Assessing Self-Adaptation Strategies Using Cost-Benefit Analysis

Ilias Gerostathopoulos
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
i.g.gerostathopoulos@vu.nl

Claudia Raibulet
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
c.raibulet@vu.nl

Elvin Alberts
Universiteit van Amsterdam
Amsterdam, the Netherlands
elvin.alberts@student.uva.nl

***Abstract*—Self-adaptive software systems stand out from traditional ones in that they are able to autonomously change their behavior and structure during their execution using one or more self-adaptation strategies. The main objective of such a strategy is to maintain or even improve the functionalities/qualities of the system despite uncertainty in its operational environment. To date, a number of self-adaptation strategies have been proposed – following the architectural, control-theoretic, or online search paradigm – for several application domains. However, it is still unclear when a particular self-adaptation strategy needs to be developed and when it needs to be used at runtime (when a system can choose among several available strategies). In this paper, we aim to answer the above questions by relying on the assessment of a strategy's costs (e.g., development effort, performance decrease) and benefits (e.g., re-usability, performance improvement) at design time and runtime. The main novelty is that we provide a holistic view over the return on investment of a strategy and propose that the system itself uses cost-benefit analysis to decide on which strategy to apply at runtime.**

***Index Terms*—Self-adaptive systems, self-adaptation strategies, costs, benefits, cost-benefit analysis, return on investment.**

I. Introduction

Self-adaptive software systems are able to change their behavior and structure at runtime to accommodate changes in their operational environment and maintain or even improve their functionalities and qualities without human intervention. To do so, they are equipped with algorithms and mechanisms, i.e., called here self-adaptation strategies, to perform runtime changes. Examples of strategies are elasticity policies that govern cloud autoscaling [1], architecture adaptation scripts [2], MPC controllers synthesized for an adaptation problem [3], online search algorithms [4]. Strategies need to be developed or at least configured for each system. They also end up inevitably increasing the overall complexity of the system.

The above observations raise the first question we investigate in this paper: *Does the development of a particular self-adaptation strategy for a software system pay off?* To answer this, we formulate the decision of developing a self-adaptation strategy as an architectural design decision. In particular, we focus on the costs and benefits associated with the development of a strategy (at design time) and the costs and benefits associated with its use (at runtime). We associate a *Return on Investment* (ROI) value to a self-adaptation strategy, based on its costs and benefits. ROI is a metric borrowed from finance. It captures the probability to have a return, i.e., benefits, from an investment, i.e., effort or cost.

A second question concerns the execution of a system: *Which of the self-adaptation strategies is the most appropriate (in terms of costs and benefits) to be applied to the current context?* The novel idea we contribute in this paper is that a decision-making mechanism based on costs and benefits can be employed not only at design time (by software architects and business analysts), but also at runtime, by the system itself. In this setting, a self-adaptive system is able to use different strategies and needs to decide which to use (including the case of using no strategy at all) in each of its contexts. We propose to rely on the quantification of the associated costs and benefits to effectively rank and select strategies in different contexts.

Performing cost-benefit analysis for the self-adaptation strategies at runtime is potentially useful not only for selecting an appropriate strategy for each context, but also for informing the evolution of the strategies. Indeed, continuously monitoring the runtime costs and benefits of strategies can be used for detecting when a strategy has ceased to be cost-effective. This can happen, since the system and its environment evolve over time and what used to be a cost-effective strategy may not be anymore. Monitoring cost-benefit mechanisms may signal to the architects that a strategy needs modification or even that a new strategy needs to be introduced (e.g., to handle some exceptional cases not anticipated by the current strategy).

In the following, we describe the cost-benefit analysis for self-adaptive systems (Section II) at design (Section II-A) and runtime (Section II-B). We compute the runtime cost-benefit on a self-adaptation exemplar (Section III), compare to related work (Section IV) and provide our conclusions (Section V).

II. Cost vs Benefits in Self-Adaptive Systems

The starting point of designing a new self-adaptation strategy is to identify the need for such a strategy. This relies on the basic assumption that a system is able to compute some kind of health metric – we call this *Overall System Utility* (OSU). One can look at *scenarios* where a system experiences a drop in its OSU (e.g., due to an increase in its users) or a fault that prevents it from functioning properly (e.g., due to a bug).

