} \in D$; d_k denotes a particular

capability, including connectivity, data collection, data management, *etc*; and d_{ka} indicates the software architecture components and connections that implement this capability. For example, the architects decided to diversify the *data collection* capability (d_1), where video could be captured using fixed cameras (d_{11}), mobile cameras (d_{12}), or both (d_{13}). Another diversification decision is concerned with the *connectivity and processing* capability (d_2), where the things can connect, track, and process the captured video on the cloud (d_{21}) or both cloud and fog (d_{22}). So dao_1 comprises d_{11} and d_{21}, whereas dao_2 consists of d_{12} and d_{22} and so forth. Table 3 depicts the selected options, with decisions designed for cloud, fog, mobile, or fixed.

Table 3. Possible Diversified Architecture Options for *iTransport* application. The diversity in each *dao* can refer to using fixed and/or mobile fog devices for data collection capability; using different cloud providers and heterogeneous fog devices for data processing capability.

Decision d_k / Option dao_i	Data Collection Capability (d_1)	Data Processing Capability (d_2)
1	Fixed (d_{11})	Cloud (d_{21})
2	Mobile (d_{12})	Cloud (d_{21})
3	Fixed and Mobile (d_{13})	Cloud (d_{21})
4	Fixed (d_{11})	Fog and Cloud (d_{22})
5	Mobile (d_{12})	Fog and Cloud (d_{22})
6	Fixed and Mobile (d_{13})	Fog and Cloud (d_{22})

4.3 Challenges

In *iTransport*, there are several design trade-offs concerning the critical QoS attributes (e.g., response time, energy consumption, network usage, *etc*) and cost, subject to constraints such as the predefined coverage and availability of the things. In the context of *iTransport*, we consider deployment cost (the expenses related to the infrastructure deployment in cloud/fog environment), execution cost (the computational costs of running the processing tasks on cloud/fog devices), and networking costs (related to the bandwidth requirements and associated expenses). For instance, data uploading cost from end devices/sensors and inter-nodal data sharing cost) [80]. Further, the switching costs in *iTransport* embrace the migration costs to/from the cloud/fog, thing's connectivity and other costs (if any). Nevertheless, the approach is flexible enough to include other costs. The design trade-offs can inform the diversification design decisions and the deployment of *dao*. Addressing the following scenarios require us to consider trade-offs when deciding on *dao*:

- dao_1 uses fixed camera sensors to provide more stable and better response time to fulfill the predefined coverage. However, achieving coverage at the scale of highways using just fixed cameras may incur higher costs and static coverage. In contrast, the use of mobile crowdsensing (using smart vehicles) [66], as in dao_2, could be an alternative solution due to its low cost. But the mobile crowdsensing in dao_2 may be unstable in terms of response time and it can consume much more power at this scale due to the simultaneous transmission, processing, and remote execution of the images on the cloud. Further, availability in dao_2 is more restricted than that of dao_1.
- When comparing with the case where the cloud is used as the sole computation paradigm (in dao_1 to dao_3), the partial use of fog (in dao_4 to dao_6) could provide faster response time (e.g., for emergency notification and online analytics) and lower network usage, due to the offloading of computation to nearby fog devices. But it may incur more energy consumption, given the large number of required fog things and the additional overhead that may be

required to synchronize and store the processed information on the cloud. In addition, the fog option needs to fulfill the constraints on the proximity of the thing to the fog and the availability of the fog.

The design-time evaluation and reactive learning are not enough to handle complexity of *DAO* trade-offs (mentioned in prior scenarios). Because there are some situations, where the design-time knowledge and reactive learning may not be able to choose the "suitable" options. This is due to the run-time uncertainties and dynamics caused by various environmental factors, which emergently affects the benefit of the options. Even though the use of reactive learning is better in some scenarios as compared with design-time evaluation [101], yet there are other scenarios which require future knowledge (discussed in Section 5).

So the question here is *do proactive approaches using forecasting analytics provide useful information for evaluation which reactive approaches may miss?* By focusing on economic gains/losses, without forecasting which architecture decisions have the potential to materialize into future economic gains, IoT systems cannot keep up with the rate at which these potentials appear/disappear in dynamic IoT environments. In particular, by the time the reactive approach suggests an architecture decision in response to a detected problem (i.e. the current architecture is getting worse), the problem itself may cease to exist (i.e. the current architecture is improving). Therefore, a reactive approach can lead to missing opportunities for future economic gains. Similarly, without forecasting whether architecture decisions will continue to deliver value for a long time or not, unnecessary costs can materialize into economic losses for the IoT system when a decision is implemented. The opportunity for future economic gains which should have been enabled by this decision (i.e. recommended by the reactive approach) can disappear. In that situation, implementing such a decision is useless and incurs nothing but an economic loss. We aim to illustrate how proactive approaches using forecasting analytics can handle the prior scenarios, in contrast to design-time and reactive approaches (Section 5).

4.4 Data Synthesis and Experimental Environment

The data synthesis process is performed using iFogSim, whereas the Massive Online Analysis framework (MOA) [22] and Matlab are exploited for data analysis.

4.4.1 iFogSim. At design-time, the architect has followed the procedure of Section 4.2 to preliminary decide on the *DAO* (and their composing *D*) for implementing this functionality. To simulate the qualities of interest of each architecture decision over time, we adopt the *iFogSim* [60] tool. This tool is a high-performance open source toolkit for fog computing, edge computing and IoT, which builds on Cloudsim [32]; it provides the architect with the freedom of hierarchically composing the fog devices, clouds, and data streams. The iFogSim and CloudSim simulators provide the flexibility to simulate the dynamics of architectures in an uncertain environment, without implementing any machine learning algorithms which would aid the architect in determining how the architecture options will behave in the future. In this context, our approach used iFogSim to gather the QoS related to architecture options. Nevertheless, our framework can use other simulators, as seen fit by the evaluator and suitable for the architecture under investigation.

In iFogSim, we have hierarchically composed the application as shown in Figure 3. The candidate *DAO* used in this study are shown in Table 3. In particular, each *dao* is composed of different types of data collection (type of sensors) and connectivity (computing locations) as architecture decisions, meaning that the processing performed by each *dao* is executed differently. Connectivity was simulated by either executing the *object detector* and *tracker* modules (shown in Figure 3) in the cloud and/or fog. For data collection, two gateways were used, where each gateway is connected to an average of 50 smart cameras (Figure 5): fixed and/or mobile, having a total of 100 fog devices.

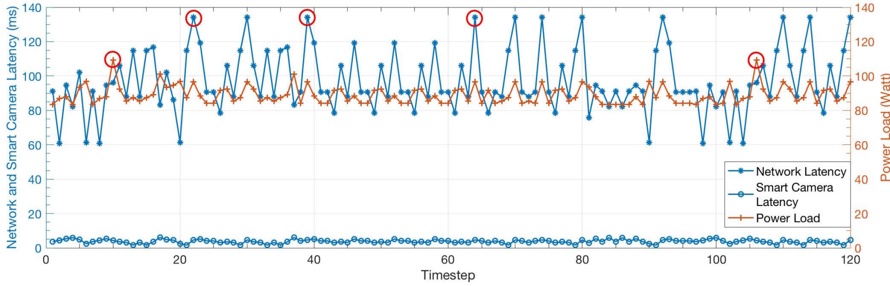


Fig. 4. Sample of changing environmental conditions facing *iTransport* (i.e. input for one of the *DAO*). *The red circles denote some examples of accidental spikes.*

The fog devices and cloud are configured based on [59, 60]. Note that iFogSim internally computes the QoS of each component in a *dao* and it then outputs the overall QoS of this *dao*. Therefore, we do not need to compute the aggregate functions from Table 1 separately – the simulator already provides the aggregated results for us.

The goal is to continuously optimize the following two conflicting requirements:

- **Benefit:** It needs to be maximized and is based on three quality attributes of interest:
  1. *Response Time: iTransport* is time-critical application, so it should respond as early as possible. In *iTransport*, the response time (RT) of an application is the application's end to end delay (in milliseconds *ms*) and measured using iFogSim.
  2. *Network Usage:* High network usage would cause network congestion, so it should be as low as possible. The network usage (NU in mega bytes *MB*) is also measured using iFogSim.
  3. *Energy Consumption:* IoT applications could, in theory, consume arbitrary amounts of energy and therefore the energy needs to be carefully controlled. In this context, when designing IoT architectures, architects have to consider energy consumption as a major concern [72]. In *iTransport*, the energy consumption (EC in mega joules *MJ*) is the total energy consumption by all the devices in the application, which is also determined through iFogSim.
- **Cost:** It needs to be minimized. It is a composite of operating cost, and switching cost that is added once we switch. In the context of *iTransport*, the average cost encompasses a thing's connectivity (includes switching), execution in the cloud and/or fog, and other costs mentioned in Section 4. The mean overall costs have been collected from iFogSim.

The iFogSim tool takes as *input*: the network latency, power load, smart camera's latency (i.e. fog devices), number of cameras, number of gateways, tuple configurations, cloud and fog devices configurations. The sources of uncertainty in *iTransport* come from the varying network delays due to network congestion. There are other factors, which as well impact the environmental conditions, such as uncertainty in QoS of cloud service providers (e.g. Amazon, Google Cloud, *etc*) and software-defined capabilities of fog service providers in terms of their processing power, as well as the hyperconnectivity of the nodes. In this context, we have generated data corresponding to each diversified architecture option including typical as well as worst case scenarios. For instance, we varied the power load as 80-110 watt with a fluctuation of 5-10%, following the typical power load values from [59]. The latency was varied in the range of low to high, i.e. 1-6ms with a fluctuation of 20-30% on the smart camera. We also varied the network latency with an average of 100*ms* and fluctuation of 20-25%, as exemplified in Figure 4. The choice of this fluctuation was based on [37],

as it normally provides acceptable throughput across various networking protocols, but also causes accidental spikes that represent worst case scenarios.

We also simulated changes by using diverse smart cameras' configuration [59, 60], as depicted in Table 4. Based on that, iFogSim outputs the energy consumption of devices, application's response time, and network usage. This setting was intentionally designed as a worst case that goes beyond a stable setting. Further, the iFogSim takes the pricing configurations for IoT devices (i.e. fog devices) and cloud to generate the mean costs of each *dao* (i.e. application architecture). All the pricing configurations are used with respect to AWS IoT services [11]. We have run the simulations for each *dao* for 120 timesteps.

Table 4. Simulation Parameters for iTransport. *Note that StrictEC case: low energy consumption is favoured over response time and network usage; StrictRTNU case: low response time and low network usage are favoured over energy consumption; StrictALL case: the user constrained the application to favor the three QoS in a strictly manner.*

Parameter	**Value**
Device Configurations – (iFogSim Input)	**CPU (GHz), RAM (GB), Cost ($/day)**
Cloud Datacenter	3, 40, 0.1-0.3
Wifi and ISP Gateways	3, 4,0.0053-0.0056
Smart Camera	[1.6, 1.867, and 2.113], 4, 0.0053-0.0056
Number of Smart Cameras	80-120 cameras
Network Configurations (From-To) – (iFogSim Input)	**Avg Latency**
ISP Gateway-Cloud Datacenter	80-120 ms
Wifi Gateway-ISP Gateway	1-6 ms
Smart Camera-Wifi Gateway	1-6 ms
Tuple Configurations (Msg. size) – (iFogSim Input):	**CPU (MIPS), Network (Bytes)**
Raw Video Stream:	1000, 20000
Motion Video Stream:	2000, 2000
Object Location:	500, 2000
Warning:	1000, 100
Tracking parameters	28,100
Average QoS – (iFogSim Output):	
Applications Response Time (RT)	300-4000 ms
Energy Consumption of devices (EC)	80-120MJ
Network Usage (NU)	500 KBytes- 2 MBytes
Evaluation Settings:	
θ	0.9
α_1, α_2	$\{95, 99\%\}$
Detection timestep	5
Normal(Constraint[RT, EC, NU]; Weights[w_{RT}, w_{EC}, w_{NU}])	[400ms, 130MJ, 1Mbps]; [0.4,0.3,0.3]
StrictEC(Constraint[RT, EC, NU]; Weights[w_{RT}, w_{EC}, w_{NU}])	[4000ms, 110MJ, 2Mbps]; [0.2,0.6,0.2]
StrictRTNU(Constraint[RT, EC, NU]; Weights[w_{RT}, w_{EC}, w_{NU}])	[350ms, 130MJ, 500Kbps]; [0.4,0.2,0.4]
StrictALL(Constraint[RT, EC, NU]; Weights[w_{RT}, w_{EC}, w_{NU}])	[380ms,120MJ, 800Kbps]; [0.4,0.3,0.3]
Normalized Operating Cost	$\{0.3 - 0.5\}$
Normalized Switching Cost	$\{0.2 - 0.4\}$
Error Threshold τ_q	[0.15(RT),0.25 (EC),0.20 (NU)]

4.4.2 Implementation and Experimental Environment. To test and evaluate the forecast models, we have used MOA [22]; an open source framework for data stream mining that comprises a group of online methods. This framework receives the input attributes (i.e. list of data quality observations), and then it outputs the forecasts with respect to the selected forecasting models. We also used Matlab to implement the proposed approach. All experiments were executed on Intel Core i7 processor machine with 16GB of RAM. Further details on the experimental design used for each part of the experiments are provided in Sections 5.1 and 5.2.

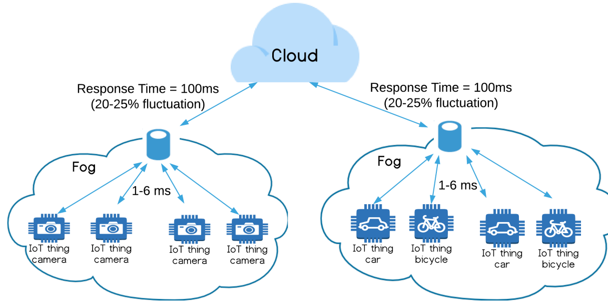


Fig. 5. The Initial Experiment Configuration for iFogsim.

5 EXPERIMENTAL EVALUATION

We performed experiments with the aim of evaluating the proposed proactive approach and its worthiness through architecting the urban traffic monitoring system *iTransport* described in Section 4. The experiments are divided into two parts. The first part (Section 5.1) focuses on evaluating the forecasting ability of the forecast models used by the proposed proactive approach and understanding the conditions under which they do or do not perform well. Together with the proposed approach as described in Section 3.2, it provides the answer to RQ1: *How proactivity in continuous evaluation, when supported by simulated instances of the system-to-be, can be realized and conducted? For instance, can continuous time series forecasting analytics, leveraging simulated instance of the system-to-be, enable us to predict the behaviour of software architectures over time? If so, how well?* (outlined in Section 1). The second part (Section 5.2) investigates whether and when it is worth adopting the proposed proactive approach for architecture evaluation. Together with the proposed approach as described in Sections 3.3 and 3.4, it provides the answer to RQ2: *How can proactive approaches complement reactive ones to provide a well-rounded and more effective continuous architecture evaluation, when supported by simulated instance of the architecture? What benefits can they bring to continuous software architecture evaluation and decision-making at run-time?* (outlined in Section 1). Section 4.4 explained the experimental setup.

