Proactive utility: Framework and approach

Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud

H2020-ICT-2018-2020
Leadership in Enabling and Industrial Technologies: Information and Communication Technologies

Grant Agreement Number
871643

Duration
1 January 2020 –
31 December 2022

www.morphemic.cloud

Deliverable reference
D2.3

Date
30 June 2021

Responsible partner
UiO

Editor(s)
Marta Różańska

Reviewers
Paweł Skrzypek, Alexandros Raikos

Distribution
Public

Availability

www.MORPHEMIC.cloud

Executive summary

This deliverable provides an initial description of the framework for estimating the proactive utility based on the overall goals set by the DevOps and the predicted execution context. The intention is to present and discuss the approaches for modelling utility functions as well as analyse the architecture of components involved in proactive adaptation and the sequence flows of the key processes, such as utility function construction and proactive optimization.

This document provides the description of various approaches for utility function modelling that leads to Cloud application resources optimization together with the discussion on fundamental concerns about forecasting in control loops. Various ways of modelling and estimating the utility function are described, discussed, and initially evaluated. What is more, this document also provides an architectural overview of the Proactive Utility framework, which is a part of the proactive adaptation feature of the MORPHEMIC Pre-processor platform.

Authors
Geir Horn, Kyriakos Kritikos, Maciej Riedl, Marta Różańska, Michał Semczuk, Jean-Didier Totow, Anna Warno, Yiannis Verginadis

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643

Table of Contents

List of Tables

List of Figures

1 Introduction

This document provides an architectural overview of the Proactive Utility framework, which is a part of the proactive adaptation feature of the MORPHEMIC Pre-processor platform. The goal of the proactive adaptation (also known as proactive reconfiguration) feature is to provide the ability to optimize the application's deployment in a fixed time horizon in the future. It means that the application will be optimized for future execution conditions, allowing ample time to reconfigure resources before they are really needed by the application.

The proactive adaptation can be discerned into reactive or proactive depending on when it reacts to a specific problem like a Service Level Objective (SLO) violation. While reactive adaptation is fully accurate in the sense that the adaptation is triggered when the problem has appeared, it suffers from the issue that a reaction has a certain latency which can take multiple minutes in the context of application scaling in the Cloud. In this sense, such a delay might be as critical as: the problem can become exaggerated once we finally once we detect it as very negatively impactful, which can lead to major economic loss due to huge violations of Service Level Agreements (SLAs), or it might not even exist anymore, leading to wasted resources with no actual gain. On the other hand, proactivity allows us to prevent these kinds of situations. However, it suffers from the fact that there is an uncertainty in terms of whether the problem will appear, and of the intensity or criticality of the problem, which leads to the argument that resources might be wasted either with no actual concrete cause, or by overreacting to a problem which is not as big as it has been predicted. As such, it could be advocated that there is a complementarity between these two approaches such that they are applied in conjunction or that there is a need to couple the most efficient type of approach, i.e., the proactive one, with suitable and highly accurate prediction techniques. These concerns are further discussed in Section 3.

This document also provides a methodology whereby the DevOps may model the functional form of the utility for each measurable metric considered by choosing from a catalogue of parameterized template function forms, and then the functions of individual metrics are combined to form the overall utility function for the application. It provides the description of various approaches for utility function modelling that leads to Cloud application resources optimization together with the discussion on fundamental concerns about forecasting in control loops. Various ways of modelling and estimating the utility function are described, discussed, and initially evaluated. The directly modelled utility functions, described in Section 5.1, from high-level policies can simply be used by use case partners during the preparation of a CAMEL Model for their use-case applications. The marginal metric utility function approach, described in Section 5.2, gives good results in terms of evaluating the change in the utility when new measurements are being gathered. This approach needs to be investigated further on how to include that kind of utility function in the decision-making process. On the other hand, the utility metric approach, described in Section 5.3, gives promising results, and it is already fully designed and included in the software architecture of the proactive adaptation feature in the MORPHEMIC platform. It is important to notice that this document provides the initial description of the overall proactive utility framework and the final results of the work as well as the implementation details will be described in deliverable D2.4 *Proactive utility: Algorithms and evaluation*.

1.1. Structure of the document and Intended Audience

The document contains the following sections which are intended for the target audience:

- User preferences in Cloud application resources optimization – in this section the state-of-the-art analysis, i.e., the discussion about utility-based and rule-based approaches in Cloud application resources optimization that gives the motivation of the work reported in the later sections is provided. This section should be read by all research partners to understand the current work and approaches for proactive utility and optimization
- Forecasting in control loops – in this section the fundamental concerns about forecasting in control loops are discussed. This section should be read by all partners, especially those involved in the work reported in WP2 Proactive model morphing and WP3 Polymorphic planning and adaptation work packages.
- Proactive utility modelling – this section presents two new approaches for modelling a utility function using high-level utility policies and template functions. Both approaches are discussed and initially evaluated. The description of utility function modelling and representation in CAMEL model is also provided. What is more, this section proposes a set of predefined utility functions that can be used by use cases partners and any other person interested in designing the utility function for Cloud application resources optimization. This is the reason that this section

should be read by use case partners while it can be also interesting for MORPHEMIC users from outside the project.
- Architecture: Proactive adaptation approach – this section presents the overall architecture of the proactive adaptation feature together with the sequence diagrams of three key processes that are a part of proactive adaptation: creating the utility function formula from high-level policies, proactive optimization, and utility function value calculation. In addition, the description of key components involved in the proactive adaptation is provided. This section should be read by all technical and use case partners.
- Conclusions - this section summarizes the deliverable and draws directions for further work.

2 User preferences in Cloud application resources optimization

Maintaining complex distributed applications using multiple deployment possibilities at the same time and adapting the application's configuration and resources depending on time variate demand is a daunting task that should be supported by autonomic application management platforms [2] The decisions made by the platform must reflect the implicit and explicit goals of the application's users and owner.

In this section, the description of two main approaches for the user preferences in the optimization of Cloud application resources is provided. Three main surveys [3] [4] [5] have been found as relevant towards resource optimization and auto-scaling. Most of these surveys indicate that the reviewed approaches are either proactive or reactive. Further, they indicate that the reviewed approaches tend to focus on either low-level, high-level or both kinds of metrics. They also agree that the utilized resource optimization technique is considered as a dimension, various categories of approaches can be deduced, including utility-based (which means based on the utility function) and rule-based approaches (which means based on the set of rules). The utility-based approaches are the most flexible and allow for complex modelling of user preferences that enables the most accurate optimisation of Cloud applications[6]. In MORPHEMIC, we generally follow a mixed approach where a utility-based approach is followed to find the best deployment configuration, and a rule-based approach is followed that results into a set of Service Level Objectives for the proactive triggering of the reconfiguration process.

2.1 Utility-based approaches

The decisions made by the optimization platform must reflect the implicit and explicit goals of the application's users and owner. Consequently, the decisions should aim at maximizing the application's *utility* as the utility is an established concept representing choices of a rational economic actor [7]. The iterative autonomic computing feedback loop, Monitor, Analyse, Plan and Execute - with Knowledge (MAPE-K) [8], has been used to build management frameworks for mobile applications [9], for ubiquitous computing systems [10], and for autonomic management of applications deployed simultaneously to multiple Cloud providers [6]. All of these approaches are based on capturing the application's utility in a *functional* expression, $U\big(\boldsymbol{c}(t_k)\big|\,\boldsymbol{\theta}(t_k)\big)\colon V \mapsto [0,1]$, mapping the application configuration $\boldsymbol{c}$ from the application's *variability space*, V, to the unit interval $[0,1]$ given a vector of measurements, $\boldsymbol{\theta}(t_k)$, representing the application's *execution context* at the current time t_k . The best configuration at time t_k is then taken to be the configuration maximizing the utility expression, $\boldsymbol{c}^*(t_k) = max_{\vec{c}\in V} U\big(\boldsymbol{c}(t_k)\,\big|\,\boldsymbol{\theta}(t_k)\big)$.

The key to the success of these approaches is consequently how well the utility function is able to capture the goals of the application's owner. Experiments conducted with experienced DevOps engineers show that formulating explicitly the utility function is difficult and discouraging [11]. The main problem is that it is difficult to foresee and model how a change in the configuration will affect the execution context of the application as the measurements represent an indicator of how well the application is performing. For instance, Horn and Skrzypek presented an application decrypting encrypted documents as on demand by the application's users [6]. The utility of that application had possibly conflicting goals: the cost of the deployment to be minimized and the performance of the application to be kept stable and at a satisfactory level for the application's users. The utility value returned from the utility function should balance these goals and reflect the DevOps engineer's preference for a given application configuration. The DevOps engineer may relatively easily decide on the cost utility as a function of the number of instances, and the performance utility as a function of the average response time per document requested by the application's users. However, it can be very difficult to model these utilities as required by the current approaches as a functional expression that captures the dependency between the metrics influenced by the application deployment configuration being the choice of the Cloud provider, the Virtual Machine type to use, and the number of Virtual Machines.

The utility in autonomic computing is almost certainly *multidimensional* combining objectives measured on different scales. Overall, the multiple objectives can be conflicting and the utility function balances the different utility dimensions. A partial remedy is to formulate the utility function as an affine combination of the normalized utility decomposed into its various *dimensions*, $d \in 1, \ldots, D \subset N^+$. The utility is then specified independently for dimensions like performance and cost. This leads to a multi objective optimization problem, and it may be possible to find robust weights of the affine combination for the bi-objective case [12]. Utility modelling starts from the *attributes* used by the decision maker to decide on the utility, *i.e.* the attributes may well correspond to the utility dimensions. Keenley showed that if the attributes are independent, then the utility function is either *additive* or *multiplicative* in individual marginal univariate utility functions over the individual attributes[13]. The additive property was used by the UTilité Additive (UTA) [14] method for constructing marginal utility functions from piecewise linear functions given a set of known 'actions' and the evaluation of these actions with respect to all decision criteria and a partial ranking of the different criteria. Yang and Sen proposed an interactive step trade-off approach soliciting the decision maker's evaluation feedback in the UTA method to eliminate possibly inconsistent user preference information[15]. The UTA methodology has later been extended for non-additive utility functions that allows for interaction among the decision criteria [16]. Ultimately, one may directly learn the decision maker's preferences from a set of pairwise criteria comparisons using an artificial neural network[17]. All of these approaches can collectively be understood as driven by preference data provided by the decision maker evaluating multiple alternative configurations. However, the sheer size of the application's variability space will most likely prevent the application of interactive data driven approaches in autonomic computing.