Once one or more such scenarios are identified, a decision is taken on whether to account for or simply ignore them. The

decision needs to be based on three factors: (1) the severity of the scenario, i.e., the difference between OSU and the reduced one; and (2) the duration of the scenario, (3) the frequency of occurrence within a time frame of reference (e.g., a month). Clearly, severe scenarios that last long and also occur frequently are the hardest to ignore. If there is a decision to address the identified scenario, software architects need to identify potential self-adaptation strategies. *The decision to select which self-adaptation strategy to implement and use is actually an architecture design decision, since it influences how the system will be evolved to accommodate each strategy.*

We propose to employ cost-benefit analysis to aid decision making in this setting. Our approach lies on the observation that each strategy s carries benefits (B) and costs (C) at design (des) and runtime (run). Overall, benefits and costs are used to calculate the return on investment (ROI) of a strategy as:

$$ROI_{all}^{s} = \frac{(B_{des}^{s} + B_{run}^{s}) - (C_{des}^{s} + C_{run}^{s})}{C_{des}^{s} + C_{run}^{s}}$$

A. Design Time

At design time, software architects explore the trade-offs of decisions related to the design, development, maintenance, and evolution of a system. In this section, we look at such decisions for self-adaptation strategies from a cost-benefit analysis perspective and underline the key design aspects that may influence their costs and benefits.

First, strategies should address or affect one or more quality attributes (e.g., performance) of a system. A potential benefit is to provide robust mechanisms to measure such quality attributes at runtime. Such mechanisms may be reused by other strategies or simply allow architects to gain more insights.

Second, strategies should be *independent* software components easily *interchanged* with other strategies. This leads to the benefit of defining customizable strategies based on various approaches (e.g., rule-based, machine learning-based, online search-based) and re-using them in various scenarios/systems. To facilitate such reuse, self-adaptive systems should be designed with *external* feedback loops [5].

Third, strategies have a cost proportional to their development and integration effort, and they also increase the effort to develop, maintain, and evolve the whole system, due to the extra complexity they add.

Fourth, strategies may be *centralized*, i.e., having an effect on the entire system, or *decentralized*, i.e., having an effect on part of the system. A decentralized strategy may need complex synchronization mechanisms at the system level, and this leads to a higher cost for its design and development.

Fifth, strategies may be *reactive*, triggered by an event or change, or *proactive*, triggered by internal mechanisms based on estimations, probabilities, predictions. In the first case, the design of the strategy does not require historical information about the system; in the second case its design requires mechanisms for historical information (e.g., a knowledge base, machine learning algorithm training) to achieve proactivity. This has an impact in terms of costs for the strategy design, e.g., to acquire or collect the necessary data.

In summary, the benefit (B_{des}^{s}) of a strategy at design time concerns the re-usability of measurement mechanisms for quality attributes, and the re-usability of the strategy itself in different scenarios or systems (if developed as independent components). The cost (C_{des}^{s}) is typically associated with the development and integration effort of the strategy itself but also of the overall system, and the cost of acquiring data or models needed for the operation of the strategy.

B. Runtime

When a self-adaptation strategy is deployed to a system, we can start measuring its runtime benefits and costs. Intuitively, a strategy brings a *benefit* at runtime when it increases its OSU compared to the baseline case where no strategy is used. It follows then that a strategy incurs a *cost* when it decreases its OSU compared to the baseline case. This is illustrated in the simplified diagram in Fig. 1. The baseline case (grey line) corresponds to the scenario where the system experiences a large drop in its OSU (e.g., due to a change in user load). With no strategy in place, the low OSU will continue until the end of the scenario, when it will get back to its prior value.

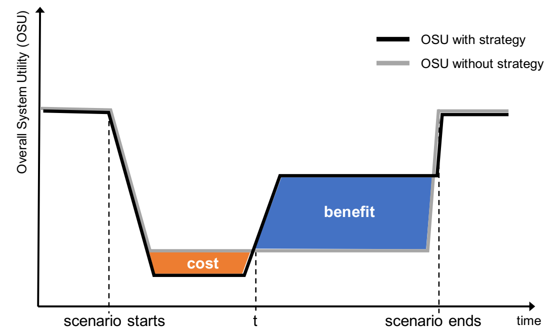

Fig. 1. Illustration of cost and benefit in a scenario.