5.1 Forecasting Ability of the Proposed Proactive Approach

In this section, we aim to provide answers to **RQ1**. This question is further divided into the sub-questions below. First, we show how to choose the suitable experimental parameters for enhancing the evaluation of architecture decisions in our experiment followed by some recommendations for the architect on how to apply this in practice.

***RQ1.1:** How to determine the number of input attributes needed to provide acceptable forecast?*
Motivation: We need to understand what are good values for the number of input attributes ζ, i.e., what ζ values yield the smallest forecasting errors.

Experimental Design: We use the *kNN learner* against different numbers of input attributes: 2, 6, and 14. These choices aimed to show how low, mid, and high values could affect the forecasting error. Forecasting error is measured based on the mean absolute error ($MAE = \frac{\sum |\hat{q}_{dao_i}(t) - q'_{dao_i}(t)|}{\eta}$) and root mean square error ($RMSE = \sqrt{\frac{\sum (\hat{q}_{dao_i}(t) - q'_{dao_i}(t))^2}{\eta}}$), where η is the number of training examples, $q'_{dao_i}(t)$ is the actual normalized quality for a *dao* as seen in Section 3.1, and $\hat{q}_{dao_i}(t)$ is the quality attribute forecast as seen in Section 3.2.

The simulations are performed for 120 timesteps. Therefore, if $\zeta = 2$, then $\eta = 118$, if $\zeta = 6$, then $\eta = 114$ and if $\zeta = 14$, then $\eta = 106$. The comparison of the different numbers of input attributes will be supported by non-parametric Friedman tests [45]. This test is a rank-based test widely used for comparisons across multiple groups. In our case, each group is a given number of input attributes for a given quality attribute, and each observation within a group corresponds to the forecasting error obtained for each of the six *DAO*. The null hypothesis is that all groups have similar forecasting error. The alternative hypothesis is that at least one of the groups has different forecasting error. The same procedure was repeated for Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) as forecasting error metrics, and for each quality attribute. The level of significance of the test was set to 0.05.

Table 6 reports the forecasting errors for the group of input attributes, whereas Table 5 shows their average ranking and p-value. If the null hypothesis is rejected, we will perform the post-hoc tests, Bonferroni-Dunn [45], between each group and the top ranked one to determine which groups are significantly different from the best ranked one.

Table 5. Average ranking and p-value for determining the statistical difference between the use of 2, 6, and 14 input attributes. *Note that MAE denotes Mean Absolute Error and RMSE denotes Root Mean Square Error.*

# of Input Attributes	Average Ranking of input attributes					
	Response Time		Energy Consumption		Network Usage	
	MAE	RMSE	MAE	RMSE	MAE	RMSE
2	1.17	1.50	1.17	1.50	1.50	1.67
6	1.17	1.50	1.00	1.17	1.50	1.67
14	1.17	1.00	1.67	1.67	1.17	1.33
p-value	**1**	**0.3679**	**0.2466**	**0.2466**	**0.3679**	**0.8187**

The input attribute with minimum (not necessarily significantly better) MAE and the minimum RMSE (not necessarily significantly better) are highlighted in light gray.

Analysis: Based on Table 6, the p-value of MAE is 1 (response time), 0.2466 (energy consumption), and 0.3679 (network usage). So the null hypothesis of MAE is not rejected, for all quality attributes. This implies that, for all the quality attributes, there is no significant statistical difference between the number of input attributes. Same applies for RMSE with p-value of: 0.3679 (response time), 0.2466 (energy consumption), and 0.8187 (network usage).

To decide the number of input attributes to be used in the remaining of the experiments, we have exploited their Friedman rankings. Based on Table 5, the rankings of varying input attributes are very similar. However, $\zeta = 6$ provided lower error for energy consumption quality than the rest. Whereas, $\zeta = 14$ produced more favorable outcome for response time than the others, but it requires 14 timesteps before the initial forecasts can start being made. Therefore, we decided to

adopt $\zeta = 6$ in the remaining of the experiments. We recommend a similar procedure to be followed to decide what ζ to adopt in practice.

Overall Observations: *The forecasting models seem to be quite robust to different values of ζ, given that the rankings were all very similar. So, architects may not need to fine tune it so much.*

RQ1.2: *How to determine the number of training examples required for the forecasting error to become acceptable?*

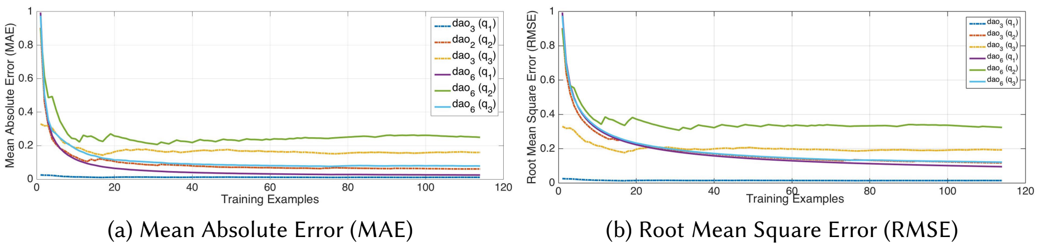
Motivation: In this experiment, we aim to determine the minimum number of training examples η_{min} (i.e. forecast quality values) after which the method could use the forecasts $\hat{q}_{dao_i}(t)$ made for each quality attribute to compute the forecasting benefit for selection of a *dao*. This reveals how many timesteps are typically required for the forecasts to be considered reliable and their error estimates to stabilize. Before that, the small number of examples could lead to poor forecasts and unreliable error estimates. In this context, this experiment will show to the architect how to choose the suitable number of training examples after which forecasts can start being used.

Experimental Design: In this experiment, the Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) metric will be used to measure the error between the actual quality attribute values and their forecasts for various h values, where $h \in \{1, 5, 8, 15\}$. We will plot the average MAE (Figure 6a) and average RMSE (Figure 6b) over time for kNN learner for t+1. Whereas other plots related to $t + h$ are shown in Appendix A for reference. The results for dao_3 and dao_6 are reported here, whereas the other *DAO* showed similar results.

Analysis: As depicted in Figure 6a and 6b, the forecasting error sharply reduces over the beginning of the learning period, reaching a more stable value after 13-15 training examples, for all *DAO* for $t + 1$. Same applies for other $t + h$, where $h \in \{5, 8, 15\}$. Therefore, η_{min} does not need to be a large value. In our remaining experiments, we will adopt $\eta_{min} = 15$.

Despite the sharp decrease in the error during the initial stage of the learning, how low the error can get depends on the *dao* and quality attribute being forecast. In practice, once the error estimates are deemed more reliable (i.e., once η_{min} is reached), the parameter τ_q can be used to prevent the adoption of forecasting models whose forecasting error is deemed high.

Overall Observations: *The error sharply reduces over the initial learning period. Therefore, η_{min} does not need to be a large value. In these experiments, $\eta_{min} = 15$ was a reasonable value.*



(a) Mean Absolute Error (MAE) (b) Root Mean Square Error (RMSE)

Fig. 6.

Table 6. Evaluation for kNN learner for 6 *DAO* (2, 6, and 14 input attributes). *Note that MAE denotes Mean Absolute Error and RMSE denotes Root Mean Square Error.*

# of Input Attributes	**Error Metric in *Response Time* for 6 DAO**											
	dao_1		dao_2		dao_3		dao_4		dao_5		dao_6	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
2	0.01	0.01	0.15	0.18	0.01	0.01	0.05	0.17	0.06	0.18	0.06	0.18
6	0.01	0.01	0.15	0.18	0.01	0.01	0.05	0.17	0.06	0.18	0.06	0.18
14	0.01	0.01	0.01	0.01	0.01	0.01	0.06	0.17	0.06	0.17	0.06	0.18
# of Input Attributes	**Error Metric in *Energy Consumption* for 6 DAO**											
	dao_1		dao_2		dao_3		dao_4		dao_5		dao_6	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
2	0.09	0.16	0.13	0.21	0.10	0.18	0.17	0.20	0.23	0.29	0.25	0.34
6	0.08	0.14	0.13	0.21	0.10	0.18	0.17	0.20	0.23	0.29	0.25	0.34
14	0.09	0.16	0.15	0.21	0.10	0.19	0.21	0.26	0.28	0.34	0.18	0.23
# of Input Attributes	**Error Metric in *Network Usage* for 6 DAO**											
	dao_1		dao_2		dao_3		dao_4		dao_5		dao_6	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
2	0.15	0.18	0.14	0.19	0.17	0.20	0.08	0.15	0.11	0.19	0.11	0.19
6	0.15	0.18	0.14	0.19	0.17	0.20	0.08	0.15	0.11	0.19	0.11	0.19
14	0.16	0.21	0.14	0.18	0.17	0.20	0.08	0.16	0.11	0.17	0.11	0.18

The input attribute with minimum (not necessarily significantly better) MAE and the minimum RMSE (not necessarily significantly better) are highlighted in lightgray.

RQ1.3: *Is it necessary to adopt learning algorithms for non-stationary environments?*
Motivation: This experiment aims to show which type of learning algorithms (from the ones introduced in Section 2.2) could be adopted for non-stationary environments, such as IoT. The architect can benefit from this experiment to select the most suitable learner for each quality attribute.
Experimental Design: In this experiment, we will evaluate the performance of the sole use of single learners for stationary (kNN, Perceptron, and SGD Multiclass) and non-stationary environments (FIMTDD, and FTM). We will also analyze the ensemble learner (AddExp) for non-stationary environments by adopting each of the prior single learners as experts to AddExp. Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) are utilized as error metrics to evaluate the learners' performance. Six attributes (i.e. ζ) are used as input for the forecasting method based on Experiment RQ1.1. We have also investigated the case of parameter tuning for each learner. However, these set of trials did not lead to any significant improvement for the forecast error. Therefore, we adopted the default values for AddExp and single learners except for FIMTDD. The final parameters used in this experiment are depicted in Table 12.
Analysis: Based on the experiment, the kNN, FIMTDD, and FTM (used with/without AddExp) provided the minimum errors for all the *DAO*. The Friedman test was conducted to check whether there is a significant statistical difference between groups. A MAE p-value of 5.95×10^{-7} (response time), 9.00×10^{-8}, (energy consumption) and 2.37×10^{-8} (network usage); a RMSE p-value of 1.60×10^{-7} (response time), 2.81×10^{-7}, (energy consumption) and 1.43×10^{-7} (network usage) indicated: *the null hypothesis is rejected (i.e., the learners are significantly different in terms of MAE and RMSE*).

The kNN learner was the one that obtained the top Friedman rankings. By employing the Bonferroni-Dunn test [45] between each learner and kNN, we found that in most of the quality attributes for both RMSE and MAE, there is a significant difference between kNN (top ranked) and two learners (SGD Multiclass and AddExp with SGD Multiclass). For RMSE, there was also a significant difference between kNN and two learners (Perceptron and AddExp with Perceptron). To this regard, kNN provided better forecasting error than the following learners: SGD Multiclass, AddExp with SGD Multiclass, Perceptron and AddExp with Perceptron.

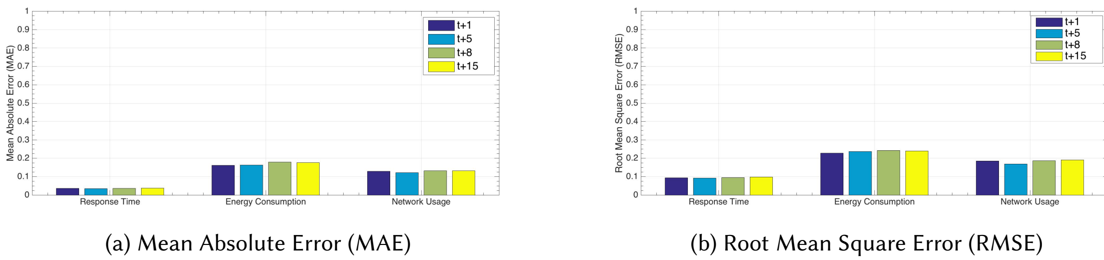
Interestingly, kNN is an algorithm for stationary environments, whereas the algorithms for non-stationary environments did not perform best. This suggests that, in this domain, even though the input attributes themselves suffer changes over time, such changes do not affect the relationship between input attributes and the output value being forecast. Therefore, machine learning algorithms able to cope with changes in the relationship between input attributes and the output value were not necessary. When such algorithms are not necessary, they can be detrimental, because they can confuse noise with changes in the relationship between input attributes and output. It is worth noting that dao_1, dao_2, and dao_3 had less error than other *DAO*. This is due to the use of cloud rather than fog-cloud, which resulted in lower fluctuation in QoS than fog-based ones. This aided the experts to forecast quality values close to the actual ones. However, the energy consumption for dao_5 and dao_6 was highly fluctuating, which affected the performance of forecast models.

Overall Observations: *Even though IoT is a non-stationary environment that will cause the quality attributes to significantly vary over time, our experiments indicate that the relationship between input attributes and output does not vary, not requiring forecasting algorithms for non-stationary environments.*

RQ1.4: *How far ahead can we predict the future benefit based on the quality attribute forecasts?*
Motivation: This experiment could help the architect in improving the forecasts by tuning h parameter and showing to what extent the forecasts could be beneficial.

Experimental Design: In this experiment, the forecasting method is evaluated with respect to anticipations made for timesteps further ahead, such as t + 1, t + 5, t + 8, and t + 15. The Mean Square Error (MAE) and Root Mean Square Error (RMSE) metric are also exploited to measure the difference between the forecast and actual value, and how the latter metric varies over time. In this context, a bar plot is used to compare between four categories of forecast (Figure 7a and 7b). Further, the Friedman test is also conducted to investigate the statistical significance of the differences between different choices of h values.
Analysis: As depicted in Figure 7a and 7b, the MAE and RMSE are slightly increasing when h increases. For instance, for the response time quality, MAE is almost 0.035 (t+1), 0.035 (t+5), 0.036 (t+8), and 0.038 (t+15). The Friedman test confirms that there is no statistical difference between different h values (p-value = 0.0602). This indicates that the forecasting model is actually able to provide competitive forecasts for timesteps further ahead in the future.