A different approach is used when modelling uncertainties in the utility function assuming that there is a deterministic utility represented by a known multivariate *value function* balancing the different decision criteria, and an uncertain univariate utility function taking the value returned by the value function as an argument. For the example above one may model the cost of an application configuration as a value function over the application configuration as the price of the VMs used for the deployment. However, there is a *risk* that keeping the cost low will impact negatively the other decision criterion, namely the application performance. Matheson and Abbas show how describing the risk attitude of the decision maker for one decision attribute allows the determination of the utility function for the other attributes from the value function [18]. Risk aversion is linked to the partial derivatives of the utility function. Abbas gives general formulas for these derivatives [19], while Alghalith demonstrates how the utility function can be derived using Taylor series expansion of the univariate utility function over the given value function [20]. The issue with the value function approach is that specifying the value function is just as cumbersome as specifying directly the utility function, and therefore it does not help to overcome the issue of modelling the utility function for autonomic computing.

There is an alternative provided that there is systematic variation in the attribute values implying that more of every attribute is better than less. As an example, consider the Cloud application deployment attribute dimensions 'unconsumed budget' and 'application performance'. Obviously, one will desire more of both, and there will be minimal values in both dimensions. Furthermore, if one additionally has that the utility will be zero whenever any single attribute attains its minimum value, then the utility function will belong to the class of *attribute dominance utility functions* [21]. This class has similar mathematical properties as multivariate joint cumulative probability distributions, and the joint utility function can be derived from the univariate 'marginal' utility functions for each of the utility dimensions. Abbas and Howard suggested using *copula theory* [22] developed for joint probability distributions to model the utility, although this would natively result in a utility function where all mixed partial derivatives with respect to the attribute dimensions would be positive (an n-increasing utility) *functions* [21]. This consequence can be undesirable for many utility functions. Abbas subsequently proposed specialized *utility copulas* relaxing the requirement of zero utility at the lower limit of any single attribute and the n-increasing property [23], and defined the *Archimedean utility copulas* inspired by the similar copula concept from statistics [24]. Abbas and Sun developed a methodology for constructing an Archimedean utility copula from a set of preferences given by the decision maker [25], and recently they analysed a form of the Archimedean utility copula with polynomial generating functions [26].

It should be noted that the marginal utility functions used for the utility copula are functions of the decision attributes. In other words, the utility copula theory provides an excellent framework for combining the utility *dimensions* for an autonomic computing system. However, the user must still define the link between the marginal utility and the possible actions and system state represented respectively by the configuration $\boldsymbol{c}(t_k)$ and the application execution context $\boldsymbol{\theta}(t_k)$. The marginal metric utility function approach described in section 5.2 closes this gap with a methodology for directly modelling each marginal utility dimension based on the decision maker's assessment on the impact of the application execution context *only*.

2.2 Rule-based approaches

Rule-based approaches to resource optimisation are approaches that utilise rules, usually specified by the user, in order to drive the resource optimisation. Such rules are most of the time scaling, indicating when exactly to horizontally scale an application as, e.g., it is lacking some resources to anticipate the current workload. The condition part of these rules includes constraints over specific quality attributes or metrics like average response time. For instance, a scaling rule could indicate that when response time is greater than 10 minutes, then additional resources (e.g., VMs) need to be added to the application at hand. Scaling rules can be characterised as local. This means that they usually concern a specific part of an application, like a component. This results in horizontally scaling the instances of application components as well as the need to introduce a load balancing component (in the application architecture) on top of these application component instances in order to evenly distribute the load among them.

Cloud applications are usually multi-layer such that they might have different resource requirements and scaling policies per each tier. Furthermore, it might be possible that the scaling policies (per tier) might need to change over time due to the dynamicity of the workloads, the unpredicted behaviour of Cloud environments and the need to further optimise them so as to perfectly cover the most suitable scaling behaviour for the Cloud application. In this respect, rule-based approaches, which means the application of fixed scaling policies or rules even with a different content per tier would not suffice or work, would lead to significant cost spending while it would also cause Service Level Agreement (SLA) [27] violations and medium or low service levels[1] being delivered. As such violations concern formal agreements with application clients, penalties can be enforced leading to economic loss while the reputation of the cloud provider can be reduced.

Finally, we should put in the table the aspect of agility: rule-based approaches can be discerned into reactive or proactive depending on when they react to a specific problem like an SLO violation. In the following, we analyse the related work in Cloud application scaling in separate categories depending on the complexity of the rules that are being employed, or derived and subsequently applied by respective approaches in sight of the issues that were aforementioned.

The norm in the Cloud market is that most of the Cloud providers only supply the ability to specify and execute simple scaling rules per application components or groups of components which define specific single or complex conditions on metrics and the respective number of instances to increase or decrease. Focusing on the most popular Cloud Provider, Amazon Web Services (AWS)[2], there has been some evolution towards more flexibility in the definition of such rules. In particular, different scaling rules[3] can now be specified:

- The well-known simple scaling rules with some flexibility in terms of how to determine the actual number of resources to increase or decrease: either this number is fixed or related to the actual current number of resources as a percentage (e.g., 20% increase)
- Step-Wise, violation-based scaling rules which supply mappings from different, non-overlapping violation ranges[4] to specific modifications on the number of Cloud resources being exploited. Each mapping focuses on a different violation range with respect to a threshold. For instance, suppose that we have an upper threshold of 80% on average CPU utilisation. Then, we have the ability to specify two mappings as contents of the desired scaling rule:
- (0,10] violation → 20% increase: this rule indicates that if the actual average CPU utilisation is between (80,90], then there should be an increase in the amount of allocated resources by 20%

[1] A service level is a specific set of quality capabilities or requirements concerning an application or service. Such a level then comprises a set of Service Level Objectives (SLOs) as constraints on quality metrics or attributes. For instance, one service level might comprise two SLOs: one indicating that average response time of the application should be less than 5 seconds and another highlighting that average application availability cannot be less than 99.99%.
[2] https://aws.amazon.com/
[3] https://docs.aws.amazon.com/autoscaling/ec2/userguide/scaling_plan.html
[4] A range indicating how much a specific SLO like average CPU utilisation < 80% is violated

- (10,20] violation → 40% increase: this rule indicates that if the actual average CPU utilisation is between (90,100], then there should be an increase in the amount of allocated resources by 40%
- Target tracking scaling rules which determine the amount of resources to increase or decrease to horizontally scale the Cloud application based on a specific target condition given by the user. Such target conditions are mainly specified for negatively monotonic metrics[5] for which we need to supply upper thresholds. The resource update determination relies on the distance between the current metric value and the given threshold while the timing of the scaling execution varies depending on whether there is a need to scale-out or scale-in the application. In particular, scaling-in is more granular in comparison to scaling-out which is applied as fast as possible. As AWS states [6], such rules are not so efficient for scaling groups of a small size. However, they can be combined with other kinds of scaling rules on other metrics in such a way that scaling-out is conducted when one of the rule fires while scaling-in is conducted when all scaling rules fire.

AWS and Google Cloud Platform (GCP) [7] now support the notion of predictive scaling, which further improves local scaling especially when combined with target tracking. This enables to predict when a specific rule will be violated in the future and trigger it to scale the application on time such that the respective SLO will not be eventually violated. Proactiveness enables to save costs as adaptation issues are anticipated as early as possible before they become larger and more difficult to address. However, as we talk about local scalability rules, this means that the application is still adapted locally while, e.g., an increased workload could create performance issues in multiple parts of an application and not just one.

While we do acknowledge the increased flexibility in scaling rule specification, there are still various problems to address, such as how can we dynamically adapt the first two rule kinds (simple & step-wise), how to determine the best thresholds and resource number updates in the first place and how to guarantee that the Cloud application is globally scaled in an optimal manner. This last problem is a known issue for local-scalability rules. The target tracking scaling rule type seems to be promising as it realises the concept of an auto-scaler, but the main question is how effective this is in sight of the issue that it does not work well for small scaling groups. Furthermore, it does not cover all possible metric kinds (e.g., positively monotonic) while it might not be so appropriate when the given threshold is not violated, e.g., it would still keep some excessive resources up even if they are not needed.

RightScale[8] has proposed a more efficient autoscaling algorithm that combines reactive scaling rules with voting. In particular, each VM instance includes a set of predefined rules that enable it to decide whether to scale-out or scale-in. All VM instances then communicate and via majority voting [9] [10], they can decide whether there is a need to scale-in or -out the Cloud application. While this scaling approach seems interesting, it still relies on predefined rules whose content need to be determined while it seems not to take into consideration the workload trend. In Simmons et al. [28], these issues are attempted to be resolved through a strategy-tree, a structure that enables to evaluate the deployed policy set and to switch between alternative strategies in a hierarchical manner. Three scaling policies were defined for this purpose which are matching different input workloads and the method relying on the strategy-tree will dynamically select one of them over time by considering the current workload trend.

Another interesting idea is called smart kill, which relies on the observation that partial VM usage hours are priced as full[29]. Thus, it is proposed that the respective VMs should not be destroyed before the whole usage hour is spent when the workload is low such that they could be re-used in case it is needed, e.g., change in workload.