When a strategy is applied (black line in Fig. 1, it may start with a decrease in the OSU (due to e.g., trying out different adaptation actions that lead to lower OSU), until a point t where it starts increasing the OSU (e.g., when it learns that a particular adaptation action is successful). In this case, the benefit of the strategy is proportional to the area between the two lines in the diagram where OSU is higher than the baseline (blue area), whereas the cost is proportional to the orange area. Clearly, these two areas do not have to be continuous or shaped as in Fig. 1. The above observation gives us a straightforward way to precisely calculate benefits and costs in each scenario. However, the problem is that such a calculation is only possible when a scenario can be replicated – which is not the case with real, non-simulated systems.

As an alternative, we propose to calculate the expected benefit and cost of a strategy s in a scenario class σ as follows: (1) collect OSU traces from ***different scenarios of the same class***, with and without applying the strategy (baseline), (2) split each trace into k ***equally spaced*** segments, (3) calculate the ***average OSU per segment*** for both the strategy s and the baseline b, (4) calculate the ***percentage increase or decrease***

between the strategy and the baseline for each segment, (5) calculate ***benefit*** based on segments with percentage increase, and (6) calculate ***cost*** based on segments with percentage decrease. In particular, given the average OSU for a segment m, $\overline{OSU^s_{\sigma,m}}$ (step 3), the percentage change is computed as:

$$PC^s_{\sigma,m} = \frac{\overline{OSU^s_{\sigma,m}} - \overline{OSU^b_{\sigma,m}}}{\overline{OSU^b_{\sigma,m}}}$$

A segment has a positive change value, a percentage *increase* (PI^s_σ), or a negative one, a percentage *decrease* (PD^s_σ).

To calculate the benefit or cost of a segment m in terms of monetary value we introduce the functions $B^s_{\sigma,m}(PI^s_\sigma, t)$ and $C^s_{\sigma,m}(PD^s_\sigma, t)$, where t is the duration of the segment. These functions have to be empirically constructed and return the amount of money that was gained or lost from the percentage change. The overall expected benefit B^s_σ, resp. cost C^s_σ, of a strategy s in a scenario of class σ is calculated by summing up the amount that was gained, resp. lost, in each segment.

1) Calculating runtime benefit and cost across different scenario classes: The process described above can be used to calculate the expected benefit and cost of a strategy in scenarios belonging to different classes. With such estimates and with the frequency of occurrence of scenarios for each class, we calculate the overall runtime benefit B^s_{run} and cost C^s_{run} of a strategy s over a period of time. For instance, consider a strategy based on a PD controller that has an expected benefit of 200€ for a scenario of a sharp increase in the request rate for a system (class 1) and an expected benefit of 300€ for a scenario of a smooth increase in the request rate (class 2). If 4 smooth increases and 2 sharp increases are expected in a month, the monthly benefit of that strategy would be $4*300+2*200 = 1600$€. Once the overall runtime benefit and cost is known (or can be reliably estimated) for a strategy, it can be used together with its design time counterparts to inform the architecture design decision of using it.

2) Using runtime benefit and cost to switch between strategies at runtime: Knowing the expected benefit and cost of more than one strategy applicable to a scenario of a particular class, a further possibility is to use a simple cost-benefit analysis to automatically decide which strategy to use for a particular scenario at runtime. Such analysis can take the form of simply comparing the expected *net profit*, i.e., benefit minus cost, of two or more strategies and applying the one with the highest net profit for the scenario class at hand. One of the challenges is to be able to accurately determine, *at the start* of a scenario, which class it belongs to – which is necessary prerequisite if a strategy is to be selected based on that class.

III. Illustration on SWIM

In this section, we illustrate our approach of cost-benefit analysis with a self-adaptive exemplar, the Simulator for Web Infrastructure and Management (SWIM) [6]. SWIM simulates a three-tier web application, consisting of client, web, and data tiers. The client tier sends requests to the web tier according to a provided requests trace. The web tier serves client requests using a load balancer that assigns each request to an available server. The time for serving a request by a server (i.e., rendering HTML pages with dynamic content retrieved from the data tier) is simulated using OMNeT++ [7].