Overall Observations: *The errors obtained by the forecasting models varied depending on the quality attribute being forecast and the dao being monitored. The MAE was smaller than 0.1 in several cases. Since most of the quality attributes when normalised, ranged from approximately 0.2 to 0.9. Therefore, a MAE of 0.1 is considerably low, indicating that forecasts are possible and thus could potentially be used to enable proactivity.*



(a) Mean Absolute Error (MAE)

(b) Root Mean Square Error (RMSE)

Fig. 7.

5.2 Usability and Efficiency of the Proactive Approach

In this section, we aim to show how the forecasting method could be applied in *iTransport* and evaluate its performance against other approaches. This series of experiments aims to provide answers to **RQ2**, which is further elaborated through several sub-questions explained next.

RQ2.1: *What is the performance of the proactive approach in comparison to the reactive approach when using short-term forecasts (i.e., $h = 1$)?*
Motivation: This experiment examines the proactive approach in comparison to the reactive approach in the scenario where the proactive approach is the closest to the reactive one, i.e., when the forecast is a very short-term forecast ($h = 1$). It reveals to what extent forecasts, even if short-term, can be beneficial to run-time architecture evaluation. In particular, it shows whether the *dao* suggestions by the proactive approach (even if short-term forecasts) in comparison to the reactive approach could lead to improvement in mean exponentially smoothed benefit, mean cost, and application's stability.
Experimental Design: For the *Proactive Informed-selection* approach (Section 3), the change detection and the selection of optimal *dao* to switch to is according to the *forecast* values. The

Reactive Informed-Selection approach (Section 2.1) evaluates, detects the change, and selects the optimal *dao* based on the *actual* quality values.

We examine two scenarios for evaluation: (Best Case) The design-time evaluation approach suggests the systems to start operation using the *dao* that are believed to provide the optimal cost-benefit trade-offs at run-time (i.e. dao_6); (Worst Case) The design-time evaluation approach suggests the systems to start operation using the *dao* that turns out to provide the worst balanced cost-benefit trade-offs at run-time (i.e. dao_2). This was for all the cases, except for StrictEC case (low energy consumption is favoured over response time and network usage), where the design-time knowledge has recommended dao_3 which turned out to be the best, whereas dao_2 turned out to be the worst. We have developed these scenarios to demonstrate how the reactive and proactive approaches can deal with extreme scenarios. In particular, we assume that the stakeholders have a very good knowledge about the *dao* and its dynamicity for best case and vice versa. In addition, we have generated the design-time evaluation based on the gathered run-time information. For instance, we found that dao_6 almost provides the best QoS over time in most of the cases and so forth.

For this experiment, we have used the following parameters based on experiments in Section 5.1. More specifically, the number of input parameters ζ is 6 (Experiment RQ1.1), the minimum number of training examples η_{min} is 15 (Experiment RQ1.2), and the adopted learning algorithm is kNN (Experiment RQ1.3). The choice of the prior parameters is based on the minimum acceptable forecasting error. We have also set the error threshold τ_q to [0.15 (RT),0.25 (EC),0.20 (NU)] based on the average forecasting error for t+1 (Experiment RQ1.4). For this experiment, we only tested the two approaches for number of further ahead timesteps $h = 1$, whereas other values for h will be evaluated in the next experiment.

Regarding the IoT context, the use of fog computing and cloud computing in mobile crowd sensing applications is highly affected by the QoS requirements. Consider a scenario where the stakeholders require the *iTransport* application to quickly track the accident in city center, especially in rush hours. In this context, low response time and network usage is higher concern rather than energy consumption. Therefore, the ranking score (i.e. weights) for response time and network usage qualities are higher than energy consumption. Also the use of fog-cloud computing is advisable over cloud computing. So object detector and tracker modules will be executed in the fog. We are uncertain about the network latency (due to dynamic traffic and variable load) and mobility of devices (nodes join/leave the network). This will cause instability, which may require a switch to another architecture option.

We constructed four different scenarios for evaluation: *normal* and *strict* (3 cases) constraints, which are introduced in Table 4. For instance, we consider that the architect has constrained the application to handle the request in less than *400 ms* and network usage does not exceed *1Mbps*; whereas, the energy consumption should not exceed *130 MJ* (normal case) and so forth. These cases are also used in the next experiment. Historical performance and experts' judgment can inform the adjustment of the prior constraints [1, 60]. The architect can adjust the quality weights to reflect priorities from stakeholders. For example, for the normal case, we assume a w_{RT} of 0.4 for response time, w_{EC} of 0.3 for energy consumption, and w_{NU} for network usage of 0.3. We have used the typical experiment settings for the method presented in Table 4.

In order to check how good the proactive approach is compared to the reactive approach, we compare their exponentially smoothed benefit, cost, and number of *DAO* switches. An approach with higher exponentially smoothed benefit and lower cost is a better approach. An approach with less *DAO* switches is more stable. Stability is desirable, so long as it does not hurt the exponentially smoothed benefit and cost.

Table 7. The number of switches, mean exponentially smoothed benefit and mean cost for Reactive and Procative informed-selection approach for short-term forecasts (i.e. $t + 1$). The best cases are highlighted in gray. *Note that StrictEC case: low energy consumption is favoured over response time and network usage; StrictRTNU case: low response time and low network usage are favoured over energy consumption; StrictALL case: the user constrained the application to favor the three QoS in a strictly manner.*

Cases		Reactive Approach			Proactive Approach		
		# of Switches	Mean Exponential Smoothed benefit	Mean Cost	# of Switches	Mean Exponential Smoothed benefit	Mean Cost
Normal	Best	4	0.7871	0.5184	2	0.7876	0.5140
	Worst	5	0.7475	0.5218	3	0.7481	0.5174
StrictEC	Best	2	0.4367	0.5084	2	0.4367	0.5084
	Worst	0	0.3801	0.5585	1	0.3511	0.5265
StrictRTNU	Best	2	0.4862	0.5084	2	0.4862	0.5082
	Worst	3	0.4502	0.5133	3	0.4502	0.5133
StrictALL	Best	3	0.6274	0.5036	4	0.7173	0.5212
	Worst	4	0.5879	0.5070	5	0.6778	0.5246

Analysis: Table 7 summarizes the mean exponentially smoothed benefit, mean cost and number of switches for all the scenarios. In this experiment, the forecast is performed for the next timestep (i.e. $t + 1$). Our proactive approach showed promising results for some of the cases, as follows (Table 7):

- For the Normal scenario, the average exponentially smoothed benefit and cost by the proactive approach is similar/slightly better than the reactive approach, but with less switches (e.g. for best case two switches instead of four switches), which enhances the iTransport application's stability.
- In the StrictEC and StrictRTNU cases, the reactive and proactive approaches provided the same outcome. This may be due to the following: (i) The energy consumption constraint is too strict in StrictEC case, which was violated most of the times and hence the forecasts did not provide extra benefits; (ii) For StrictRTNU case, knowledge for further ahead timesteps is necessary to provide better results.
- For the StrictALL case, it provides high benefit and high cost, with a slight increase in number of switches, as compared with reactive approach (e.g. for best case provided four switches instead of three). This may be due to the fact that the approach has slightly favored the benefit over the cost.

To this regard, using the proactive approach is recommended over the reactive approach, as it provides better/similar outcome (i.e. based on the context).

Overall Observations: *The proactive approach provided similar or better results than the reactive one. When it was better, it either improved the system stability while maintaining similar exponentially smoothed benefit and cost, or considerably improved the exponentially smoothed benefit through a higher number of switches.*

RQ2.2: *What is the performance of the proactive approach against state-of-the-art architecture evaluation approaches?*

Motivation: The previous section only compared the proactive approach against the reactive one, and when using $h = 1$. RQ1.4 also showed that the error of forecasts using $t + 15$ was similar to that obtained using $t + 1$. However, this does not mean that the proactive approach itself would benefit more from using $t + 15$. RQ2.2 is an extension to these experiments, where we evaluate

how the proactive approach works under varying forecast timesteps ($t+1$, $t+5$, $t+8$, and $t+15$), as compared with state-of-the-art architecture evaluation techniques. We also aim to evaluate how averaging the forecast values for $t+h$, where $h \in 1, 5, 8, 15$ will affect the selection of *dao*. This experiment can also illustrate how to choose a suitable h for use with the proposed approach. Architects willing to use the approach could investigate it with different values of h during an initial period of time, and then decide which h value to carry on using, based on experiments and analyses similar to those performed in this section.

Experimental Design: In this experiment, we compare the proactive system against the reactive approach and the following baseline architecture evaluation systems:

(1) Static-Selection System: It is a typical type of system used in practice [100, 104], where the expert implements a single *dao* based on its assessed value at design-time. Further illustration of how this system works is available in Appendix B.

(2) Predefined-Selection System: This is inspired by [57, 81], where the architect will choose the *dao* that is likely to perform the best for different possible contexts. This selection is typically based on experience, backed up by back-of-the-envelope calculations for the cost, benefits, and technical potential. However, the selection may fail to predict potential fluctuations in value, quality potentials, and costs. Our case used dao_6 during week days (because of peak hours) and dao_3 during weekends (because of less demand).

(3) Random-Selection System: Our design for the baseline system follows the argument of [85, 108] but for the context of services. When a significant change is detected, it selects a *dao* randomly independent of its QoS over time. This is because the *DAO* are deemed to be functionally equivalent but deployed in different environments and geographical location (i.e., distributed Fogs/clouds). All the results related to the Random-selection approach are based on the average of 30 runs (the choice of 30 runs is recommended by [8]).

(4) Reactive Informed-Selection system: It evaluates, detects the change, and selects the optimal *dao* based on *actual* rather than the forecast quality values. This approach is introduced in Section 2.1.

In this experiment, we aim to demonstrate when the proactive approach will/not work well and how does the number of further ahead h forecasts affect the selection. Same applies to the average forecasts, where we will compute the mean of forecasts along with their mean error. We will then evaluate the impact of average forecasts on the decision-making.

We have constructed two cases: *best* and *worst* cases, similar to RQ2.1. For all cases, we have experimented using the error threshold shown in Table 4. We have summarized the results of four cases in Table 9 and 10. We have also conducted Friedman test [45] to check whether there is a significant statistical difference between all the approaches. We have implemented Friedman tests by including the exponentially smoothed benefit for all the cases (i.e. Normal, StrictEC, *etc*, as well as best and worst cases). If there is a significant statistical difference between groups (i.e. p-value < 0.05), we will run Bonferroni-Dunn test [45] between each approach and the top-ranked approach. In addition, to provide a detailed understanding of the behaviour of the proposed approach, we use representative examples for the evaluation as follows: Figure 8 to illustrate the differences between the five approaches for StrictRTNU case for best scenario. Figures showing other cases were omitted. The behaviour of the proposed approach in those cases can be explained in a similar way to the cases illustrated in Figure 9.

Analysis – Overall Results Across Cases:

We first report the results of the Friedman test (Table 8). We have found that there is a significant statistical difference between the approaches in terms of exponentially smoothed benefit (i.e. p-value = $1.49 \times 10^{-8} < 0.05$). The proactive for $t+5$ was the one that obtained the top Friedman rankings. By employing the Bonferroni-Dunn test [45] between each approach and proactive for

Table 8. The average rank of approaches and statistical difference between them measured using p-value with respect to the exponentially smoothed benefit. *The top ranked approach is highlighted in light gray.*

Approach	Average Rank with respect to exponentially smoothed benefit
Static-Selection	2.250
Predefined-Selection	1.750
Random-Selection	2.000
Reactive Informed-Selection	3.875
Proactive Informed-Selection for $t + 1$	4.375
Proactive Informed-Selection for $t + 5$	5.875
Proactive Informed-Selection for $t + 8$	5.000
Proactive Informed-Selection for $t + 15$	5.250
Proactive Informed-Selection for average forecasts	5.125
p-value	$\mathbf{1.49 \times 10^{-8}}$

Table 9. The number of switches, mean exponentially smoothed benefit and mean cost for Static-, Predefined-, and Random-Selection approaches. *Note that StrictEC case: low energy consumption is favoured over response time and network usage; StrictRTNU case: low response time and low network usage are favoured over energy consumption; StrictALL case: the user constrained the application to favor the three QoS in a strictly manner.*

Cases \ Approach		**Static-Selection**			**Predefined-Selection**			**Random-Selection**		
		# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost
Normal	Best	0	0.7057	0.5131	8	0.5268	0.5439	7	0.5966	0.5222
	Worst	0	0	0.5585	8	0.5268	0.5439	7	0.5491	0.5231
StrictEC	Best	0	0.4055	0.5131	8	0.3098	0.5439	4	0.3434	0.5111
	Worst	0	0	0.5841	8	0.3094	0.5458	4	0.2914	0.5120
StrictRTNU	Best	0	0.4907	0.5841	8	0.4161	0.5439	1	0.4291	0.5510
	Worst	0	0.3801	0.5585	8	0.4161	0.5439	0	0.3801	0.5585
StrictALL	Best	0	0.6924	0.5148	8	0.5168	0.5439	7	0.5447	0.5240
	Worst	0	0	0.5585	8	0.5168	0.5439	7	0.4924	0.5296

$t + 5$, we found that there is a significant difference between proactive for $t + 5$ (top ranked) and three approaches (static-selection, predefined-selection, and random-selection). However, there is

Table 10. The number of switches, mean exponentially smoothed benefit and mean cost for Reactive and Proactive informed-selection approach for $t + h$. The best cases are highlighted ingray. *Note that StrictEC case: low energy consumption is favoured over response time and network usage; StrictRTNU case: low response time and low network usage are favoured over energy consumption; StrictALL case: the user constrained the application to favor the three QoS in a strictly manner.*

Cases \ Approach		Reactive			Proactive t+1			Proactive t+5			Proactive t+8			Proactive t+15			Proactive Average Forecasts		
		# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost
Normal	Best	4	0.7871	0.5184	2	0.7876	0.5140	3	0.7885	0.5170	3	0.7885	0.5170	3	0.7885	0.5170	3	0.7885	0.5178
	Worst	5	0.7475	0.5218	3	0.7481	0.5174	4	0.7490	0.5212	4	0.7490	0.5212	4	0.7490	0.5212	4	0.7490	0.5212
StrictEC	Best	2	0.4367	0.5087	2	0.4367	0.5087	2	0.4367	0.5087	2	0.4367	0.5087	2	0.4367	0.5087	2	0.4367	0.5087
	Worst	0	0.3801	0.5585	1	0.3511	0.5265	2	0.4015	0.5652	1	0.3531	0.5251	3	0.3925	0.5679	0	0.3801	0.5585
StrictRTNU	Best	2	0.4862	0.5084	2	0.4862	0.5082	1	0.6762	0.5279	1	0.6762	0.5279	1	0.6762	0.5279	1	0.6762	0.5279
	Worst	3	0.4495	0.5119	3	0.4495	0.5119	2	0.6395	0.5316	2	0.6395	0.5316	2	0.6395	0.5316	2	0.6395	0.5316
StrictALL	Best	3	0.6274	0.5036	4	0.7173	0.5212	3	0.7192	0.5166	4	0.7163	0.5218	4	0.7163	0.5218	2	0.7086	0.5195
	Worst	4	0.5879	0.5070	5	0.6778	0.5246	4	0.6796	0.5200	5	0.6767	0.5252	5	0.6767	0.5252	3	0.6691	0.5229

no significant difference between the proactive for $t + 5$ (top ranked) and reactive, proactive for $t + 1$, $t + 8$, $t + 15$ and average forecasts.