Finally, the concept of dynamic thresholds is proposed by Lorido-Botran et al.[30]. Their suggestion is that, once the original threshold values are set up, they can be automatically updated based on the observed SLA violations during application runtime. Such an updating is conducted through meta-rules which explicate how the threshold values can be changed to better adapt to the workload.

[5] A negatively monotonic metric is a metric whose values should be minimised as much as possible. In other words, the lower is the value of the metric, the better is the respective utility (for the application). Examples of negatively monotonic metrics include response time, (network) latency and execution time.
[6] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html#target-tracking-considerations
[7] https://cloud.google.com/
[8] http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags/index.html
[9] https://docs.rightscale.com/cm/rs101/understanding_the_voting_process.html
[10] https://en.wikipedia.org/wiki/Majority_rule

Approaches utilising local scalability rules seem to have improved over time. In fact, we believe that the use of predictive scaling with target tracking is the right way to go forward, potentially with the addition of dynamic threshold capabilities (to, e.g., correct imprecise rule conditions). However, the aforementioned issues of how to specify rules in the first place and how to globally optimise an application still apply. In the first case, the specification of a local scalability rules condition is the most difficult problem, especially as there is a need to split an overall application quality requirement into multiple local ones. This issue is avoided by MELODIC/MORPHEMIC as resource optimisation relies on global application quality constraints that are directly derived from non-functional business requirements. However, we do acknowledge that MELODIC did not consider proactive adaptation and that is why now MORPHEMIC follows a predictive approach towards globally adapting a cloud application in order to gain the advantages of proactiveness. Further, MORPHEMIC globally adapts the cloud application such that it does not suffer from the second aforementioned issue of local adaptation. In our view, MORPHEMIC combines the best from both worlds: it utilises preferences to specify an overall utility formula that guides the global application optimisation while it does follow both a proactive and reactive approach towards such optimisation by considering precise, global SLO rules. The proactive approach adapts the application even before a SLO violation occurs while the reactive approach takes over to make some necessary corrections. Such corrections can happen in two cases: (a) there is a still a subtle SLO violation that needs small treatment even if proactiveness was applied; (b) prediction is not possible or accurate which can well happen in the beginning of application provisioning.

3 Forecasting in control loops

The utility function will be used as a representation of the goals and the deployment objectives of the organisation deploying the application. It will be used as a substitute of a response from that organisation when evaluating different deployment options, and its value must reflect the assessment on a proposed deployment configuration as if it was made by the organisation owning and operating the application managed by MORPHEMIC. Thus, in a nutshell, MORPHEMIC tries to find the configuration that maximizes the utility function. This section discusses the implications this has with respect to the proactive decisions.

3.1 Utility and optimisation

From economic theory we know that a rational agent, like MORPHEMIC, should make *decisions* that maximizes the agent's utility [1]. In contrast with classical control theory where the evolution of the controlled system whose efficiency is optimized can be described by the first principles and physical properties of an ideal system, potentially operating in noisy conditions, the decisions made by a rational agent will be influenced by the agent's context. The 'context' can here be thought of as everything in the agent's environment or psychology that directly or indirectly influences the decision. Hence, the decisions are *conditioned* on a context that must be incorporated into the utility function before the utility can be maximized by a decision that can be called *optimal for the current context*.

Even though the utility value can be computed in many ways opaque to MORPHEMIC, it is convenient to think about the utility as a *function* that is evaluated and maximized. As per the previous paragraph, the utility function is conceptually a function specific to the current context. Each decision has a separate utility function, but this can hopefully be represented by a *family* of functions whose parameters are context dependent. For instance, a decision to buy an electrical vehicle will depend on the decision maker's parameters for cost and on range, and the values to assign to these parameters depend on the decision maker's context like salary, family conditions, and need to travel far. Furthermore, the utility function represents the trade-off between the cost and the range based on the anticipated use of the car or the probability of having to make longer journeys: For instance, the decision maker's empirical trip length probability distribution parameters can be included as parameters in the utility function balancing the cost and range factors. Therefore, it is assumed in the following that the utility function for the specific decisions the Cloud deployment configuration of a particular application can be represented as a single function with context dependent parameters.

The values of these context dependent parameters must be obtained by monitoring the running application and thereby obtaining quantifiable measurements of the application's execution context. Computing systems have offered many standard context variables as the load on the Central Processing Unit (CPU) or the memory consumption of an application. Additionally, MORPHEMIC allows the Cloud platform level monitoring to be augmented with application-level sensors that monitor values across all Cloud platforms used to deploy the application. This allows the application's execution context to be monitored with more relevant parameters at different application levels. For instance, the number of concurrent application users can be directly measured without having to infer this as a complex mapping from the observed load on the computing infrastructure. It is therefore an important task for the DevOps engineers responsible for the application to identify the parameters that are most relevant for characterizing the application's execution context, and then provide the necessary sensors coupling the information available only inside the application to the distributed Event Monitoring System (EMS) of MORPHEMIC.

These measurements are then used to specify the context parameters of the utility function. In practice, we will see that the utility can be described as combination of unary functions of the context measurements. However, the result is that MORPHEMIC will find the best application configuration for the current context by optimizing the utility function for these specific application execution context parameters, and this configuration may no longer be the best when the application's execution context changes. Furthermore, if the best application configuration found in this way is different from the configuration currently running, the currently running application will be reconfigured to the newly found configuration as this will be optimal for the application's current execution context.

Two important observations must be made at this point: The optimization problem to be solved will be different each time since it is solved for different context parameters; and after the reconfiguration the measurements of the execution context will be different because of the new application configuration deployed.

3.2 Time, and time again

Each context parameter measurement represents an *event* occurring at a *time point*. There is no reason to assume that the sampling of events is *regular*, *i.e.,* that the interval between two time points for the series of measurement is constant. Consider as an example the context parameter measuring the number of users of an application: The users obviously arrive or depart at their own will at irregular time points. Regular sampling may only be used for context parameters whose underlying signal is *continuous*, albeit one may philosophically question if something in a computer will ever be continuous as it is always a question of the granularity of the clock available for the sampling. Should one decide for regular sampling of events, it is important that the sampling frequency is high enough to satisfy the conditions of the Whittaker-Shannon sampling theorem so that the underlying continuous signal can be reconstructed from the sampled values [44].

The implication of this observation is that each context parameter is measured on a set of time points that is specific to the context parameter. The set of event times for two different context parameters may not have any common elements. However, the application's execution context for the time point *now* is defined as the vector of all the current context parameter values. In other words, the application's execution context changes whenever anyone single context value changes, and the time axis of the application's execution context change events is the union of all the event time points sets of all the context parameters.

This means that the application's execution context is constant only until the next event time of any of the context parameters. As the utility of the deployed configuration is conditioned on the application's execution context, this means that the utility value of the current configuration changes whenever the application's execution context changes. Keeping the deployment optimal would therefore require finding again the deployment configuration maximizing the utility for the application's current execution context whenever that context changes. It is important to notice that the reasoning and redeployment takes time. It may therefore be computationally challenging or even impossible to keep the deployment optimal given the potentially frequent changes in the application's execution context.

It should be noted that even though the utility of the current deployment configuration may no longer be optimal after a change in the application's execution context, it does not mean that the current deployment configuration is *infeasible*. Generally, the utility is maximized subject to some operational *constraints*, and the deployment configuration stays feasible if none of the constraints are violated. Thus, one may continue to use the current configuration for a changed execution context provided the configuration is feasible also for this new context. It is computationally possible to assert the operational constraints for each change in the application's execution context, and only if one of the constraints is violated will it be necessary to solve the optimization problem to find the optimal configuration for the changed context. For this reason, MORPHEMIC guarantees an *optimized application configuration*, but not necessarily the optimal application configuration for the application's current execution context.

The constraints that trigger a reconfiguration are called *Service Level Objectives* (SLOs) in the application modelling language CAMEL and they will be evaluated for every measurement arriving from the managed application as each change of a metric indicates a change in the application's execution context. The assessment of the SLOs is made by the *Meta Solver* component, and if any of the SLOs is violated, it will request a new configuration optimized for the current set of metric values, *i.e.* the application's *current* execution context. Solving the optimisation problem will take some time, and after the best configuration has been found, the current deployed application must be reconfigured to this new configuration. The reconfiguration involves the *Adapter* first planning and ordering the reconfiguration actions, *e.g.* starting and stopping VMs and connecting them in the right sequence, and then the *Executionware* will enact this plan to change the deployed application configuration into the optimal configuration for the context. This process is illustrated in Figure 1.

The figure illustrates two main problems:

1. Solving the optimisation problem and enacting the adaptation take time. This time is a *random variable* of unknown distribution since the time it takes to do the two steps depends on the complexity of the problem and the type of changes observed in the application's execution context.
2. During the reconfiguration lag the application keeps on running in the current constraint violating configuration. This means that other SLOs than the one(s) triggering the reconfiguration process may also be violated during the reconfiguration lag. However, the optimization problem is solved for the application's execution context as it was when the first SLO violation was detected. This implies that the configuration found optimal for this application execution context may immediately cause new SLO violations when deployed as the continued changes in the application's execution context during the reconfiguration lag have not been taken into consideration.

The first problem is further discussed in this deliverable. The main idea is to forecast the application execution context at the time when the reconfiguration is completed. Given that the reconfiguration lag is a random variable, one may compute the appropriate upper quantile of its empirical distribution and use this as an upper bound for the reconfiguration time, and thereby also for the application execution context prediction horizon.

The second problem can be overcome by using a solver that is tolerant to changing conditions since a change in the application execution context should result in a change in the constraints and in the *utility function* whose maximisation is the objective of the solver. This means that if the solver evaluates the utility function and the constraints for the same candidate application configuration at different time points, it will have different values returned from the utility function, and the constraints may suddenly deem the proposed configuration as infeasible. Thus, seen from the solver, it operates in a random environment, and through multiple interactions with this environment it will over time *learn* the best average solution for the application's current execution context. The implication is therefore that the solver should be *stateful* and run continuously alongside the application and based on *reinforcement learning* as it may need to learn and then unlearn optimal configurations as the execution progresses. Different solvers and alternatives tried and yet to be investigated is the subject of Deliverable *D3.3 Optimized planning and adaptation approach*.