To cope with changes in the request load over time while generating profit, SWIM provides 2 self-adaptations: (1) add/remove a web server, (2) change the proportion of responses including optional content (e.g., advertisements) – *dimmer* value. More servers handle more load but also have higher infrastructure cost, while higher dimmer values yield more profit but increase the time to process requests. We capture OSU using a utility function that considers the profit from serving basic and optional content, the server utilization, and the average response time[1]. Compared to the utility function in [3], we modified the case of the response time being above a minimum threshold to include the infrastructure cost (servers in use). To illustrate the calculation of costs and benefits at runtime, we focus on the scenario class characterized by a sharp increase in the number of requests – shown on the top of Fig. 2. Then, we compare the 2 SWIM self-adaptation strategies based on their runtime costs and benefits:

- *Reactive* strategy [6]. Provided by SWIM, this strategy keeps adding servers if the average response time is below a threshold and then increases the dimmer to maximize profit. If no server can be added, the dimmer is decreased. Finally, when an excessive number of servers is detected, it keeps removing servers.
- *ϵ-greedy* strategy [8]. This strategy employs ϵ-greedy, an online learning algorithm to find a good performing configuration (we only experimented with the number of servers). It is configured by an ϵ value that controls how often the algorithm will explore (evaluate a new configuration) versus exploit (use the best configuration it has found so far). In particular, we employed an exponentially decaying ϵ to speed up its convergence.

Fig. 2 shows a representative run of the two strategies and the baseline (no adaptation applied) on the "sharp increase" scenario. We note that Reactive improves the utility faster than ϵ-greedy. However, Reactive also incurs cost at the beginning since the increase in the number of servers during the first 500 secs increases the web server cost without bringing the response time below the threshold (a point heavily penalized by the utility function).

In Fig. 3 the average utilities and the percentage changes of both strategies over 30 runs are shown. The scenario was split into 16 segments of equal length. Each run used the same request rate trace, but randomized the time to serve a request and the algorithm (in case of ϵ-greedy). Looking at the percentage changes, we deduce that the pattern of initial cost of Reactive in Fig. 2 is also manifested in the per-segment analysis. Despite this initial cost, Reactive has benefit that is on average much higher than that of ϵ-greedy (since the latter may be trapped in local optima in its exploration). Hence, the system selects Reactive over ϵ-greedy for this scenario class.

[1] tools/plotResults.R at https://github.com/EGAlberts/swim/tree/ICSA2022

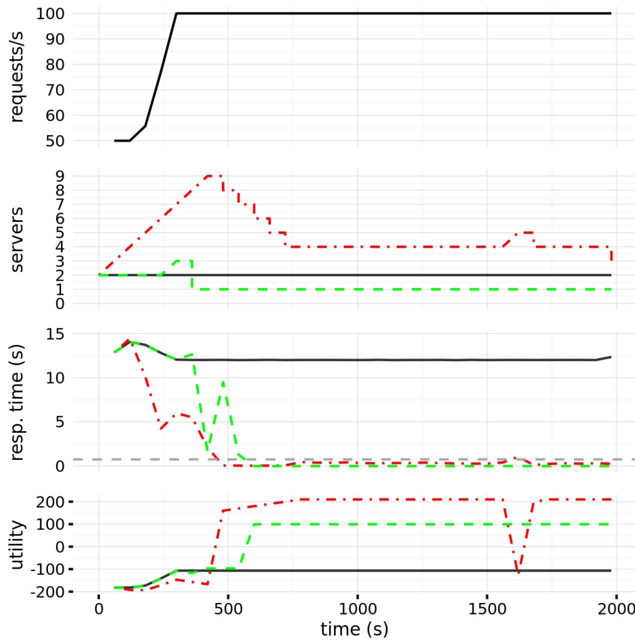

Fig. 2. Comparison of strategies on a "sharp increase" scenario. Black solid lines denote the baseline; red dot-and-dashed lines denote the Reactive strategy; green dashed lines denote the ϵ-greedy.

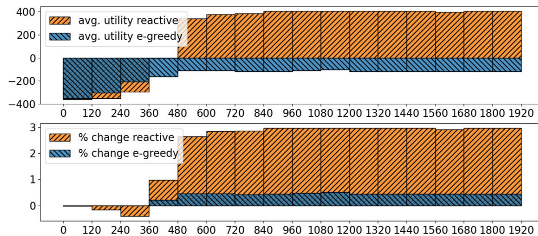

Fig. 3. Average utility and percentage change over 30 runs.

IV. Related Work

In software architecture, the Cost Benefit Analysis Method (CBAM) [9] provides guidelines for the analysis of architectural decisions from an economical perspective considering costs, benefits, and risks at design time. Our approach is inspired from CBAM and aims to assist the continuous decision-making at design and runtime for dynamic systems, e.g., self-adaptive. As mentioned in [10], there is no method considering design and runtime evaluations. Benefits and costs in self-adaptation have been first addressed in [11]. Cost-benefit analysis has been also used within a framework based on the technical debt paradigm and online learning to decide when and whether to adapt at runtime [12] – but not to compare strategies at runtime as we propose. Finally, 3 components of experimentation cost, i.e. time, adaptation, and endurability, have been proposed in a framework for online experiment-driven adaptation [13]. To summarize, we are not aware of any other work that defines a ROI for self-adaptive systems.