When conducting statistical tests *across cases*, a lack of significant difference between two approaches can happen due to three possible reasons:

1. The two approaches achieved similar results for all cases.
2. One approach performed better for some of the cases, and the other approach performed better for the remaining cases, leading to no significant difference across cases.
3. The two approaches performed similar in several cases, and one approach performed better for the remaining cases. As a result, the test is unable to conclude that the latter approach performed in general better than the former approach across cases.

When analyzing Table 10, we can see that the reason for the lack of significant difference between proactive for $t+5$ and reactive is reason #3. For instance, the reactive and proactive for $t+5$ produced similar exponentially smoothed benefit for normal and StrictEC cases. On the contrary, the proactive for $t + 5$ obtained considerable improvements in mean exponentially smoothed benefit than the reactive approach in Cases 2 and 3.

It is thus clear that the proactive approach with $t + 5$ obtains in general better exponentially smoothed benefit than the static-selection, predefined-selection, and random-selection approaches, and similar or better exponentially smoothed benefit than the reactive approach. Therefore, the proactive approach with $t + 5$ is recommended over other approaches.

As for the case of also having no statistical difference between top-ranked proactive approach and other proactive approaches across cases, this may have occurred because these approaches provided similar exponentially smoothed benefit across the cases (reason #1 above). Specifically, we can see from Table 10 that the average exponentially smoothed benefit for all the proactive approaches was very similar. However, there is a difference between the proactive approaches in terms of the number of switches. In particular, average forecasts provided 1–2 switches less than other proactive approaches in StrictALL (best and worst), and 1-3 switches less for StrictEC (worst).

Therefore, the proactive approach with average forecasts is recommended over the other proactive approaches.

Over the rest of this section, the behaviour of the proposed approach compared to other approaches is discussed in more detail. A discussion on the specific behaviour of the static, predefined and random approaches can be found in Appendix D, and further illustrates why these approaches do not perform well.

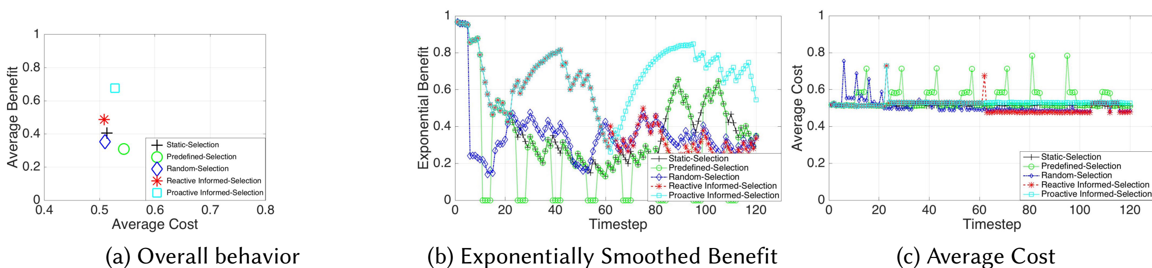
Analysis – Summary of Results With Respect to Each Case Separately:

- *Normal Case (Best/Worst Scenario)*: The results for reactive and all proactive approaches were similar in terms of exponentially smoothed benefit and cost. In particular, Table 10 shows that the reactive and proactive approaches provided mean exponentially smoothed benefit of around 0.79 and cost of around 0.51 or 0.52 for the best case, and mean exponentially smoothed benefit of around 0.75 and cost of around 0.52 for the worst case. However, the proactive approaches led to less switches than the reactive approach. Further, the proactive approach produced 30-50% improvement in mean exponentially smoothed benefit, 5% smaller mean cost, with improved stability (i.e. 3-4 switches less) compared to predefined and random approaches on the best case (Tables 9 and 10). It also provided a 10% increase in mean benefit and similar cost, but with an increase in the number of switches (i.e. 2-3 switches) compared to the static-selection, which adopts a fixed diversified architecture option over time. Similar results were obtained for the worst case scenario, except when comparing against the static-selection approach. This approach obtained a very poor exponentially smoothed benefit of zero (due to its violations to the constraints), and a 7% higher cost than the proactive approaches.
- *StrictEC Case (Best/Worst Scenario)*: The reactive and proactive approaches obtained the same results in the best scenario (same number of switches, mean exponentially smoothed benefit and cost, as shown in Table 10). The proactive approach showed better mean exponentially smoothed benefit (i.e. 10-30%) and cost (i.e. 3-7%) than baseline and state-of-the-art approaches with better stability, except that the static-selection did not perform any switches (Table 9 and 10). In the worst scenario, the reactive and proactive for average forecasts obtained similar results, whereas the other proactive approaches led to varied results (either slightly worse exponentially smoothed benefit and better cost or slightly better exponentially smoothed benefit and worst cost) with larger number of switches. This may be due to the very strict quality constraint, which has been violated in most of the times. This caused a high fluctuation in the exponentially smoothed benefit, which ended up triggering additional switches and producing varying behavior between different $t + h$ values. Therefore, among the proactive approaches, we recommend the one with average forecasts as it considers the forecasts for varying h values. We have seen similar behavior in the worst case scenario, except that static-selection showed the worst benefit and cost (Table 9 and 10). In particular, the proactive approach for average forecasts provided better benefit and cost with same number of switches as static-selection.
- *StrictRTNU Case (Best/Worst Scenario)*: The proactive approaches for long-term forecasts (i.e. $h \in \{5, 8, 15\}$) and average forecasts produced the best results in the best case, in comparison to the reactive and state-of-the-art approaches. In particular, they obtained an increase of about 40% in the average exponentially smoothed benefit, a reduction of about 5% in cost, and less number of switches (i.e. one instead of two), compared to the reactive approach (Table 10). They also obtained a rise of 40-60% in the mean exponentially smoothed benefit with 5-10% reduction in cost and less number of switches, in comparison to state-of-the-art and baseline approaches (Table 9 and 10). Similar behavior occurred for the worst case.

- *StrictALL Case (Best/Worst Scenario)*: The proactive approaches achieved considerably better exponentially smoothed benefit at a slightly higher cost than the reactive approach and state-of-the-art approaches (approximately 5-40% increase in benefit) for the best scenario (Table 9 and 10). The proactive approaches using specific h values led to less switches than the predefined and random approaches, but did not lead to an advantage in terms of number of switches compared to the reactive approach. However, the proactive approach with average forecasts led to the best behaviour in terms of number of switches, except in comparison to the static approach, which uses a fixed *dao* throughout time. Therefore, the proactive approach using average forecasts depicted the best overall results for the best case. A similar behavior was obtained for the worst scenario.

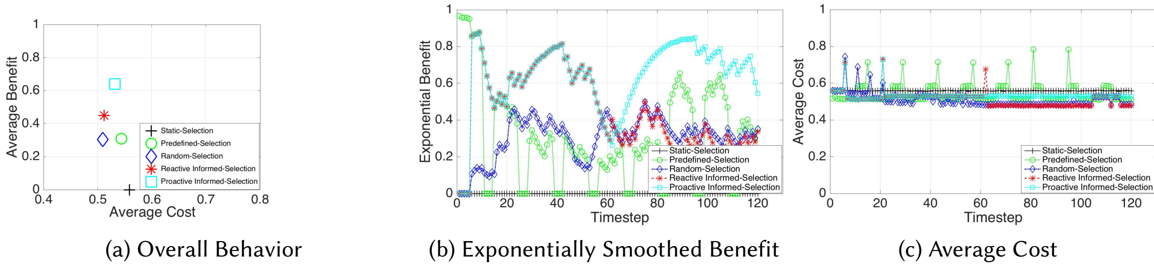
Analysis – Key Representative Examples For All The Approaches Behaviour For StrictRTNU Case: Next, we will demonstrate key representative examples for the diversified architecture options' evaluation by all approaches and its corresponding behavior for StrictRTNU case. The underlying behaviour of the approaches for other cases is similar, and therefore were omitted. Further discussion related to StrictRTNU case is found in Appendix D.2.

The different exponential smoothing benefits and costs obtained by different approaches result from the different choices of *dao* performed by these approaches over time. For StrictRTNU case, the proactive approach with $t + 1$ always led to the *same dao* choices as the reactive approach, whereas the proactive approach with $t + 5$, $t + 8$ and $t + 15$ led to the *same dao* choices as the proactive approach with average forecasts. Therefore, we only plotted the exponential benefit and cost for average forecasts in Figure 8. Interpreting the results and their significance are context dependent and should be discussed with caution. They are meant to serve as the basis for sensitivity analysis of the exponential smoothing benefits over time. For this example, the exponential smoothing benefits were tracked over 120 timestamps (i.e. short-term). The accumulated benefits of 0.05 (as an example) over longer timestamps can be significant (or not) over longer period of time for some context, if the trend would be maintained. Fluctuations are undesirable because they mean that the qualities of interest are fluctuating over time, potentially oscillating between values that may (not) be acceptable in practice and leading to instability. In other cases, fluctuations are not necessarily bad if they respond to a change in the environment. However, in practice we should try to reduce as much as possible any drop in qualities of interest caused by changes in the environment. A service that provides very high quality in one moment and very poor quality in another moment would be deemed unreliable. Therefore, we showed how the approach has successfully chosen the most balanced architectural options (Table 10 and 11) throughout time in the face of uncertainty.



(a) Overall behavior (b) Exponentially Smoothed Benefit (c) Average Cost

Fig. 8. The evaluation of the five approaches' decision-making process for **StrictRTNU** (Average Forecasts) for *best* scenarios. *Note that StrictRTNU: low response time and low network usage are favoured over energy consumption.*



(a) Overall Behavior (b) Exponentially Smoothed Benefit (c) Average Cost

Fig. 9. The evaluation of the five approaches' decision-making process for **StrictRTNU** (Average Forecasts) for *worst* scenarios. *Note that StrictRTNU: low response time and low network usage are favoured over energy consumption.*

The behaviour of the approaches for the worst case scenario was similar to that obtained for the best case scenario, in terms of their *DAO* evaluation and selection. For example, the predefined-selection used the same predefined design-time decisions, whereas the random-selection detected significant changes in benefit and selected adhoc *DAO* (Figure 9b). The proactive approach and reactive approach initially suffered from low benefit in the first 5 timesteps (Figure 9b), but then they quickly handled that and switched to the same *DAO* as in the best case scenario over time (Figure 8b and 9b). The static-selection adopted a single *dao* over time regardless of run-time changes. In particular, it has recommended an architecture option (dao_2), which seemed to violate the constraints all the time. This explains why it produces zero benefit over 120 timesteps (e.g. Figure 9a).

Overall Observations: *The proactive approach for average forecasts has shown in general similar or better results than state-of-the-art, reactive and proactive approaches with other values for h. For instance, it frequently showed significant improvement in benefit and less number of switches, with a similar/slight increase in cost than reactive and state-of-the-art approaches. This is because the average forecasts have benefited the DAO evaluation in terms of using the mean of forecasts of $t + h$ and hence being more realistic than specific h values. In particular, it forecasts the future changes, selects diversified architecture options which are currently optimal and expected to be optimal as well. This in turn has avoided unnecessary switches (i.e. improved the system's stability).*

RQ2.3: *What is the performance of the proactive approach against state-of-the-art architecture evaluation approaches in other complex scenarios?*

Motivation: In this experiment, we aim to convey how our proactive approach aids the architect in evaluating the *DAO* across other complex scenarios, such as Denial of Service (DoS) attack [48, 69], in addition to the one introduced in RQ2.1 and RQ2.2. DoS denotes the presence of unexpected behavior with an irregular pattern (i.e. randomized DoS that does not follow an exponential distribution) [48]. The most common and popular type of DoS is Volumetric or flooding DoS attacks [5]. This type of attacks causes service degradation by flooding the network with large number of requests or traffic [5] (i.e. data). In the context of IoT environments, there are several reasons for DoS attacks [21]: (1) the highly heterogeneity and mobility of IoT devices; (2) some or parts of IoT systems, could be physically unprotected and/or controlled by diverse parties; (3) it is impossible to request permissions for installations and user interactions, due to the large number of IoT devices. One typical example of DoS attack is "a software update for a popular IP-enabled IoT device that causes the device to use the DNS more frequently (e.g., regularly lookup random domain names to check for network availability) could stress the DNS in individual networks when millions of

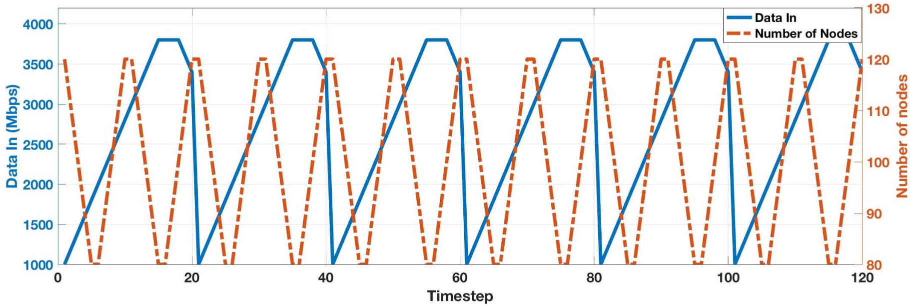


Fig. 10. Sample of changing environmental conditions to simulate DoS attacks facing facing *iTransport* (i.e. input for all of the *DAO*) along with other environmental conditions in Figure 4.

devices automatically install the update at the same time. [39]". To sum up, forecasting DoS attacks and mitigating the risks for their presence in IoT architectures is a non-trivial case.