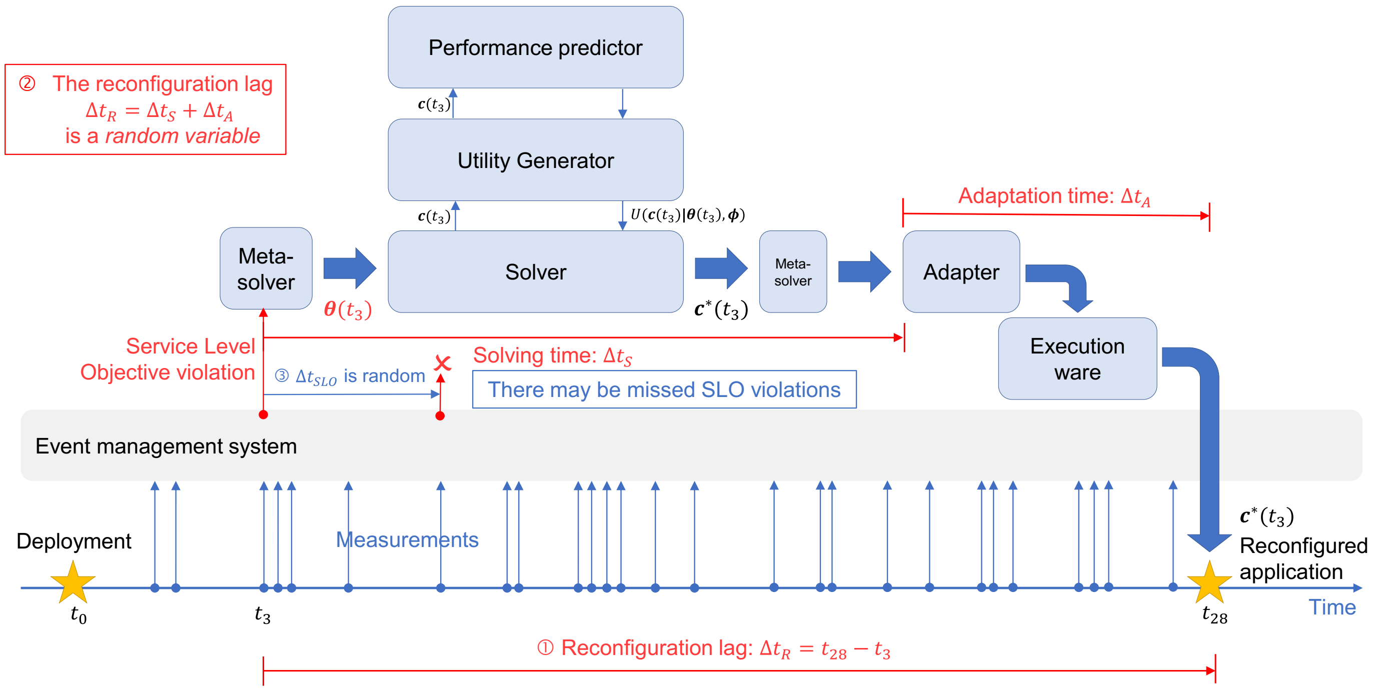

② The reconfiguration lag $\Delta t_R = \Delta t_S + \Delta t_A$ is a *random variable*
Performance predictor
$c(t_3)$
Utility Generator
$c(t_3)$
$U(c(t_3)|\theta(t_3),\phi)$
Adaptation time: Δt_A
Meta-solver
Solver
Meta-solver
Adapter
$\theta(t_3)$
$c^*(t_3)$
Execution ware
Service Level Objective violation
③ Δt_{SLO} is random
Solving time: Δt_S
There may be missed SLO violations
Event management system
Deployment
Measurements
$c^*(t_3)$
Reconfigured application
Time
t_0
t_3
t_{28}
① Reconfiguration lag: $\Delta t_R = t_{28} - t_3$

Figure 1 - The application reconfiguration process and the components involved starts from the violation of a Service Level Objective constraint violation and ends with the Executionware completing the adaptation actions transforming the deployed application configuration into the optimal configuration found by the solver for the application's current execution context.

3.3 The data farming example

To illustrate the utility guided optimisation discussed in the previous section, consider a data farming application. This class of applications consists of a set of jobs to be executed. These jobs can, for instance, be a set of simulations of some complex phenomenon where each simulation is executed on different system parameter settings to see which parameters that will produce a simulated system response most closely resemble the observations, and thereby reveal the best fitting system parameters for the situation at hand. The jobs are data parallel and so there are no dependencies among the jobs during execution. The jobs are executed on one or more workers, and there is a coordinating dispatcher or scheduler submitting jobs to ready workers and collects the results of the executions. Note that this execution pattern also matches many big data processing applications executed by frameworks like Spark[11] and Hadoop[12]. The pattern with one scheduler and many independent workers is also known as High Throughput Computing (HTC), for which the HTCondor[13] is a generic scheduler. The pattern is similar to many High Performance Computing (HPC) applications; however, HTC tasks submitted to the workers have data dependencies and exchange data and results over the course of the computations.

One will often have a deadline for completing all jobs in a data farming application since it may be necessary to know the system parameters before the system under investigation changes state, or to find timely a solution to a data mining problem. This is where the elasticity of Cloud computing comes in handy as one may reduce the completion time, known in scheduling as the *makespan*, by adding more workers. The minimal makespan can obviously be achieved by having one worker per job, with the makespan being the time it takes to compute the most time-consuming job. Running the application in this optimal way is also the costliest as more worker machines must be needed, and if the computational deadline is much longer than the longest computation time for any job, it will be beneficial to run several jobs sequentially on a smaller number of workers. Consequently, the best utility for the owner of a data farming application is the compromise between performance and cost. These are two conflicting *utility dimensions* of the problem. The

[11] https://spark.apache.org/
[12] https://hadoop.apache.org/
[13] https://research.cs.wisc.edu/htcondor/

desired best utility will obviously be for the application deployment configuration that provides the least number of workers that is sufficient for the data farming experiment to reach its deadline.

One may safely assume that one job needs at least one core. Thus, the parameter that decides on the performance is the 'number of cores' across all workers. The cores can be provided by scaling the workers vertically, *i.e.,* by providing more cores per worker Virtual Machine (VM), or horizontal scaling by providing more VMs with one worker installed per VM. In either case one may simplify the application modelling by assuming that all workers are of the same VM type with the same number of cores. The performance parameter of interest is, therefore, the product of the number of worker VMs with the number of cores per VM. With respect to the cost utility, it is important to consider these separately as it may be cheaper to deploy many smaller VMs than a few larger VMs, or vice versa, depending on the pricing models of the Cloud providers being used. Thus, the performance parameter of interest for the DevOps engineer is composed of two variables that can be set independently for the deployment: the number of workers, and the number of cores per worker.

The time needed for a worker to finish a job is *a priori* unknown. To assess if the deadline is met by the current configuration of workers, one needs an upper bound on the time needed for the remaining jobs. The easiest way to obtain this upper bound is to observe the time it took to finish each job and record the maximal completion time. A crude estimate for the upper bound on the total time needed is then found by multiplying the recorded maximal completion time with the number of jobs remaining. Under the worst-case assumption that each job gets only one computational core, one can divide by the number of cores to get an estimate for the computational time needed to complete the remaining jobs. This approach is sensitive to outliers, *i.e.* extreme jobs taking much longer time than the others, as it assumes that all future jobs will take the longest possible time. The computation time for each job is really a random variable with an unknown distribution. One may estimate empirically this distribution as the work progresses by measuring and recording the time taken to process each job when the job is completed. Depending on the amount of error one can tolerate, the corresponding upper quantile is computed from the empirical distribution and multiplied with the number of remaining jobs to give an estimate for upper bound on the time needed to complete the remaining jobs. This approach leads to two context parameter values available at the time point whenever a job finishes: the number of remaining jobs, and the upper quantile of the job computation time distribution.

The utility of a deployment configuration candidate can then be found by playing a 'what-if-game' against the currently deployed configuration: Given the current configuration and measurements, what would the utility be if we changed the configuration? This means that one has to include the current configuration is a 'measured' part of the application's execution context as regards potentially better configurations suggested by the solver during the optimisation process. Hence, there are two context parameters for the current configuration corresponding to the variables set for the configuration candidate proposed by the solver: the current number of worker VMs, and the current number of cores on each worker VM. Table 1 summarises the context parameters of the performance utility, its decision variables, and the concepts that can be derived through functional relations to other concepts.