V. Conclusions and Future Plans

This paper presented our novel approach for cost-benefit analysis of adaptation strategies through the computation of ROI. This enables us to have an economics-driven and holistic perspective on strategies based on benefits and costs. We answer the initial research questions (Section I) as follows:

Q1: Does the development of a particular self-adaptation strategy for a software system pay off? Answer: Calculate the ROI for each self-adaptation strategy to estimate if it pays off.

Q2: Which of the self-adaptation strategies is the most appropriate (in terms of costs and benefits) to be applied to the current context? Answer: Compare the net profit, i.e., runtime benefit minus cost, of each of two or more strategies when more than one strategy may be applied and choose the one with the highest net profit for the current scenario.

Future developments will consider further costs and benefits (as proposed in [3], [14]–[16] for the ROI computation. We plan to apply our approach to other artifacts (e.g., SEAMS artifacts[2]) to enrich and generalize it.

References

[1] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda, "BUNGEE: an elasticity benchmark for self-adaptive iaas cloud environments," in *SEAMS 2015*, 2015, pp. 46–56.

[2] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. R. Schmerl, and R. Ventura, "Evolving an adaptive industrial software system to use architecture-based self-adaptation," in *SEAMS 2013*, 2013, pp. 13–22.

[3] G. A. Moreno, A. V. Papadopoulos, K. Angelopoulos, J. Cámara, and B. R. Schmerl, "Comparing model-based predictive approaches to self-adaptation: Cobra and PLA," in *SEAMS 2017*, 2017, pp. 42–53.

[4] E. M. Fredericks, I. Gerostathopoulos, C. Krupitzer, and T. Vogel, "Planning as Optimization: Dynamically Discovering Optimal Configurations for Runtime Situations," in *SASO 2019*, 2019, pp. 1–10.

[5] D. Weyns, M. Usman Iftikhar, and J. Söderlund, "Do external feedback loops improve the design of self-adaptive systems? A controlled experiment," in *SEAMS 2013*, May 2013, pp. 3–12, iSSN: 2157-2321.

[6] G. A. Moreno, B. Schmerl, and D. Garlan, "SWIM: an exemplar for evaluation and comparison of self-adaptation approaches for web applications," in *SEAMS 2018*, 2018, pp. 137–143.

[7] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment." ICST, Mar. 2008, p. 60.

[8] C. J. C. H. Watkins, "Learning from delayed rewards," PhD dissertation, King's College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[9] R. Kazman, J. Asundi, and M. Klien, "Making Architecture Design Decisions: An Economic Approach:," Defense Technical Information Center, Fort Belvoir, VA, Tech. Rep., Sep. 2002.

[10] D. Sobhy, L. L. Minku, R. Bahsoon, T. Chen, and R. Kazman, "Run-time evaluation of architectures: A case study of diversification in iot," *Journal of Systems and Softwware*, vol. 159, 2020.

[11] S. Cheng, D. Garlan, and B. R. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *SEAMS 2009*, 2009, pp. 132–141.

[12] T. Chen, R. Bahsoon, S. Wang, and X. Yao, "To adapt or not to adapt?: Technical debt and learning driven self-adaptation for managing runtime performance," in *ICPE 2018*, 2018, pp. 48–55.

[13] I. Gerostathopoulos, F. Plasil, C. Prehofer, J. Thomas, and B. Bischl, "Automated online experiment-driven adaptation-mechanics and cost aspects," *IEEE Access*, vol. 9, pp. 58 079–58 087, 2021.

[14] E. Kaddoum, C. Raibulet, J. Georgé, G. Picard, and M. Gleizes, "Criteria for the evaluation of self-* systems," in *SEAMS 2010*, 2010, pp. 29–38.

[15] C. Raibulet and F. Arcelli-Fontana, "Evaluation of self-adaptive systems: a women perspective," in *ECSA 2017*, 2017, pp. 23–30.

[16] C. Raibulet, F. Arcelli-Fontana, and S. Carettoni, "A preliminary analysis of self-adaptive systems according to different issues," *Softw. Qual. J.*, vol. 28, no. 3, pp. 1213–1243, 2020.

[2] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/