Experimental Design: To simulate DoS attacks, we varied the data transferred to cameras from 1000 to 3000Mbps as well as the number of nodes as shown in Figure 10 (along other environmental conditions in Figure 4). We then reported the response time, network usage and energy consumption quality concerns over time. For this experiment, we have used the same experimental parameters and environmental conditions as in Experiment RQ2.2 (Figure 11). We examined two scenarios: Best Case and Worst Case as Experiment RQ2.2 and generated the exponentially smoothed benefit with respect to average number of forecasts h (based on Experiment RQ2.2).

Analysis: The variation in the data transferred has caused sudden drops in the QoS of *DAO*. Therefore, our continuous and proactive approach have managed to discover and recover from these failures by using the forecasting ability for change detection and selection of the most balanced *dao* over time. In particular, from Table 11, it is clearly depicted that the proactive approach outperformed the reactive and state-of-the-art approaches in most of the cases.

Overall Observations: *The proactive approach has shown its ability to handle a common challenge in IoT (i.e. DoS attacks), as compared with the state-of-the-art and reactive approaches.*

6 DISCUSSION

6.1 Reflection on the experimental evaluation

In Section 5, we provided the architect with guidelines on: (i) how to determine the input attributes required for forecasting (Table 5 and 6) and the number of training examples to potentially benefit from forecasting (Figure 6a and 6b); (ii) how to select the learners; (iii) how to compare between the further ahead timesteps (Figure 7a and 7b). We also provided a fair comparison of the proactive approach against reactive, state-of-the-art, and baseline approaches (Table 9, 10, and 11 and Figure 8 and 9). Our proactive approach showed better results (Table 7 and 10) than reactive approach in terms of stability (i.e. lower number of switches) and improved exponentially smoothed benefit (Table 11)). Further, it outperformed the baseline, state-of-the-art, and reactive approaches (Table 9 and 10).

From the results in Section 5, we can see that the proactive approach outperformed the state-of-the-art approaches in most of the cases in terms of exponentially smoothed benefit and cost with less switches than the random and predefined approaches. It also provided similar/better benefit than the reactive approach. Even though we cannot generalise the results of each case (normal, StrictEC, StrictRTNU and StrictALL) to other cases in this or other domains, there are

Table 11. The number of switches, mean exponentially smoothed benefit and mean cost for Static-, Predefined-Selection, Random-Selection, Reactive and Proactive approaches for DoS attack case scenario. *The best cases of the proactive approach are highlighted in gray.*

Cases \ Approach		Static-Selection			Predefined-Selection			Random-Selection			Reactive			Proactive		
		# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost	# of switches	Mean exponentially smoothed benefit	Mean Cost
Normal	Best	0	0.1973	0.4593	8	0.1461	0.495	0	0.1973	0.4593	0	0.1973	0.4593	2	0.3121	0.4650
	Worst	0	0.1973	0.4593	8	0.1461	0.495	0	0.1973	0.4593	0	0.1973	0.4593	2	0.3121	0.4650
StrictEC	Best	0	0.2034	0.4577	8	0.1507	0.4950	0	0.2034	0.4577	0	0.2034	0.4577	2	0.2347	0.4602
	Worst	0	0	0.5581	8	0.1507	0.495	3	0.1775	0.4909	1	0.2713	0.4841	2	0.2015	0.4815
StrictRTNU	Best	0	0.173	0.5589	8	0.1461	0.495	0	0.173	0.5589	0	0.173	0.5589	1	0.3836	0.4941
	Worst	0	0.3762	0.5581	8	0.1761	0.495	2	0.3218	0.5351	0	0.3762	0.5581	1	0.4749	0.4841
StrictALL	Best	0	0.1226	0.4577	8	0.0909	0.495	0	0.1226	0.4577	0	0.1226	0.4577	1	0.4163	0.4629
	Worst	0	0	0.5581	8	0.0909	0.495	2	0.0693	0.4894	1	0.3962	0.4841	1	0.3962	0.4841

some observations on the behaviour of the approach that can explain the situations in which one approach may be expected to perform similarly or better than the other approaches.

For instance, when there is a lot of variability in exponentially smoothed benefit over time, the proactive approach with average forecasts may recommend less switches than the reactive approach or the proactive approaches with single values for h. This is because this approach may be able to forecast that, despite a given *dao* being potentially poor at present, it will become better again in the near future, meaning that no switch is required.

Furthermore, when the constraint for one of the quality attributes (e.g. energy consumption) is too strict, the proactive approach provided similar behaviour to the reactive approach. This is because the constraint was violated at some time intervals and the variability in the quality was low, and so we did not benefit from the forecasts.

Additionally, when there are trends in the changes in exponentially smoothed benefit, the proactive approach may achieve better exponentially smoothed benefit and cost trade-offs than the reactive and state-of-the-art approaches, since it has the ability to detect the changes that will happen in the future and hence may recommend better architecture options.

Thrashing was a problem in the reactive approach as the chances for reacting with configuring a *dao* that can get the system into thrashing or halt can be probable. However, the proactive approach goes beyond the reactive approach by profiling these *DAO*, where the knowledge can help in informing when, whether and likely performance of switching to ensure seamless and smooth dynamic configuration and to significantly reduce the likelihood of trashing.

6.2 Approach Assumptions

Our proposed method is fundamentally consistent with ATAM and CBAM practices in capturing human and stakeholders input through various types of scenarios. We are only concerned with requirements and qualities that have impact on the architecture (often referred as architecture significant requirements); our techniques can work on any quality of interest with architecture

relevance, where this quality has been observed over time and data associated with its observation is available and converted into numerical format for our use.

Additionally, architecture design decisions could be *static* or *dynamic* in nature. Several structural design decisions are static in nature; this implies that these decisions can be expensive to change and cannot be altered very frequently at run-time. Henceforth, the architect should evaluate them cautiously at design-time. Example of these decisions include network-related decisions such as the physical connectivity between devices (e.g. how data bits are moving in/out of the IoT device), logical connectivity (e.g. what protocols the software uses to transport these bits, such as MQTT), and also the network topology. These decisions are affected by the expected incoming data volumes, cost, memory requirements, *etc*. Therefore, they are quite difficult to change.

However, there are other decisions, which are dynamic in nature and could be customized at run-time (e.g., predefined decisions that could be tailored to fit the run-time context; strategies and tactics to address behavioural requirements). For instance, different deployment strategies, such as the use of cloud, fog-cloud, *etc*, are an example of a decision that can be best evaluated dynamically. When deemed to be necessary, diversification was also employed to provide "malleability" to alter the structure through inclusion of limited number of tactics that can better meet the behavioural requirements. In this context, our work is particularly interested in investigating and evaluating dynamic design decisions.

6.3 Beneficiaries of approach

Who can benefit from the proposed approach? Architects from several domains could benefit from our continuous evaluation approach. For instance, architects using DevOps are currently deploying some industrial monitoring tools (e.g. AppDynamics [2] and New Relic [3]). Our approach could be integrated to one of these tools, to aid the architect in evaluating, refining and/or phasing-out of architecture design decisions. The forecasting ability could reduce the number of unnecessary adaptations causing a decrease in development and implementation costs. Other applications operating in dynamic environments, such as cloud-based architectures and service oriented architectures could employ the approach to assess abstract architecture model and its possible concrete instantiations over different releases.

The evaluation of self-adaptive systems introduces new challenges that has to deal with evaluating uncertainties and dynamism, making design-time evaluation approaches limited or unfit. Classical approaches for designing adaptive systems tend to take design-time reasoning and evaluation, when deciding on encoding switches, policies, adaptive decisions, etc. Continuous architecture evaluation approach can benefit self-adaptive evaluation: (i) assisting in the systematic inception, elaboration, refinement and evaluation of the adaptive design decisions and/or models (i.e. configurations) that can influence QoS and justifying their need before the system is deployed in the next release cycle; (ii) valuing, profiling, and forecasting the likely performance of these decisions and/or models in meeting qualities in relation to alternatives over time as the software evolves; (iii) continuously evaluating their cost-effectiveness and what to be encoded from these decisions and/or models; and (iv) determine the frequency of adaptation, which have a significant impact on the architecture's stability and hence decide on which options could better improve the architecture's stability for run-time deployment.

7 THREATS TO VALIDITY

In this section, we aim to discuss the threats to construct, internal, and external validity.

7.1 Threats to Construct Validity

These are related to used metrics, which reflect what we intend to measure [92]. For that, we have adopted the Mean Square Error (MAE) and Root Mean Square Error (RMSE) as forecasting performance measures. These measures are unbiased towards under or over estimations, which make them adequate for determining the appropriate number of input attributes; selecting the suitable number of training examples to start forecasting; comparing between the forecasting algorithms; and choosing the suitable number of future ahead timestep. Friedman, post-hoc tests, and Bonferroni-Dunn rank-based tests [45] were adopted to demonstrate the significance of the statistical differences between groups (i.e. forecasting algorithms and further ahead timesteps) in MAE and RMSE.

7.2 Threats to Internal Validity

These are concerned with the impact of experimental parameters (e.g. the relative importance of present/past, the confidence interval, *etc*) on the proposed approach. These experimental parameters are treated as in our previous work [101], where we handled the threats related to internal validity as follows: we analyzed our reactive approach by varying the evaluation parameters and hence selected the ones which provide acceptable accuracy and stability. Note that the same experimental parameters are used by the proactive approach.

7.3 Threats to External Validity

These are linked to the generalization of the experiments [92]. Though we have tested our approach across state-of-the-art, baseline and reactive approaches for architecture evaluation and on a dynamic environment, such as IoT, we cannot claim generality of the results to other data sets and domains. In particular, the number of input attributes and examples found in RQ1.1 and 1.2 are not to be adopted as default values for other use cases. Rather, the procedure followed in to reach those values should be adopted.

Evaluation that is based on simulation can still be considered as design-time if the evaluation is performed at design-time and before deployment. However, we also see potentials for the same simulated approach to work in parallel with the running system, with info-symbiotic feedback between the simulator and the running system to perform anticipatory evaluation of key design decisions and their possible variants based on the run-time context, which may be difficult without the aid of simulation.

It is possible to instantiate our framework by investigating what-if scenarios based on real implementations of the different architecture decisions, and feeding them to the machine learning model. However, in most cases, this would be very expensive in practice. Even if the evaluation is conducted at the deployment stage, it is expected that architecture evaluation is a stage that dominates the inception and elaboration stages of the evolved system and pre-requisite for a detailed low design and implementation of the system to-be. In other words, a software system is expected to evolve over time, and evaluation should typically be done as a prior step to the implementation of such evolved software to prevent high costs. Our framework is in line with this.

Our framework goes beyond classical use cases to leverage simulators to derive a rich and substantial amount of data that can assist the proactive evaluation, supported by machine learning. In particular, the generated dataset can consider eventualities that relate to the norm, extreme, classical or predicted uses. Additionally, several datasets can be compiled, as a result that may relate to different modes of usage of the system. In fact, our work is forward looking in the way it uses data for machine learning. It follows the ethos of what is now referred to as a "Digital Twin" to generate data for predictive and proactive analysis, but linked to the architecture under the evaluation (i.e.,

the architecture is analogous to the "twining" under test) and as an artifact to test the system before being implemented. The simulation environments are approximations to real environments and variation is expected. Variation can sometimes provide the analysts with insights into situations that can rarely be encountered in real settings, providing room for what-if analysis. Nevertheless, our experimentation has investigated the robustness of our reactive approach to noise [101], to mimic variations (low/mid/high noise levels) and fluctuations in real settings and to check how well the approach can handle this issue. Our results show that our reactive approach has managed to select optimal options, even when introducing higher noise levels. It also managed to self-repair from suboptimal choices (due to varying noise levels). These observations can be attributed to the fact that our reactive approach makes use of reinforcement learning and an exponential smoothing function for tracking and factoring in the most recent performing options over time.

Additionally and as explained in Section 3.3, a change detection method was also used to detect whether the time-decayed benefit is deteriorating significantly. Our proposed proactive approach was superior to the existing reactive approach in most cases, demonstrating a good behaviour on this issue. We have also demonstrated the applicability of our approach to monitor the environment at every T' intervals if the real-time evaluation was expensive and the architects did not wish to use simulation. The simulation component of the contribution has also provided flexibility to experiment in ranges of value, i.e., eliminating experimentation bias in fixing the value. Therefore, on the one hand, the use of simulation leads to a threat to external validity in that real data may differ from simulated data, and so the results may not generalise to real data. However, on the other hand, the use of simulations also improves generality in the sense that it enabled the approach to be investigated under different conditions.

Finally, transfer learning methodology [90] has been recently adopted in [67], where the QoS measurements are taken from a simulator and only a few samples are taken from the real system leading to much lower cost and faster learning. In this context, the approach could potentially learn from both simulated and run-time data, which is subject for future work.

8 RELATED WORK

Architecture evaluation typically done as a milestone review that aims at justifying the extent to which the architecture design decisions meet the quality requirements and their trade-offs. The evaluation can aid in early identification and mitigation of design risks. The point of the exercise is to avoid poor decisions and thus save integration, testing and evolution costs [97]. Several architecture evaluation approaches exist, which can be divided into design-time, run-time, and continuous approaches.

8.1 Design-time Architecture Evaluation Approaches

The best known systematic design-time architecture evaluation approaches include the work of [19, 73, 86]. These approaches focus on selecting design decisions and patterns that are most appropriate for quality requirements of interest and their trade-offs. These approaches heavily rely on human inputs and expert judgment. Uncertainties and risks, linked to the deployment, are identified through envisioning a set of scenarios (architectural test cases), taking the form of use case, growth, and exploratory scenarios [73, 76]. Hence, the evaluation and its conclusion are highly dependent on the choice of these scenarios. For example, the Architectural Trade-off Analysis Method (ATAM) [73] is a generic design-time architecture evaluation method that uses a structured walkthrough of scenarios to assess the value of, and risks in, architecture design decisions. The Cost-Benefit Analysis Method (CBAM) [86] is an architecture evaluation method that extends the ATAM to provide cost/benefit analysis for a set of proposed architecture design decisions (called "architecture strategies" in the CBAM). Though the CBAM uses cost/benefit information

to value the architecture design decisions and to justify their selection, this method is unable to dynamically profile the added value of architecture decisions, which is essential for applications operating in uncertain environments (such as IoT). Scenario-Based Architecture Re-engineering (SBAR) [17] is another scenario-based architecture evaluation method that uses different techniques to assess the quality attributes of interest: scenarios, simulation, and mathematical modeling [47]. For example, if a quality attribute is concerned with development and design-time properties, such as maintainability and reusability, scenario-based techniques can be best utilized. Though scenario-based analysis can be still used for behavioural and run-time properties, such as performance and fault-tolerance, simulation and/or mathematical models can better provide meaningful insights and can complement scenario-based ones. Architecture Level Prediction of Software Maintenance (ALPSM) [18] is an architecture evaluation method that utilizes probabilities to determine the likelihood of the impact of scenarios at the software architecture level [47]. It focuses on a single quality concern maintainability. In [100], CBAM has been combined with options theory [63] to predict the value of architecture decisions under uncertainty. Scenario-based evaluation approaches can be best described as best-effort, where the evaluators' expertise and the choice of stakeholders and scenarios are factors that can dramatically influence the evaluation. Though these methods can partially assess and predict the value of architecture decisions under uncertainty, they may suffer from subjectivity, bias, and the limits of human attention.