Table 1 - The concepts of the performance utility dimension of the data farming example

Concept	**Symbol**	**Formula**
Event time when one job finishes computation	t_k	
Constants vector $\boldsymbol{\phi}$		
Deadline for all jobs	ϕ_1	
Scaling parameter	ϕ_2	$\phi_2 > 0$
Decision variables of the next configuration vector $\mathbf{c}(t_{k+1})$		
Number of worker VMs	$c_1(t_{k+1})$	
Cores per worker	$c_2(t_{k+1})$	

Concept	Symbol	Formula
Application's execution context parameter vector $\boldsymbol{\theta}(t_k)$		
Number of remaining jobs	$\theta_1(t_k)$	$= \theta_1(t_{k-1}) - 1$
Upper quantile of job time distribution	$\theta_2(t_k)$	From the empirical distribution of $\Delta t_i = t_i - t_{i-1}$
Total time spent until now	$\theta_3(t_k)$	$= t_k$ measured from the wall clock
Time bound on remaining jobs	$\theta_4(t_k)$	$= \theta_1(t_k) \cdot \theta_2(t_k)$
Future number of cores	$\theta_5(t_k)$	$= c_1(t_{k+1}) \cdot c_2(t_{k+1})$
Bound on the wall clock time for remaining jobs	$\theta_6(t_k)$	$= \frac{\theta_4(t_k)}{\theta_5(t_k)} = \frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1(t_{k+1}) \cdot c_2(t_{k+1})}$
Bound on completion time	$\theta_7(t_k)$	$= \theta_3(t_k) + \theta_6(t_k) = \theta_3(t_k) + \frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1(t_{k+1}) \cdot c_2(t_{k+1})}$
Margin on the deadline	$\theta_8(t_k)$	$= \theta_7(t_k) - \phi_1 = \theta_3(t_k) + \frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1(t_{k+1}) \cdot c_2(t_{k+1})} - \phi_1$

The utility value is confined to the unit interval where unity means perfect utility and zero means an unacceptable configuration. The performance utility should therefore be high when the margin on the deadline is positive and then degrade when the solution indicates that the deadline is barely met, or not met at all. However, one should remember that the bound is computed on the upper quantile of the empirical computation time distribution, and most of the computations will finish before this bound. It is therefore reasonable to find a function that maps the margin on the unit interval, and the natural choice would be the sigmoid function $1/\left(1 + e^{-\phi_2 \cdot x}\right)$ with a negative argument $x = -\theta_8(t_k)$ scaled with ϕ_2 so that the utility will be 1/2 when the deadline is exactly met, and thereafter drop towards zero utility when the deadline is exceeded. Eliminating the derived context parameters using the formulas of Table 1, this yields the performance utility of Equation (1).

$$U_P(\boldsymbol{c}(\boldsymbol{t}_{k+1}) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) = \frac{1}{1 + \exp\left(-\phi_2\left[\phi_1 - \left(\frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1(t_{k+1}) \cdot c_2(t_{k+1})} + \theta_3(t_k)\right)\right]\right)} \tag{1}$$

Finally, there is a need to assess the cost of the deployment to be able to assess the cost utility dimension. The cost utility should be decreasing as the cost of the deployment increases. A negative exponential function seems a natural assumption; however, it must be normalized relative to the available budget such that it can be ensured that the utility value is unity when the cheapest possible virtual machine is used. Table 2 gives the concepts of the cost utility extending the vectors defined for the performance utility in Table 1.

Table 2 - The concepts of the cost utility dimension of the data farming example

Concept	Symbol	Formula
Constants vector $\boldsymbol{\phi}$		
Available budget	ϕ_3	
Price of cheapest available worker VM	ϕ_4	$\phi_3 \geq \phi_4$
Scale parameter	ϕ_5	$\phi_5 > 0$
Shape parameter	ϕ_6	$\phi_6 > 0$
Maximum conserved budget	ϕ_7	$= \phi_3 - \phi_4$
Application's execution context parameter vector $\boldsymbol{\theta}(t_k)$		

Concept	**Symbol**	**Formula**
Price of the cheapest VM with the required number of cores	$\theta_9(t_k)$	$= \text{Price}\big(c_2(t_{k+1})\big)$
Currently conserved budget	$\theta_{10}(t_k)$	$= \phi_3 - c_1(t_{k+1}) \cdot \theta_9(t_k)$

A functional negative exponential family of functions based on the scale and shape parameter useable to represent the cost utility is given by Equation (2).

$$U_C(\boldsymbol{c}(\boldsymbol{t_{k+1}})\,|\,\boldsymbol{\theta}(t_k),\boldsymbol{\phi}) = \exp\left(\frac{\phi_5}{{\phi_7}^{\phi_6}} - \frac{\phi_5}{\theta_{10}(t_k)^{\phi_6}}\right) = \exp\left(\frac{\phi_5}{[\phi_3-\phi_4]^{\phi_6}} - \frac{\phi_5}{[\phi_3 - c_1(t_{k+1})\cdot \text{Price}(c_2(t_{k+1})]^{\phi_6}}\right) \quad (2)$$

These two dimensions are then combined into the overall utility for this class of applications as a simple affine combination where $w \in [0,1]$ is the weight given to the performance utility

$$\text{U}(\boldsymbol{c}(\boldsymbol{t_{k+1}})\,|\,\boldsymbol{\theta}(t_k),\boldsymbol{\phi}) = w \cdot \text{U}_P(\boldsymbol{c}(\boldsymbol{t_{k+1}})\,|\,\boldsymbol{\theta}(t_k),\boldsymbol{\phi}) + (1-w)\cdot \text{U}_C(\boldsymbol{c}(\boldsymbol{t_{k+1}})\,|\,\boldsymbol{\theta}(t_k),\boldsymbol{\phi}) \quad (3)$$

A simulation experiment was set up to verify the above utility modelling. It is assumed that the duration of the jobs are uniformly distributed between one and 5 minutes, and that the number of jobs will take just above 5 hours to execute on a single core. The goal is to complete the jobs in one hour, $\phi_1 = 3600s$. The scale parameter for the performance utility was taken as $\phi_2 = 15$. The available budget was taken to be $\phi_3 = 11$ units, and the price of the cheapest virtual machine was set to unity, $\phi_4 = 1$, and the scale and shape parameters $\phi_5 = 10$ and $\phi_6 = 2$. The number of cores is taken to be unity in this simulation so that only the number of worker instances, c_1, is the only configuration variable. The price of this small machine equals the cheapest machine, which was set to unity. The performance dimension and the cost dimension of the utility is equally weighted, $w = 1/2$.

The result of the simulation is shown in Figure 2 for the ideal situation where the reconfiguration lag is zero. An Auto-Regressive Integrated Moving Average (ARIMA) model is fitted to the number of jobs remaining for the first 1100 simulated seconds. Since the number of jobs remaining ideally should be on the straight line from the initial number of jobs to zero jobs at the expected deadline, the ARIMA model is a good fit for this example. The corresponding utility function values evaluated at each job completion event are shown in Figure 3, and the controller succeeds in keeping the utility value high with a preference for performance in the beginning shifting dynamically to a reduction of cost when it seems that the deadline most likely will be met.

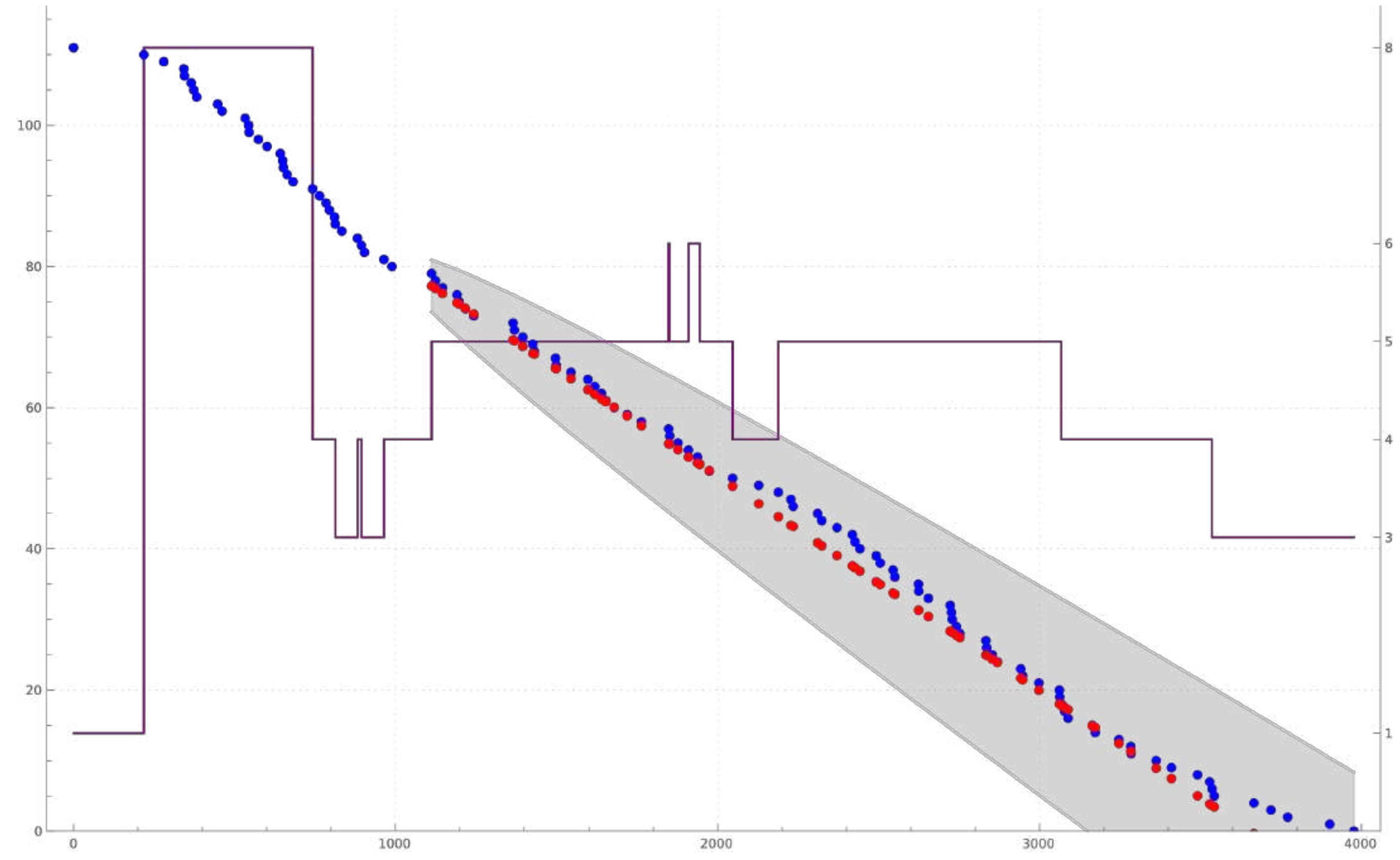


Figure 2 - Simulation of the data farming deployment with the goal of finishing all simulation in one hour. The right-hand ordinate axis and the blue dots show the number of jobs remaining, and the right hand axis the staircase purple line show the number of cores used. The red dots are the predictions of an ARIMA model fitted to all the job completions over the first 1100s and the shaded area is the 95% prediction limits.