Apart from the scenario-based design-time approaches, Palladio [16] is a modelling based tool that operates on the specification to generate models, skeleton code and various views to support simulation of the software under consideration (i.e., architecture description and modelling tool with analysis capability). It could be used to simulate the dynamic behaviour of architecture options when evaluating them. In contrary, our continuous evaluation approach complements existing classical trade-off architecture evaluation methods and focuses on architecturally significant requirements/qualities.

Other design-time methods perform decision-making using mathematical models (e.g. [52]), linear programming (e.g. [4]), and evolutionary algorithms (e.g. [6]), and goal refinement (e.g. [31]). In particular, these design-time methods use simulation and optimization to assist in the selection of the candidate architectures and decisions. New domains can impose novelty and new challenges to the evaluation [58, 80, 95], which can make it hard for the architect to either experiment nor evaluate architecture decisions at design-time only. Our continuous evaluation can assist in overcoming the limitations of design-time approaches by providing continuous learning as built-in mechanisms for proactive evaluation that intertwines design-time and run-time evaluation.

8.2 Run-time Architecture Evaluation Approaches

By run-time evaluation, we refer to approaches that use run-time and/or simulated data (e.g. QoS) to capture the dynamic behaviour of architecture decisions under uncertainty and use such information to profile and evaluate design choices. We examine some of these approaches in light of our proposed approach. For instance, though the notion of models@run.time [25] has been implicitly exploited as a way to evaluate the behavior of software systems, the effort was not discussed from the architecture evaluation angle. In particular, models@run.time refers to "models of the functional and/or non-functional software behavior are analyzed at run-time, in order to select system configurations that satisfy the requirements" [33]. Models@run.time operates on the assumption that possible run-time configurations have been already evaluated and encoded in the system, where evaluation can be an after-thought through profiling configurations and recommending alternatives.

In the spirit of models@run.time, several approaches which are architecture-specific have been discussed in the context of self-adaptive and managed architectures [43, 103]. Examples of these

approaches include [7, 36, 40, 56, 84], which use formal analysis of their architectural models. For instance, the Rainbow framework [56] uses Markov processes to compute the expected aggregated impact of each strategy on each quality attribute. This also requires high human intervention to determine the effects of strategies with respect to quality attributes (i.e. predefined probabilities) [38]. Epifani et al. [50] proposed a model-driven approach leveraging a Discrete Time Markov Chain approach and Bayesian estimators to provide continuous automatic verification of requirements at run-time and to support failure detection and prediction. These architecture-based approaches rely on analyical models for the analysis.

This led others (e.g. [51, 74, 106]) to leverage machine learning techniques which gather observations of system properties over time. For example, FUSION [51] is an autonomous online fine-tuning approach for adaptation logic, which is specifically tailored to feature modelling. Unlike FUSION, our approach not only detects goal violations, i.e. constraints, but also checks if the current architecture option is getting worse based on forecast values. A reinforcement learning online planning technique was used by Kim et al. [74] to improve a robot's practices with respect to changes in the environment by dynamically discovering appropriate adaptation plans. However, it does not continuously evaluate the cost-effectiveness of architecture decisions over time. The majority of the aforementioned approaches tend to be reactive when simplistic learning, partial or incomplete knowledge is used [77]: *may suggest wrong decisions due to the unexpected future environment changes*; and *recommend unnecessary switches due to the lack of future knowledge about the candidate architecture decisions*.

New methods have been proposed to proactively deal with effective decision-making for systems facing uncertainties (e.g. [82, 83]). Moreno et al. [82] for example recently proposed a proactive latency-aware adaptation approach that constructs most of the Markov Decision Processes offline through stochastic dynamic programming. Their method focuses on optimizing the latency of adaptation action based on forecasts, without considering the cost of architecture decisions and multiple stakeholder QoS concerns.

Self-adaptive and managed architectures building on models@run.time can particularly benefit from our continuous software architecture evaluation method to better handle uncertainty when configuring concrete instances from abstract models. This is due to the following: (i) it exploits time series forecasting algorithms to forecast and evaluate the cost-benefit of candidate architecture options; (ii) it is designed to be flexible in handling various stakeholders concerns (i.e QoS) and constraints; (iii) it also goes beyond existing approaches by using multi-objective optimization for the various evaluation concerns and provide automated support; and (iv) it is a generic architecture evaluation approach to assess architectures operating under uncertainty.

Runtime verification [13] is "the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [79]". Our continuous evaluation approach has been utilising proactive learning and evaluation to assist in dynamic re-configuration of software architecture. We view runtime verification [79] as an additional step to provide assurance and additional confidence for some of the re/configuration decisions, if would be necessary. As runtime verification can be an expensive and would be impractical to use for every single re-/configuration, we see that the proactive approach can help in prioritising decisions that require further verification. In the future, we plan to check the validity of embedding runtime verification tools to our framework. Though the focus of the paper is not on middleware, existing middleware (e.g. [23, 41, 66]) and other verification approaches (e.g. [34]) may benefit from our contribution to address proactivity.

Variability and commonality has been studied in Software Product Lines (SPLs) works. In particular, some works have proposed modelling-based approaches (e.g. [30, 99, 110]) to evaluate

and predict quality attributes in a software product line. Additionally, other works have used evolutionary algorithms, e.g. to explore the configuration space of a software product line feature model to automatically generate test suites [49], to generate at run-time optimum configurations of the dynamic software product line [91], and dynamically chooses the best operators to solve the product selection problem for SPL testing [53]), as it is infeasible to evaluate every single variant. Other works employ machine learning to improve the decision-making. For instance, Temple at al. [105] have used using machine learning algorithms to infer constraints for product lines. Safdar et al. [94] have combined the use of evolutionary algorithms and machine learning algorithms to mine rules constraining configurations of communicating products across product lines. Temple at al. [105] and Safdar et al. [94] used machine learning to classify or cluster the variability points in SPLs. However, the focus of our approach resemble similarities to dynamic software product line, our work is different, as we employ regression algorithms to forecast the fitness values (i.e. quality of service values and how they changed over time) of the architecture options for IoT architectures and its dynamics configurations. Though our work is not aimed at dynamic SPL, existing work on SPL can make use of our analysis and planning techniques to improve their dynamic decision making for selecting variable options in a proactive manner.

8.3 Continuous Architecture Evaluation

There are few research efforts (e.g. [20, 50, 93]) which explicitly mention continuous architecting and assessment, while some others implicitly adopt it (e.g. [14]). As an example, Pooley and Abdullatif [93] define continuous assessment as the continuous production of performance evaluation tests. Despite their work being one of the few approaches that explicitly provides continuous assessment, it lacks run-time monitoring and forecasting of the performance of architecture decisions. In such cases architecture is, at best, a modelling tool, which may (or may not) be applicable in dynamic environments. Bersani et al. [20] propose a continuous architecting approach that aids the designers in easing the static analysis of the architecture, but lacks dynamicity and is domain-specific (i.e. for streaming applications). Further, though DevOps [14] advocates continuous integration as part of the continuous development process, it lacks a systematic mechanism for evaluating the artifacts generated out of such systems. Thus, novel continuous evaluation methods could aid the prior methods by providing extra insights about the behaviour of candidate architectures to improve decision-making. In particular, they can benefit from further analysis in terms of dynamic tracking and forecasting of architecture decisions' performances, and automated management of cost-benefit trade-offs.

To the best of our knowledge, there are no examples of continuous software architecture evaluation approaches that adopt proactive approaches and forecasting analytics in the research literature. For that, we have proposed our continuous architecture evaluation framework. In particular, our work is the first principled method that extends on classical architecture evaluation methods and their systematic procedure to incorporate run-time and dynamic input to assist in the evaluation, which is not currently captured in the current evaluation methods. Additionally, the evaluation context and so the application of evaluate-forecast-detect-select are different: first, our focus goes beyond performance to cover architecture design decisions - the treatment is fundamentally consistent with ATAM/CBAM in being generic and do not explicate one quality of interest. However, unlike ATAM/CBAM, our work is concerned with dynamic evaluation. Second, though the work builds on fundamentals of self-adaptive systems (e.g. MAPE-K), the evaluate-forecast-detect-select is a self-adaptive variant. However, our realisation in implementing the cycle is different and novel, where we have used techniques that were not previously used (e.g., reinforcement learning in the context of software architecture, change detection methods, time series forecasting, *etc*).

9 CONCLUSIONS AND FUTURE WORK

In this paper, we contributed a novel approach for continuous software architecture evaluation, by providing continuous learning as a built-in mechanism for proactive evaluation that complements reactive approaches. In other words, our work is the first to define a proactive architecture evaluation framework of the systems-to-be, supported by simulation, reinforcement learning and time series forecasting. To instantiate the framework, we have shown how the simulator component (i.e. iFogsim for this example due to its fit for the IoT case), when coupled with machine learning can render an effective evaluation method that goes beyond the current practices. The combination of these techniques (simulation, reinforcement learning and time series forecasting) are among the major components of our architecture evaluation framework. As the common practice does not often expect physical implementation as part of the architecture evaluation process, the use of simulation together with our machine learning components goes beyond has this practice to provide more assurance on the results.

We also explored how various forecasting learning techniques can be "orchestrated" at run-time to better support evaluation and to demonstrate the impact of forecasting analytics on continuous evaluation and decision-making. In most of the cases, our *proactive* run-time approach produces promising results, because it learns and forecasts, and hence makes smarter decisions. This work has paved the way for conducting further studies on continuous software architecture evaluation that can leverage our approach. Though the approach was applied to a representative case of web-operated IoT, it has the potential to benefit large scale dynamic and variant-intensive architectures. In particular, our future investigations will look at how our approach can be applied to software architectures of dynamic and uncertain environments such as volunteer services computing, mobile services architectures and microservice architectures. Findings from such investigations can help to further refine our methods. Additionally, we can explore how similar approaches can help in monitoring and forecasting the health of qualities of interest and associated trade-offs that may be difficult to evaluate using classical architecture evaluation methods, e.g., sustainability and scalability of software architectures, among others.

We have found that when the price volatility when realising/implementing the architecture design option is low, the use of exponential decay factor for cost is not necessary [88, 101]. The current approach has focused on providing a continuous measure for the benefit of each *dao* (i.e. exponentially smoothed benefit). We plan to investigate cases where price volatility can be high, where using exponential decay for the cost can be important. Cases where Dao's cost varies in the presence of dynamic scale and fluctuation in demands can be a potential scenario to explore. Findings can help in better understanding the sensitivity of the continuous architecture evaluation decisions to volatility.

We propose to extend the modeling of quality aggregation functions to consider more complex dependencies between architecture decisions as future work. Though our change detection test described in Section 3.3 produced satisfactory results, we plan to extend our approach to explore other change detection tests. Further, we intend to implement the framework in a decentralised manner, which may potentially improve the computational overhead. However, further investigations are necessary to confirm the efficiency of decentralisation mode.

Further, we aim to demonstrate the usefulness of the transfer learning methodology (introduced in Section 7.3 – the use of a simulator along with the running system) on the continuous architecture evaluation approach. For instance, the framework could be adopted to evaluate an application built using AWS IoT and Greengrass suites [11] by profiling the QoS data from the CloudWatch monitoring tool [12] along with simulated data from iFogSim (as an example). This could provide the architect with extra insights on the provision of an application's challenges in a real setting.

Learner	Parameter	Value
kNN	Number of neighbours	3
Perceptron	Learning rate	0.025
	Fading factor	0.9
SGD Multiclass	Learning rate	0.01
FIMTDD	Tie threshold	0.5
	Learning rate	constant
	Tree type	regression tree
FTM	Fading factor	0.9
AddExp	Factor for decreasing the weights of experts	0.5
	Initial expert weight	0.1
	Loss threshold	0.01

Table 12. The parameters for the single and ensemble learners.

In this context, the use of transfer methodology could learn from both simulated and run-time data and hence potentially confirm the accuracy of the forecasting models. Finally, implementing our framework as a decision support tool can accelerate the adoption of our contributions in an industrial setting.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Rajkumar Buyya and Md. Redowan Mahmud for their valuable comments and helpful suggestions with respect to iFogsim tool.

A ADDITIONAL PLOTS AND TABLES

A.1 Learning Parameters

In this section, we aim to tabulate the parameters of learners adopted for experiment (RQ 1.3, 2.1, and 2.2) in Table 12.

B BINOMIAL TREES FOR DESIGN-TIME EVALUATION

In this appendix, we aim to demonstrate how the design-time evaluation (left part of Figure 1) has chosen the *dao* which appear to provide the most balanced cost-benefit trade-offs over time as an exemplar. For that, we rely on our previous work [100], where a systematic binomial analysis method is used. The static-selection approach uses experts' (e.g. architects and other stakeholders) assumptions on the likely utilities of a *dao* over a predefined period of time [100]. In our experiments, the predefined period of time corresponded to 120 timesteps. The expert's opinion is depicted by a utility tree that is provided at design-time, without making use of any run-time information. The design-time approach [100] then uses this utility tree to compute the likely benefit of a *dao* over the predefined period time based on binomial real option analysis (the left part in Figure 1). Therefore, even though this is a design-time evaluation approach, it provides information on the expected run-time benefit of the diversified architecture options, being a meaningful design-time approach to compare against.

Next, we will discuss briefly the steps of design-time evaluation approach through an exemplar (i.e. StrictRTNU case). The other cases in Table 4.4, such as normal constraint, StrictEC, and StrictALL, and as well as worst case scenario are performed using the same procedures described in this appendix: (1) Identifying diversified architecture options and attributes of interest (From Section 4); (2) Eliciting the benefits and costs of the diversified architecture options from stakeholders; (3)

Table 13. Costs, Options, and Net values of diversified architecture options for StrictEC case (The dao_i with the highest net value is highlighted in lightgray).

dao_i	Operating Cost $c^1_{dao_i}(t)$	Switching Cost $c^2_{dao_i}(t)$	Option Value O_{dao_i}	Net Value NV_{dao_i} (i.e. after deducting switching cost)
1	\$1800	\$600	\$3611.61	\$3011.61
2	\$1000	\$500	\$4505.03	\$4005.03
3	\$1200	\$1000	\$4952.04	\$3952.04
4	\$1500	\$700	\$3030.09	\$3030.09
5	\$900	\$300	\$4494.36	\$4194.36
6	\$1100	\$900	\$7774.24	\$6874.24

Evaluating the diversified architecture options using binomial trees (the results of this step are shown in Table 13).