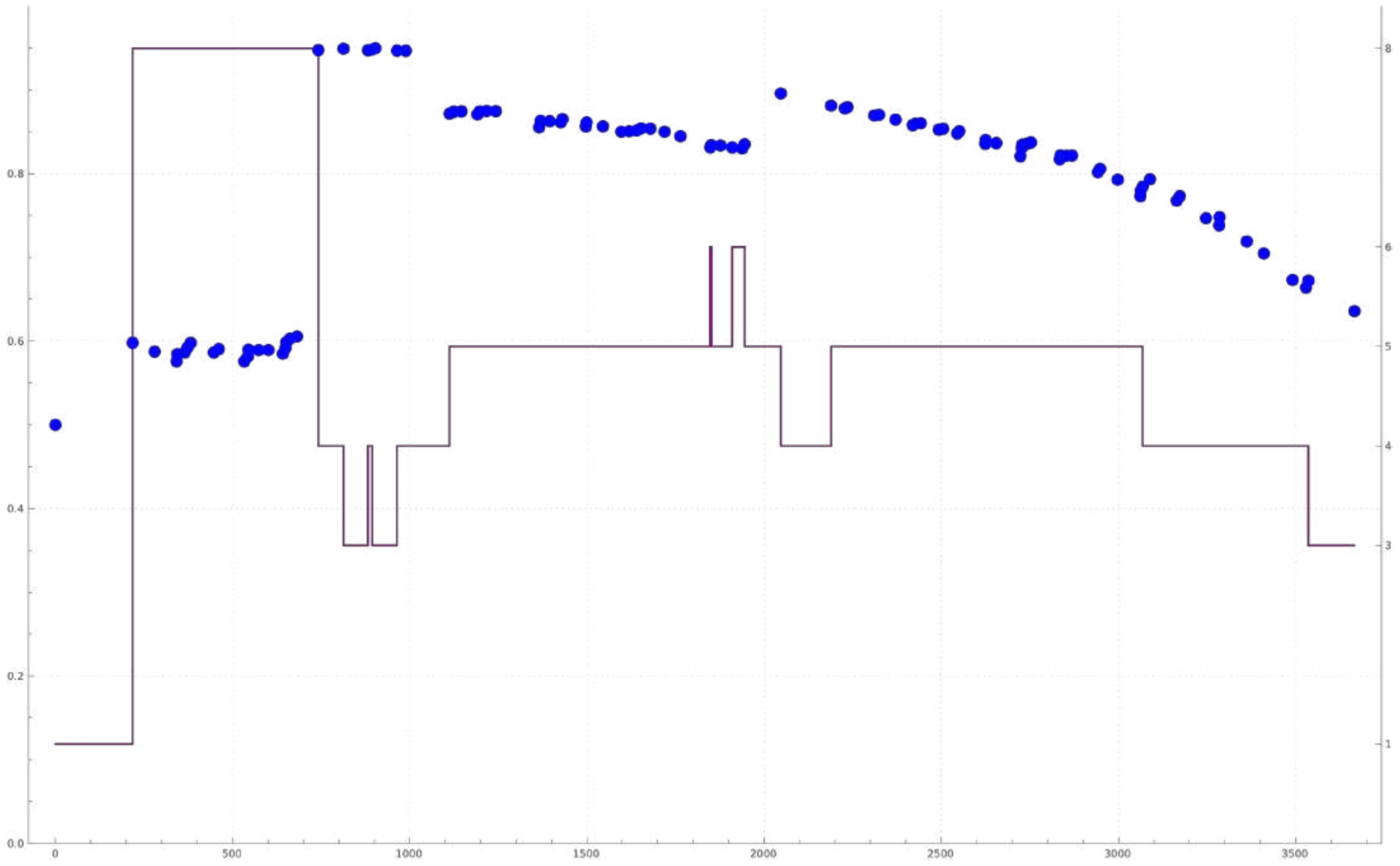


Figure 3 - The utility values computed at each job completion event are given as blue dots on the right hand ordinate axis plotted against the reconfiguration events and the number of cores given by the straight purple line valued on the right hand ordinate axis.

4 Conditional execution context forecasting

It is easy to be misled by the good prediction seen in Figure 2. The forecaster has an easy job since the number of jobs remaining is the output of the controlled system, *i.e.*, the relation is a straight line, and the slope of that line reflects the *control actions* taken by increasing or decreasing the number of cores to maximize the utility. The goal of the control actions is to ensure that the time it takes between two job completion events is approximately constant with an upper bound sufficiently short to complete the remaining jobs within the deadline. Figure 4 shows the Δt_k at each event time together with the 95% quantile of the empirical Δt_k distribution. The control is successfully achieving this result. The duration of each job was uniformly distributed between 60s and 300s with an expected duration of 180s, and the observed value of a between two job completion events is mostly below 100s as the effect of jobs executing in parallel. However, Figure 4 also shows that it is virtually impossible to forecast the Δt_k values, and even forecasting the upper bound of the empirical distribution is very difficult.

This relatively simple example reveals the fundamental issue: *The monitored metric values are conditioned on the control actions*. This means that:

1. The forecaster can only use data for the current application configuration to predict the future execution context of the application, and this prediction will only be valid until the next reconfiguration.
2. There are correlations between all the metric values used to measure the application's execution context as they are all dependent on the application configuration.
3. The dependency on the configuration reflects as a correlation between the configuration variables and the measured metric values. To illustrate: It does not help being able to predict confidently the system output, for this example the number of remaining jobs, unless one can also forecast the impact of control actions on the derivative of this curve to ensure that there will be zero remaining jobs predicted for at the time of the deadline.

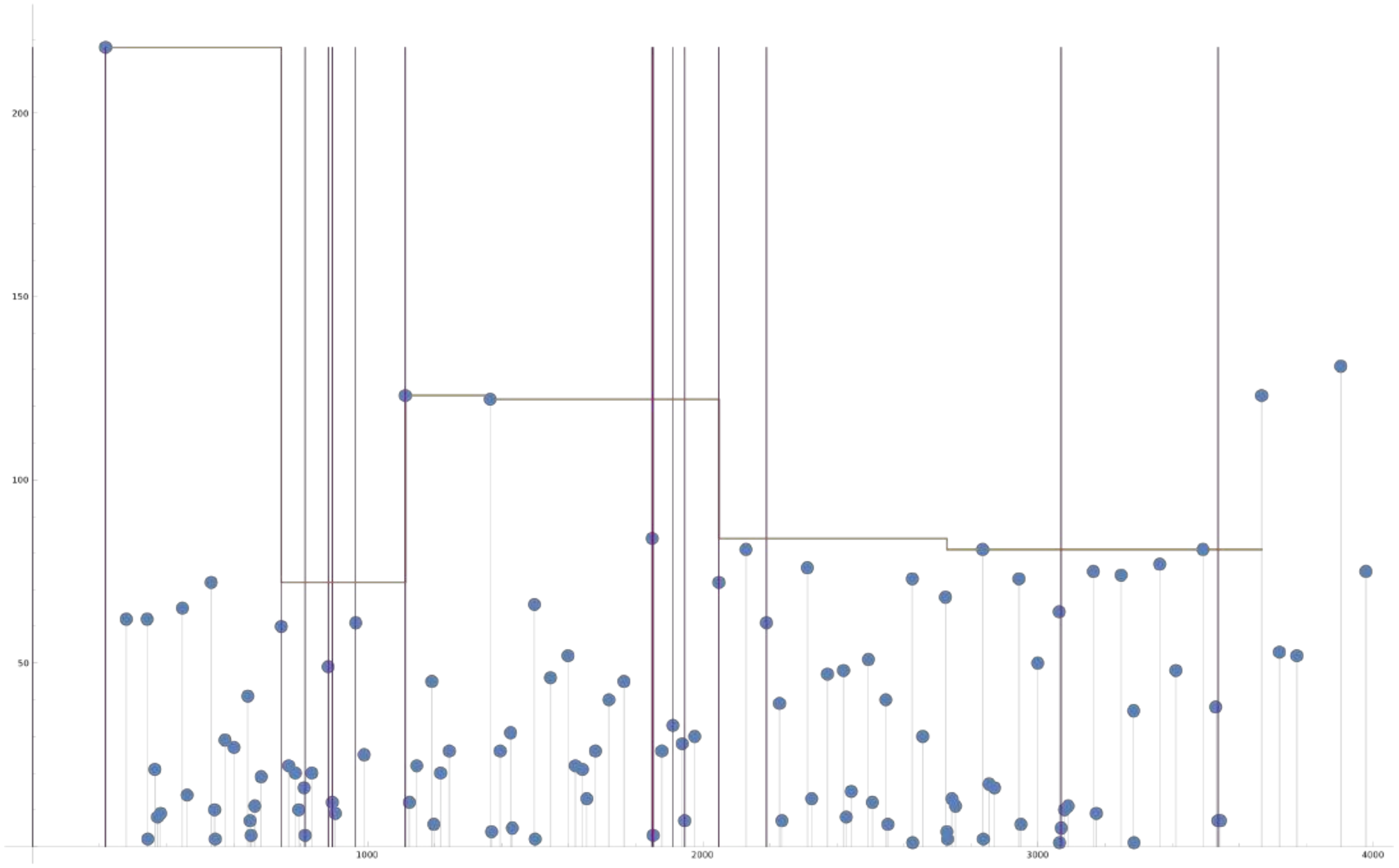


Figure 4 - The Δt_k events when a job completes as blue dots and the 95% quantile of the empirical Δt_k distribution as a solid yellow line. The purple vertical lines are the reconfiguration events.