C THE EFFECT OF ENVIRONMENTAL CONDITIONS ON DECISION-MAKING

In this section, we highlight on whether the change detection by our approach (either reactive or proactive) is aligned with the environmental conditions. We analyse this link with respect to normal case scenario, where the best *dao* is initially deployed.

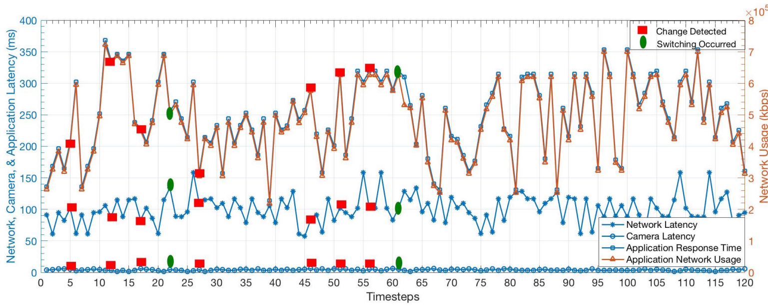
By mapping the environmental conditions to the evaluation of *DAO* based on the informed-selection approach (Figure 11a and 11b), the approach has detected significant deviations which are consistent with input environmental conditions. In particular, the change could be triggered either from high fluctuation and/or a deterioration in one/all of QoS. For instance, from timestep 1 to 5, we see an increase in the values of application response time and network usage, caused by a (smaller) increase in network and camera latency (Figure 11a). These result in a decrease in the exponentially smoothed benefit, which is considered as significant when reaching timestep 5 (Figure 11c). Since the approach is still building knowledge about the current *dao* and having highly dynamic changing conditions during the initial observation period, this caused a change detection at almost every consecutive 5 timesteps until timestep 27 (Figure 11). However, these only led to switches when another better *dao* was available (at timestep 22). This led to improvements in the exponentially smoothed benefit of the proposed informed-selection approach over the following timesteps (Figure 11c). Further, at $t = 46$ (Figure 11a), an increase in network camera latency has caused a noticeable rise in application's response time and network usage, which accordingly resulted in the detection of a significant change in environmental conditions. However, the approach found that the current *dao* still provided the most balanced cost-benefit trade-off and hence the approach kept using this *dao* for the next 15 timesteps (though other change detections were triggered). After that, at $t = 61$, the approach detected a significant change (i.e. a high power load, network and camera latency) and has then selected another *dao* which provided the most balanced cost-benefit trade-off. After $t = 61$, the exponentially smoothed benefit was just oscillating resulting in no significant changes (i.e. no changes were detected), as reflected by the exponentially smoothed benefit (Figure 11c).

D EXTRA EXPERIMENTAL EVALUATION

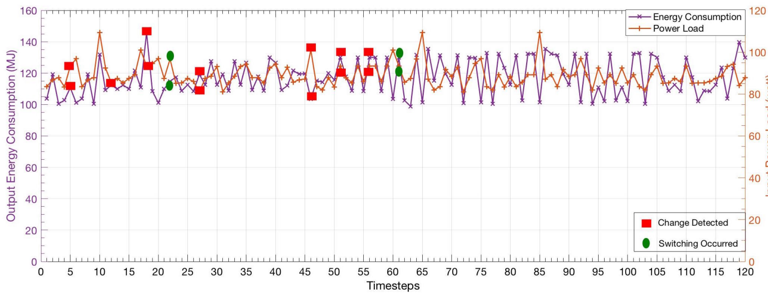
D.1 Further Analysis of RQ2.2 for Normal, StrictEC and StrictALL cases

In this section, we show the analysis of RQ2.2 in detail for Normal, StrictEC and StrictALL cases. We also present the analysis of the behaviour of the static, predefined and random approaches.

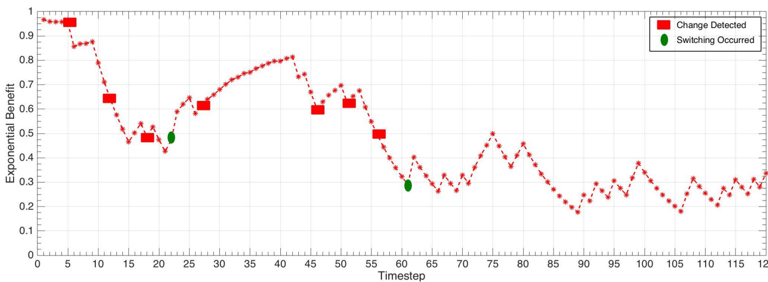
- *Static-Selection:*



(a) The impact of network and camera latency on application's response time and network usage.



(b) The impact of power load on application's energy consumption.



(c) Exponentially smoothed benefit of informed-selection with change detection and switches.

Fig. 11. An illustration of the input environmental conditions for the 120 timesteps including network latency, smart camera's latency, power load on the devices, change detection, and switching occurrence, as well as the output exponentially smoothed benefit of informed-selection approach for strict quality constraints and prioritized ranking score for StrictRTNU (best scenario) as an example. *The red squares represent change detections that did not lead to switching DAO, and green circles represent change detections followed by dao switching.*

– Generally, for the best case scenario, the static-selection for all the cases provided better mean exponentially smoothed benefit with lower cost as compared with predefined-selection and random-selection approaches (Table 9). However, its mean exponentially

smoothed benefit and cost were much worse than reactive and proactive approaches, except for StrictALL (Table 9 and 10).
  - The static-selection approach is ignoring the run-time changes. In this context, it has deployed only one *dao* over time (i.e it has zero switches as seen in Table 9) based on the design-time knowledge (i.e. benefit estimations from the architects). So no matter how this *dao* will behave over time in terms of benefit, the static-selection will keep using it. For example, for the best case scenario, a *dao* has been selected by the architects, which seemed to provide good mean exponentially smoothed benefit (Table 9), as compared to predefined and random approaches. However, for the worst case scenario, the selected *dao* was always violating the QoS constraints over time except for StrictRTNU, resulting in a zero mean exponentially smoothed benefit (Table 9).
- *Predefined selection:* In all the cases (Table 9 and 10), the predefined selection behaved the same. It is providing the lowest mean exponentially smoothed benefit, highest mean cost, and highest number of switches (8 switches). This is because it has predefined *DAO* for deployment, regardless of their benefit at run-time, it will keep using them.
- *Random-Selection:* In most of the cases, the random-selection approach produces low mean exponentially smoothed benefit with high cost and increased number of switches (Table 9) as compared with static-selection, reactive and proactive approaches. Though the random-selection approach detects the changes in *dao*'s benefit, it suggests replacements to random *DAO*. In particular, the adhoc selection in Random-selection approach caused the approach to recommend *DAO* which are not always the best, resulting in poor exponentially smoothed benefit over time. More detailed representative analysis for the behavior of the random-selection approach is shown after the summary of results.
- *Overall:* The static-selection and predefined-selection approaches are biased towards their design-time decisions, which may (not) provide good mean exponentially smoothed benefit and cost (Table 9). Further discussion related to how these approaches operate over time is shown in the representative example (i.e. StrictRTNU) explained after the summary of results. Finally, the predefined- and random-selection approaches showed the worst number of switches, mean exponentially smoothed benefit and cost, as compared with other approaches (Table 9 and 10).

D.2 Detailed Analysis of the Behaviour of the Approaches of RQ2.2 for StrictRTNU case:

Next, we will demonstrate key representative examples for the diversified architecture options' evaluation by all approaches and its corresponding behavior for StrictRTNU case. The underlying behaviour of the approaches for other cases is similar, and therefore were omitted.

From Figure 8, we can see that all approaches have initially observed a gradual decrease in the exponentially smoothed benefit in **StrictRTNU** Case (best) over the first 5 timesteps. This means that the current option (dao_6) was not working well. This may be due to internal problems in devices (e.g. ageing affects) and/or high computation in fixed devices. The approaches then behaved as follows in response to that:

- The static-selection approach never changes the diversified architecture option. Therefore, it kept using the same option (i.e. dao_6) regardless of exponentially smoothed benefit changes at run-time. This led to better mean benefit (i.e. 0.4907) and mean cost (i.e. 0.5841) than random-selection and predefined-selection approaches (Figure 8a), but much worse than reactive and proactive approaches. This is because, despite this decrease in exponentially

smoothed benefit, dao_6 still provided relatively good exponentially smoothed benefit for some time intervals (e.g. from 85th to 95th and 100th to 110th timesteps), as seen in Figure 8b.

- The predefined-selection also ignores run-time changes, but it deploys predefined diversified architecture options from design-time evaluation based on the contextual requirements (dao_3 in weekends and dao_6 in weekdays). The predefined-selection produced very high exponentially smoothed benefit up to 0.8-0.9 in some contexts, where dao_6 was deployed (e.g. Figure 8b). However, due to goal violation in other contexts (i.e. dao_3), it provided zero exponentially smoothed benefit. This describes the high fluctuation leading to low overall exponentially smoothed benefit (e.g. Figure 8a). This poor result is a consequence of the design choices not matching the architects' expectations at runtime.
  In particular, the behavior of predefined-selection was due to the following: (i) The deployed diversified architecture options did not work as expected. In particular, at run-time, it turned out that dao_3 (i.e. one of the predefined options for run-time deployment) had poor exponentially smoothed benefit, which resulted in low mean exponentially smoothed benefit; (ii) Even though dao_3 was not working well, the predefined-selection did not consider the run-time changes (i.e. no change detection test was adopted). Therefore, the predefined approach did not benefit from the design-time knowledge. Instead it applied unnecessary predefined switches between architecture options (i.e. had negative impact on the architecture stability) without noticeable improvement in benefit.
- Random-selection switches to a random diversified architecture option, which ends up producing a drastic drop in exponentially smoothed benefit (from 0.95 to 0.3 as shown in Figure 8b) until t=25, instead of improving the benefit or at least keeping it the same. This has been followed by adhoc switches, which resulted in a significant degradation of benefit over time (an average of 0.4291), and a high mean cost of 0.5510 with very high number of switches (7 switches), as seen in Figure 8a, 8b, and 8c.
- The proactive and reactive approaches have continued adopting dao_6 till $t = 25$. Though there was significant degradation in exponentially smoothed benefit from 0.95 to 0.42 (Figure 8b and 8b), other architecture options did not obtain better balanced cost-benefit than dao_6 until $t = 25$. These approaches were able to detect that and keep using dao_6 until $t = 25$, leading to competitive cost and benefit (Figures 8b and 8c). Afterwards, both approaches detected a change at $t = 27$ and hence suggested a replacement to dao_4, which resulted in a remarkable improvement in benefit from 0.42 to 0.82 (Figure 8b).
- The reactive and proactive approaches provided similar behaviour (i.e. performed same selection of dao) until $t = 61$. This is because the proactive approach will have to wait for 21 timesteps to start using the forecasts ($\eta_{min} + \zeta = 15 + 6 = 21$), as stated in RQ1.2. In this context, the proactive approach used the actual values of quality attributes as the reactive approach. After that, both approaches switched to dao_4 at $t = 27$. Since dao_4 showed a gradual increase in exponentially smoothed benefit until the 55th timestep (Figure 8b), no significant changes were detected by both approaches until this timestep.

The reactive and proactive for short-term and long-term forecasts approaches have observed a significant drop in exponentially smoothed benefit at $t = 61$. We will discuss the approaches' behavior in response to that, as follows (Figure 8):

- Due to the lack of future benefits, the reactive and proactive (i.e. $t + 1$) approaches' evaluation to DAO was based on either actual/short-term forecasts, respectively. Therefore, this resulted in several detection and replacements to inefficient dao (e.g. dao_5), which caused a gradual rise in exponentially smoothed benefit from 0.3 to 0.5 (Figure 8b) with a decrease in cost from 0.5 to 0.45 (Figure 8c) at $t = 71$. As mentioned earlier, the decrease of 0.05 in cost over longer

timestamps can be significant (or not) over longer period of time for some context, if the trend would be maintained. Further, a drop in exponentially smoothed benefit from 0.5 to 0.2 has occurred (Figure 8b) then fluctuation in exponentially smoothed benefit over time. This in turn has affected the mean exponentially smoothed benefit (i.e. 0.5), as seen in Figure 8a.

- The proactive approach using average and long-term forecasts has overcome the previous unnecessary switches. They kept evaluating dao_4 over time with respect to its long-term future benefit. This has led to a reduction in number of replacements (i.e. more stable system) because the change detection is based on long-term future value rather than actual/short-term one.
- Finally, a remarkable rise in exponentially smoothed benefit by almost 175% has occurred at $t = 61$ for proactive approach using average and long-term forecasts, as compared with short-term forecasts and reactive approach (Figure 8b and 8b). This has caused an increase of about 40% for the average exponentially smoothed benefit and less switches with 4% increase in mean cost, in reference to reactive approach (Figure 8a).

The behaviour of the approaches for the worst case scenario was similar to that obtained for the best case scenario, in terms of their *DAO* evaluation and selection. For example, the predefined-selection used the same predefined design-time decisions, whereas the random-selection detected significant changes in benefit and selected adhoc *DAO* (Figure 9b and 9b). The proactive approach and reactive approach initially suffered from low benefit in the first 5 timesteps (Figure 9b and 9b), but then they quickly handled that and switched to the same *DAO* as in the best case scenario over time (Figure 8b,8b, 9b, and 9b). The static-selection adopted a single *dao* over time regardless of run-time changes. In particular, it has recommended an architecture option (dao_2), which seemed to violate the constraints all the time. This explains why it produces zero benefit over 120 timesteps (e.g. Figure 9b and 9a).

REFERENCES

[1] 2016. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are.* Technical Report. Cisco Systems.

[2] 2018. Application Performance Monitoring and Management | AppDynamics.

[3] 2018. New Relic.

[4] Tariq Al-Naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, and Boualem Benatallah. 2005. A quality-driven systematic approach for architecting distributed software applications. In *Proceedings of the 27th international conference on Software engineering*. ACM, 244–253.

[5] P Alcoy, S Bjarnason, P Bowen, CF Chui, K Kasavchenko, and G Sockrider. 2018. 13th annual worldwide infrastructure security report. *NETSCOUT Arbor* (2018).

[6] Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. 2009. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*. IEEE, 61–71.

[7] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. 2013. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering* 39, 5 (2013), 658–683.

[8] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 1–10.

[9] Algirdas Avizienis and John PJ Kelly. 1984. Fault tolerance by design diversity: Concepts and experiments. *Computer* 8 (1984), 67–80.

[10] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on* 1, 1 (2004), 11–33.

[11] Aws. 2021. AWS IoT Core pricing. https://aws.amazon.com/iot-core/pricing/.

[12] Amazon aws. 2018. Cloud Watch.

[13] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to runtime verification. In *Lectures on Runtime Verification*. Springer, 1–33.

[14] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[15] Benoit Baudry, Martin Monperrus, Cendrine Mony, Franck Chauvel, Franck Fleurey, and Steven Clarke. 2014. DIVERSIFY: Ecology-inspired software evolution for diversity emergence. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 395–398.
[16] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 1 (2009), 3–22.
[17] PerOlof Bengtsson and Jan Bosch. 1998. Scenario-based software architecture reengineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE, 308–317.
[18] P Bengtsson and Jan Bosch. 1999. Architecture level prediction of software maintenance. In *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*. IEEE, 139–147.
[19] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. 2004. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software* 69, 1 (2004), 129–147.
[20] Marcello M Bersani, Francesco Marconi, Damian A Tamburri, Pooyan Jamshidi, and Andrea Nodari. 2016. Continuous architecting of stream-based systems. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, 146–151.
[21] Elisa Bertino and Nayeem Islam. 2017. Botnets and internet of things security. *Computer* 50, 2 (2017), 76–79.
[22] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010. MOA: Massive Online Analysis. *Journal of Machine Learning Research* 11 (2010), 1601–1604. http://portal.acm.org/citation.cfm?id=1859903
[23] Benjamin Billet and Valérie Issarny. 2014. Dioptase: a distributed data streaming middleware for the future web of things. *Journal of Internet Services and Applications* 5, 1 (2014), 13.
[24] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. 2017. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing* 4, 2 (2017), 26–35.
[25] Gordon Blair, Nelly Bencomo, and Robert B France. 2009. Models@ run. time. *Computer* 42, 10 (2009).
[26] Barry W Boehm and Kevin J Sullivan. 2000. Software economics: a roadmap. In *Proceedings of the conference on The future of Software engineering*. ACM, 319–343.
[27] Jan Bosch. 2004. Software architecture: The next step. In *European Workshop on Software Architecture*. Springer, 194–199.
[28] Léon Bottou and Olivier Bousquet. 2008. The tradeoffs of large scale learning. In *Advances in neural information processing systems*. 161–168.
[29] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. 2004. Finding knees in multi-objective optimization. In *International Conference on Parallel Problem Solving from Nature*. Springer, 722–731.
[30] Franz Brosch, Barbora Buhnova, Heiko Koziolek, and Ralf Reussner. 2011. Reliability prediction for fault-tolerant software architectures. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. 75–84.
[31] Saheed A Busari and Emmanuel Letier. 2017. Radar: A lightweight tool for requirements and architecture decision analysis. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 552–562.
[32] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience* 41, 1 (2011), 23–50.
[33] Radu Calinescu. 2013. Emerging techniques for the engineering of self-adaptive high-integrity software. In *Assurances for Self-Adaptive Systems*. Springer, 297–310.
[34] Javier Cámara, David Garlan, and Bradley Schmerl. 2017. Synthesis and quantitative verification of tradeoff spaces for families of software systems. In *European Conference on Software Architecture*. Springer, 3–21.
[35] Humberto Cervantes and Rick Kazman. 2016. *Designing software architectures: a practical approach*. Addison-Wesley Professional.
[36] Franck Chauvel, Nicolas Ferry, Brice Morin, Alessandro Rossini, and Arnor Solberg. 2013. Models@ Runtime to Support the Iterative and Continuous Design of Autonomic Reasoners.. In *MoDELS@ Run. time*. 26–38.
[37] Yuang Chen and Thomas Kunz. 2016. Performance evaluation of IoT protocols under a constrained wireless access network. In *Selected Topics in Mobile and Wireless Networking (MoWNeT), 2016 International Conference on*. IEEE, 1–7.
[38] Shang-Wen Cheng. 2008. *Rainbow: Cost-effective Software Architecture-based Self-adaptation*. Ph.D. Dissertation. Pittsburgh, PA, USA. Advisor(s) Garlan, David. AAI3305807.
[39] Michael Cooney. 2019. Reports: As the IoT grows, so do its threats to DNS. https://www.networkworld.com/article/3411437/reports-as-the-iot-grows-so-do-its-threats-to-dns.html.
[40] Deshan Cooray, Ehsan Kouroshfar, Sam Malek, and Roshanak Roshandel. 2013. Proactive self-adaptation for improving the reliability of mission-critical, embedded, and mobile software. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1714–1735.
[41] Iván Corredor, José F Martínez, Miguel S Familiar, and Lourdes López. 2012. Knowledge-aware and service-oriented middleware for deploying pervasive services. *Journal of Network and Computer Applications* 35, 2 (2012), 562–576.

[42] Li Da Xu, Wu He, and Shancang Li. 2014. Internet of Things in industries: A survey. *Industrial Informatics, IEEE Transactions on* 10, 4 (2014), 2233–2243.

[43] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.

[44] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[45] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research* 7, Jan (2006), 1–30.

[46] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10, 4 (2015), 12–25.

[47] Liliana Dobrica and Eila Niemela. 2002. A survey on software architecture analysis methods. *IEEE Transactions on software Engineering* 28, 7 (2002), 638–653.

[48] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. 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*. ACM, 7–16.

[49] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. 2012. Evolutionary search-based test generation for software product line feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 613–628.

[50] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. 2009. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 111–121.

[51] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. 2013. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE transactions on software engineering* 39, 11 (2013), 1467–1493.

[52] Naeem Esfahani, Sam Malek, and Kaveh Razavi. 2013. GuideArch: guiding the exploration of architectural solution space under uncertainty. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 43–52.

[53] Thiago N Ferreira, Jackson A Prado Lima, Andrei Strickler, Josiel N Kuk, Silvia R Vergilio, and Aurora Pozo. 2017. Hyper-heuristic based product selection for software product line testing. *IEEE Computational Intelligence Magazine* 12, 2 (2017), 34–45.

[54] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. 2004. Learning with drift detection. In *Advances in artificial intelligence–SBIA 2004*. Springer, 286–295.

[55] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 44.

[56] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (2004), 46–54.

[57] Swapna S Gokhale. 2007. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on dependable and secure computing* 4, 1 (2007).

[58] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660.

[59] Tom Guérout, Thierry Monteil, Georges Da Costa, Rodrigo Neves Calheiros, Rajkumar Buyya, and Mihai Alexandru. 2013. Energy-aware simulation with DVFS. *Simulation Modelling Practice and Theory* 39 (2013), 76–91.

[60] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296.

[61] Kevin Gurney. 2014. *An introduction to neural networks*. CRC press.

[62] Matthew J Hawthorne and Dewayne E Perry. 2004. Applying design diversity to aspects of system architectures and deployment configurations to enhance system dependability. In *DSN 2004 Workshop on Architecting Dependable Systems,(DSN-WADS 2004)*.

[63] John C Hull. 2006. *Options, futures, and other derivatives*. Pearson Education India.

[64] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[65] Elena Ikonomovska, João Gama, and Sašo Džeroski. 2011. Learning model trees from evolving data streams. *Data mining and knowledge discovery* 23, 1 (2011), 128–168.

[66] Valérie Issarny, Georgios Bouloukakis, Nikolaos Georgantas, and Benjamin Billet. 2016. Revisiting service-oriented architecture for the IoT: a middleware perspective. In *International Conference on Service-Oriented Computing*. Springer, 3–17.

[67] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer learning for improving model predictions in highly configurable software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 31–41.
[68] Anton Jansen and Jan Bosch. 2005. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. IEEE, 109–120.
[69] Jessica Jones, Jason D Hiser, Jack W Davidson, and Stephanie Forrest. 2019. Defeating denial-of-service attacks in a self-managing N-variant system. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 126–138.
[70] Magne Jorgensen and Martin Shepperd. 2007. A systematic review of software development cost estimation studies. *IEEE Transactions on software engineering* 33, 1 (2007), 33–53.
[71] Rick Kazman, Jai Asundi, and Mark Klein. 2001. Quantifying the costs and benefits of architectural decisions. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 297–306.
[72] Rick Kazman, Serge Haziyev, Andriy Yakuba, and Damian A Tamburri. 2018. Managing Energy Consumption as an Architectural Quality Attribute. *IEEE Software* 35, 5 (2018), 102–107.
[73] Rick Kazman, Mark Klein, and Paul Clements. 2000. *ATAM: Method for architecture evaluation.* Technical Report. DTIC Document.
[74] Dongsun Kim and Sooyong Park. 2009. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 76–85.
[75] Jeremy Z Kolter and Marcus A Maloof. 2005. Using additive expert ensembles to cope with concept drift. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 449–456.
[76] Heiko Koziolek. 2011. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM, 3–12.
[77] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17 (2015), 184–206.
[78] John Langford, Lihong Li, and Tong Zhang. 2009. Sparse online learning via truncated gradient. *Journal of Machine Learning Research* 10, Mar (2009), 777–801.
[79] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.
[80] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. 2018. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything*. Springer, 103–130.
[81] Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. 2011. Architecture-based reliability evaluation under uncertainty. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM, 85–94.
[82] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 147–156.
[83] Gabriel A Moreno, Alessandro Vittorio Papadopoulos, Konstantinos Angelopoulos, Javier Cámara, and Bradley Schmerl. 2017. Comparing model-based predictive approaches to self-adaptation: CobRA and PLA. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*. IEEE, 42–53.
[84] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. 2009. Models@ run. time to support dynamic adaptation. *Computer* 42, 10 (2009).
[85] Klara Nahrstedt, Hongyang Li, Phuong Nguyen, Siting Chang, and Long Vu. 2016. Internet of mobile things: Mobility-driven challenges, designs and implementations. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE, 25–36.
[86] Robert L Nord, Mario R Barbacci, Paul Clements, Rick Kazman, and Mark Klein. 2003. *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the cost benefit analysis method (CBAM).* Technical Report. DTIC Document.
[87] Gustavo HFM Oliveira, Rodolfo C Cavalcante, George G Cabral, Leandro L Minku, and Adriano LI Oliveira. 2017. Time Series Forecasting in the Presence of Concept Drift: A PSO-based Approach. In *Tools with Artificial Intelligence (ICTAI), 2017 IEEE 29th International Conference on*. IEEE, 239–246.
[88] Steffen Opel. 2018. AWS Streamlines Amazon EC2 Spot Instance Pricing Model and Operational Complexity.
[89] Ipek Ozkaya, Rick Kazman, and Mark Klein. 2007. Quality-attribute based economic valuation of architectural patterns. In *Economics of Software and Computation, 2007. ESC'07. First International Workshop on the*. IEEE, 5–5.
[90] Sinno Jialin Pan, Qiang Yang, et al. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

[91] Gustavo G Pascual, Roberto E Lopez-Herrejon, Mónica Pinto, Lidia Fuentes, and Alexander Egyed. 2015. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *Journal of Systems and Software* 103 (2015), 392–411.

[92] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[93] RJ Pooley and AAL Abdullatif. 2010. Cpasa: continuous performance assessment of software architecture. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on.* IEEE, 79–87.

[94] Safdar Aqeel Safdar, Hong Lu, Tao Yue, and Shaukat Ali. 2017. Mining cross product line rules with multi-objective search and machine learning. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 1319–1326.

[95] Pete Sawyer, Animesh Pathak, Nelly Bencomo, and Valérie Issarny. 2012. How the web of things challenges requirements engineering. In *International Conference on Web Engineering*. Springer, 170–175.

[96] Eduardo S Schwartz and Lenos Trigeorgis. 2004. *Real options and investment under uncertainty: classical readings and recent contributions.* MIT press.

[97] Software Engineering Institute (SEI). 2018. *Reduce Risk with Architecture Evaluation.* Technical Report. SEI/CMU.

[98] Shai Shalev-Shwartz and Ambuj Tewari. 2011. Stochastic methods for l1-regularized loss minimization. *Journal of Machine Learning Research* 12, Jun (2011), 1865–1892.

[99] Norbert Siegmund, Marko RosenmüLler, Christian KäStner, Paolo G Giarrusso, Sven Apel, and Sergiy S Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology* 55, 3 (2013), 491–507.

[100] Dalia Sobhy, Rami Bahsoon, Leandro Minku, and Rick Kazman. 2016. Diversifying Software Architecture for Sustainability: A Value-based Perspective. *Proceedings of 2016 the European Conference on Software Architecture (ECSA)* (2016).

[101] Dalia Sobhy, Leandro Minku, Rami Bahsoon, Tao Chen, and Rick Kazman. 2019. Run-time evaluation of architectures: A case study of diversification in iot. *Journal of Systems and Software* (2019), 110428.

[102] Minku L. Sobhy D., Bahsoon R. and Kazman R. 2021. Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review. *TOSEM* In-press (Accepted) (2021).

[103] Michael Szvetits and Uwe Zdun. 2016. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling* 15, 1 (2016), 31–69.

[104] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. 2009. *Software architecture: foundations, theory, and practice.* Wiley Publishing.

[105] Paul Temple, José A Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using machine learning to infer constraints for product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference.* 209–218.

[106] Gerald Tesauro. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing* 11, 1 (2007), 22–30.

[107] Jan Salvador van der Ven, Anton GJ Jansen, Jos AG Nijhuis, and Jan Bosch. 2006. Design decisions: The bridge between rationale and architecture. In *Rationale management in software engineering*. Springer, 329–348.

[108] Florian Wagner, Fuyuki Ishikawa, and Shinichi Honiden. 2016. Robust service compositions with functional and location diversity. *IEEE Transactions on Services Computing* 9, 2 (2016), 277–290.

[109] Shuo Wang, Leandro L Minku, and Xin Yao. 2015. Resampling-Based Ensemble Methods for Online Class Imbalance Learning. *Knowledge and Data Engineering, IEEE Transactions on* 27, 5 (2015), 1356–1368.

[110] Hongyu Zhang, Stan Jarzabek, and Bo Yang. 2003. Quality prediction and assessment for product lines. In *International Conference on Advanced Information Systems Engineering*. Springer, 681–695.

[111] Min-Ling Zhang and Zhi-Hua Zhou. 2007. ML-KNN: A lazy learning approach to multi-label learning. *Pattern recognition* 40, 7 (2007), 2038–2048.

Received September 2018