The consequence of these observations is that one cannot predict individually the metric values of the application's execution context. One must predict the entire context vector of all metric values at once, and it only makes sense to consider multivariate forecasting methods. Forecasting will be discussed in deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*.

Furthermore, one cannot predict the execution context vector apart from the deployment configuration. In other words, it makes sense to talk about the 'state' of the application as one single vector consisting of both the context variables and the metric measurements, $\boldsymbol{s}(t_k) = [\boldsymbol{c}^T(t_k), \boldsymbol{\theta}^T(t_k)]^T$. The implications of this conclusion for the optimization process and the solvers will be further discussed in deliverable *D3.3 Optimized planning and adaptation approach*. For the system identification and full system state forecasting we will investigate using Kálmán filters [31]. These have recently been proposed for Cloud data centre management [48]. Kálmán filters are the best-known estimators that take the system dynamics, *i.e.*, the control actions and the actual application configuration, into consideration. Kálmán filters are also known to be globally optimal with minimum mean square errors and producing maximum likelihood system state estimates when the system dynamics are linear, and the noise components can realistically be assumed to be Gaussian. However, linearity cannot be assumed for the application dynamics, and one alternative is to investigate linearization approaches for hybrid systems, *i.e.,* consisting of both discrete and continuous dynamics [33]. We will also investigate non-linear extensions to the Kálmán filters like recursive filtering [34], or if it is possible to model the system dynamics as a Wiener system with a linear and a non-linear part and use specialised recursive Kálmán type estimators for such systems [35].

The utility function is already a mapping from the configuration vector given the application's current execution context vector to the unit interval, in other words, the full state vector, and so considering the full state vector as opposed to separating the configuration from the application's execution context has no impact on the utility function modelling.

5 Proactive utility modelling

As it has been discussed in Section 2.1, it is often difficult for the DevOps engineer to provide a utility function formula by hand. More specifically, a set of some high-level goals, such as keep the cost as low as possible or optimize the response time to the user, can be specified, but the way a change of VMs parameters influences the performance is difficult to foresee and define explicitly. What is more, these days the need for DevOps engineers is bigger than the number of qualified people such that low-code and high-level approaches must be developed. That is the reason for the investigation of high-level utility policies and more user-friendly utility function modelling as presented in this section.

During the work on this deliverable, various ways of more user-friendly utility function modelling in terms of less effort to generate the utility function were investigated and preliminary evaluated. The first task was to indicate, with the help of MORPHEMIC use case partners, the most common and needed high-level policies for the user such as "*optimize response time*" or "*minimize cost*". These *policies* are often formulated with limited knowledge of how it translates to actual application configuration and how it relates to monitoring *measurements* taken from the deployed application configuration. The results of discussions with the MORPHEMIC use case partners and with other potential MORPHEMIC adopters during workshops and webinars (see D7.1 *Initial Dissemination and Communication Report and Plan*) showed that even different software applications may have similar high-level optimization policies that should be eventually modelled as utility functions. During the work, seven *utility dimensions* for the most popular high-level policies were modelled. They are presented in Section 5.1. The directly modelled utility functions from high-level policies can simply be re-used by the use case partners during writing the CAMEL models of their applications. It is important to notice that the number of templates can be easily extended in the future.

The second approach is the marginal metric utility function modelling, which is based on measurement evaluation and can be useful for detecting the validity and optimality of the currently deployed configuration. It is described in Section 5.2. The marginal metric utility function approach gives good results in terms of evaluating the change in the utility when new measurements are being gathered. This approach needs to be investigated further on how to include that kind of utility function in the reasoning process.

The last evaluated approach called *utility metric* is the new methodology for representing user preferences and creating a utility function. This approach, presented in Section 5.3, gives some promising results while a respective Proof of Concept for the MORPHEMIC platform is being implemented according to the design described in sections 5.4, 6.1.1 and 0.

All presented approaches in this section therefore, they will be further investigated, and the result of the final evaluation will be provided in D2.4 *Proactive utility: Algorithms and evaluation*.

5.1 High-level policies and overall goals set by the DevOps

In this section, we present the directly modelled utility functions for the utility dimensions corresponding to the identified high level policies. These directly modelled utility functions from high-level policies can simply be re-used by use case partners in their own CAMEL model. The idea is that the overall utility function can be constructed as an affine combination of these dimensional utility functions where each dimension is weighted according to the importance of that dimension for the DevOps engineer such that the sum of the weights is unity. It is important to notice that all functions are modelled for one or two component applications and in case of more components, more decision variables can be used. One should also note that each utility function for each dimension is modelled independently, and the notation and symbols used will be local to that utility dimension. Thus, when the partners are re-using these dimensional utility functions proper variable substitutions must be applied when representing the overall utility function of the managed application.

5.1.1 Expected response time

From the performance point of view, it would obviously be best if each application user had a dedicated virtual machine, as this would minimise the response time. However, this would inflate the cost of the deployment. Therefore, it is more appropriate to define the expected response time $\overline{T}$ and the maximum acceptable response time T^+, which is a threshold that should not be breached. We should also specify the default timeout time, T_D, after which the server returns the timeout and the request is not served at all. What is more, there is a need to measure the current average response time $\theta_1(t_k)$. The decision variables can be changed to be more complete, but for the simplification of the example, it is assumed that there are two main decision variables: c_1, which is the number of instances, and c_2, which is the number of cores available at each instance. If the application owner thinks that also other decision variables, such as the amount of memory, should be included, they can be incorporated in the same formula in a similar way.

We propose to represent this utility function as a probability density function with bounded support, such as the Beta distribution, $\beta(\tau \mid a, b)\ for\ \tau \in [0,1]$, and given its two constant parameters a and b that are found with respect to T_D and $\overline{T}$. The current configuration is defined as $c^*{}_1(t_k), c^*{}_2(t_k)$. The performance utility can therefore be expressed as follows:

$$\upsilon_{\text{time}}\big(\boldsymbol{c}(t_{k+h}) \,\big|\, \boldsymbol{\theta}(t_{k+h})\big) = \beta\left(\frac{\theta_1(t_k)}{T^+} \cdot \frac{c^*{}_1(t_k) \cdot c^*{}_2(t_k)}{c_1(t_{k+h}) \cdot c_2(t_{k+h})} \,\middle|\, \phi_1, \phi_2\right) / \beta^+(\phi_1, \phi_2)$$

Note that the above utility function is normalized by $\beta^+(a, b) = \max_{\tau} \beta(\tau \mid a, b)$ to ensure that the utility is a value within the unit interval.

This function was used in the Secure Document use case used as an example by Horn and Skrzypek [6], and as an example of a typical e-commerce application by Horn and Rozanska [12].

5.1.2 Finish simulation on time

This utility function can be used, for example, for all applications that are running simulations or that perform machine learning training. By the start of the computation, the number of training jobs *N* will be given. However, the time it takes to do one training will depend on the data being processed and the complexity of the model to be trained. Thus, at the event time point t_k then k jobs have completed, and the number of jobs remaining, $\theta_1(t_k)$, will be a change in the application's execution context. It is also possible to measure the upper $1 - \alpha/2,\ \alpha < 1$ quantile of the empirical training time distribution, $\theta_2(t_k)$. Thus, the predicted time needed for the parallel computations of the remaining training tasks is $\frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1 \cdot c_2}$, where the nominator is an upper bound for the time needed to complete the remaining jobs under the assumption that each calculation takes as long as the upper quantile currently observed. The denominator represents the number of available cores for the job as the number of worker machines c_1 times the number of cores per worker c_2. To know the predicted overall completion time for all calculations one must add to $\frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1 \cdot c_2}$ the total time one has spent for the jobs until now, $\theta_3(t_k)$.

The utility of the deployment should be close to unity for a given set of workers, c_1, if this predicted overall completion time is less than the deadline. The farther beyond the deadline the predicted completion is, the lower the utility should be. Thus, a natural utility function form would be a sigmoid function $1/(1 + e^{-ax})$ with a negative argument *x* scaled with *a* so that the utility will be 1/2 when the deadline is exactly met. The performance utility for the timeliness of the application is then

$$\upsilon_{\text{deadline}}\big(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}\big) = \frac{1}{1 + \exp\left(-\phi_2\left[\phi_1 - \left(\frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1(t_{k+h}) \cdot c_2(t_{k+h})} + \theta_3(t_k)\right)\right]\right)}$$

where $\vec{\varphi}$ is a vector of fixed parameters; here, the target deadline and scaling parameter $\phi_2 > 0$. The larger this scaling parameter is, the quicker will the utility value switch from unity to zero if the predicted overall completion time exceeds the soft deadline $\phi_1 = T^+$.

It is worth mentioning that this type of the function is being used to optimize the Genome Big Data application[14], and also by the MELODIC business user, the AI Investments[15] company. There is also a work in progress model the utility function based on this function by MORPHEMIC use-case partner ICON.

[14] https://h2020.melodic.cloud/genom-data-processing-success-story/
[15] https://aiinvestments.pl/

5.1.3 Locality utility

One may want to include the distance between two deployed components into account. It is possible by using the distance between two locations as it can give an estimation about the possible latency between two components without measuring it. There is a need to have decision variables c_1, c_2 as latitude and longitude of the first component and c_3, c_4 - latitude and longitude of the second component. Assuming that latitude and longitude is given in degrees, the geodesic distance between two deployed components can be calculated using spherical law of cosines: $acos(\sin(c_1 \cdot \pi/180) \cdot \sin(c_3 \cdot \pi/180) + \cos(c_1 \cdot \pi/180) \cdot \cos(c_2 \cdot \pi/180) \cdot \cos(c_4 \cdot \pi/180 - c_2 \cdot \pi/180)) \cdot R$ and it can be normalized to be within zero and one. R is the earth's radius, and the mean radius is 6371km. The locality utility is then:

$$\upsilon_{\text{locality}}(\boldsymbol{c}(t_{k+h}) \mid \boldsymbol{\varphi}) =$$
$$1/(1 + acos(\sin(c_1 \cdot \pi/180) \cdot \sin(c_3 \cdot \pi/180) + \cos(c_1 \cdot \pi/180) \cdot \cos(c_2 \cdot \pi/180) \cdot \cos(c_4 \cdot \pi/180 - c_2 \cdot \pi/180)) \cdot 6371000)$$

5.1.4 RAM usage

Utility function that optimises the percentage RAM usage may have a form of the Beta distribution function, because Beta distribution is defined on an interval from 0 to 1. The desired value of RAM used should be specified by the user, probably around 80% of the total RAM available and when the RAM usage increases, the amount of RAM is not sufficient for the proper application execution. When the RAM usage decreases, the utility is lower for the user because some of resources are wasted and the cost can be reduced. There is a need to include two decision variables: c_1 which is the number of component instances and c_2 which is the amount of RAM available in one component instance. What is more, the RAM usage on each instance θ_1 should be measured and the percentage RAM usage on all instances can be calculated as $\theta_2(t_k) = \frac{\sum \theta_1(t_k)}{c^*{}_1(t_k) \cdot c^*{}_2(t_k)}$.

Assuming that the RAM usage scales linearly when adding more machines, it is possible to express the RAM usage utility as follows:

$$\upsilon_{\text{RAM}}\big(\boldsymbol{c}(t_{k+h}) \,\big|\, \boldsymbol{\theta}(t_{k+h})\big) = \beta\left(\theta_2(t_k) \cdot \frac{c^*{}_1(t_k) \cdot c^*{}_2(t_k)}{c_1(t_{k+h}) \cdot c_2(t_{k+h})} \middle| \phi_1, \phi_2\right) / \beta^+(\phi_1, \phi_2)$$

5.1.5 CPU usage

The utility function that considers the CPU usage can be a very similar function to the previously described RAM usage utility. The decision variables can be c_1 which is the number of component instances and c_2 which is the number of cores available in one component instance. Measured CPU usage on the one instance is usually measured as an average value so $\theta_1(t_k)$ can be calculated as the average CPU usage for all instances. Assuming that the usage of the CPU scales linearly, the utility function can be defined as:

$$\upsilon_{\text{CPU}}\big(\boldsymbol{c}(t_{k+h}) \,\big|\, \boldsymbol{\theta}(t_{k+h})\big) = \beta\left(\theta_1(t_k) \cdot \frac{c^*{}_1(t_k) \cdot c^*{}_2(t_k)}{c_1(t_{k+h}) \cdot c_2(t_{k+h})} \middle| \phi_1, \phi_2\right) / \beta^+(\phi_1, \phi_2)$$

5.1.6 Cores cost utility

A negative exponential is a good candidate for the cost utility dimension as it increases when the cost decreases. It needs to be scaled so that the exponent is zero when the cost is at the minimum, and then the exponent should be more and more negative as the deployment cost increases. Let ϕ_3 be the available budget and ϕ_4 the price of the least expensive VM possible. Furthermore, it is reasonable to assume that the price of a machine is decided by the number of cores c_2 it offers as for many cases it is the most important factor. A reasonable cost utility function is therefore

$$\upsilon_{\text{core-cost}}(\boldsymbol{c}(t_{k+h}) \mid \boldsymbol{\varphi}) = \exp\left(\frac{\phi_5}{[\phi_3 - \phi_4]^{\phi_6}} - \frac{\phi_5}{[\phi_3 - c_1 \cdot Price(c_2)]^{\phi_6}}\right)$$

where $\phi_5 > 0$ is a scale parameter and $\phi_6 > 0$ is a shape parameter. Note that the denominator of each term of the exponent represents the remaining budget raised to the power of ϕ_6. In the first term the remaining budget is defined by deploying a single virtual machine with the least cost, whereas in the second term the remaining budget is given by the number of component instances multiplied by the price of each instance, *i.e.,* the total cost of the deployment configuration.

5.1.7 Cost per user

This cost utility function improves the previously described function for cost utility, which only considers the cost per deployed core, i.e., it does not take into account the current workload and the number of potentially served users. The utility is best captured using the cost per served user as a parameter. We note that the same price for VM is considered to be a lower cost if more users can be served using this VM. Let P^+ and P^- be the maximum and the minimum prices for available VMs, ${c_1}^+$ and ${c_1}^-$ be the maximum and the minimum number of application servers that are limited by the application owner, and $Price(\vec{c})$ be the price of deployment represented by $\vec{c}$. The cost utility can be expressed as:

$$\upsilon_{\text{cost-user}}\big(\boldsymbol{c}(t_{k+h})\,|\,\boldsymbol{\theta}(t_{k+h})\big) = \frac{\theta_2(t_{k+h}) \cdot P^+ \cdot {c_1}^+ - \big(Price\big(\boldsymbol{c}(t_{k+h})\big)/(1-\theta_3(t_{k+h}))\big)}{\theta_2(t_{k+h}) \cdot P^+ \cdot {c_1}^+ - P^- \cdot {c_1}^-}$$

which is a similar function to the standard normalization function $\big(x - min(x)\big)/\big(max(x) - min(x)\big)$. The numerator is changed to $max(x) - x$ to reflect the fact that minimum price gives the highest cost utility. The main parameter is the price of configuration divided by the ratio of not served requests $\theta_3(t)$, which gives the information what is the deployment cost per served request. The ratio of served requests is calculated as $1 - \theta_3(t)$.

The maximum value is calculated as the most expensive deployment, which is the biggest possible number of the most costly VMs divided by the worst possible ratio of served requests, which is only one from all requests $\theta_2(t)$ received in one second: $(P^+ \cdot {c_1}^+)/[1 - (\theta_2(t) - 1)/\theta_2(t)]$. After the transformation, the form of the maximum value is $\theta_2(t) \cdot P^+ \cdot {c_1}^+$. The minimum price for the deployment is calculated as the minimum number of the least expensive machines divided by the best possible ratio of served requests that is one.

5.2 Marginal metric-based utility function modelling

As it was discussed in Section 2.1, it may be difficult for the DevOps engineer to manually define the utility function as a mathematical formula. In this section, we propose an alternative way to model and estimate the form of this utility function that can be then used in the optimization of Cloud application resources. Arguably, DevOps engineers and application owners have a better understanding of how the application utility will vary with changes in the application's execution context, i.e., the monitored metric values. The metric-based approach is based on this observation.

In this Section, it will be demonstrated that the approach for marginal metric utility functions in its present form can readily be used in Cloud resources optimization to detect the change in the quality of the currently deployed configuration. Moreover, the biggest advantage of this modelling is the fact that it can be extended and used as a part of MAPE-K loop in the optimization process. Hence, it is possible to make the reconfiguration decisions without having an explicit utility function formula.

5.2.1 Marginal metric utility function

Continuing the example from Section 2.1: the number of decoding server instances decides the cost, and the response time experienced for a user request indicates the application's performance. The DevOps engineer may relatively easily decide on the cost utility, and the performance utility as a function of the average response time per document requested by the application's users.

Thus, the DevOps should be able to specify to the optimization platform their preferences regarding the values of such metrics. These metrics will define the execution context in which the Cloud resources optimization process operates. We are going to show that, given the set of functions $u_i\big(\theta_i(t)\big)$ specified for each metric $\theta_i(t) \in \boldsymbol{\theta}(t)$ and a single optimal deployment configuration for a particular time point t_k i.e. $\boldsymbol{c}^*(t_k)$, it is possible to calculate the utility as the execution context changes. For this, no explicit formula for the utility function $U(\boldsymbol{c}(t_k)\,|\,\boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi})$ is needed. In other words, in this approach the user is asked to specify the preferences regarding the values of the metrics rather than a complete utility function.

We show that the change in utility in the neighbourhood of a given optimal configuration depends only on the measurements and, therefore, can be defined by functions over the metrics. The utility function $U(\boldsymbol{c}(t_k) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi})$ is essentially a multivariate function of all configuration variables $\boldsymbol{c}(t_k)$ and the *given* metric 'variables' assigned the values of the metric measurements $\boldsymbol{\theta}(t_k)$. The linearised utility around an optimal configuration $\boldsymbol{c}^*(t_k) = [c_1(t_k), c_2(t_k), \ldots, c_n(t_k)]^T$ and the application execution context $\boldsymbol{\theta}(t_k) = [\theta_1(t_k), \theta_2(t_k), \ldots, \theta_m(t_k)]^T$ for which $\boldsymbol{c}^*(t_k)$ is the optimal configuration is then given by the following equation:

$$\begin{aligned} U(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}) \\ &= U(\boldsymbol{c}^*(t_k) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) + \sum_{i=1}^{n} \frac{\partial U}{\partial c_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [fc_i(t_{k+h}) - c^*{}_i(t_k)] \\ &+ \sum_{i=1}^{m} \frac{\partial U}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [f\theta_i(t_{k+h}) - \theta_i(t_k)] \end{aligned}$$

Here, the vertical bar after the partial derivatives means that the derivative is evaluated at the given point. By assumption $\boldsymbol{c}^*(t_k) = max_{\vec{c}(t_k)\in V} U(\boldsymbol{c} \,|\, \boldsymbol{\theta}(t_k))$ and therefore, by the definition of optimality, the derivatives of any function at its optimal point are zero: $\frac{\partial U}{\partial c_1} = \frac{\partial U}{\partial c_2} = \cdots = \frac{\partial U}{\partial c_n} = 0$ This means that Equation 1 reduces to:

$$U(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}) = U(\boldsymbol{c}^*(t_k) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) + \sum_{i=1}^{m} \frac{\partial u_i}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [f\theta_i(t_{k+h}) - \theta_i(t_k)]$$

and the change in the utility can therefore be expressed as

$$U(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}) \; - \; U(\boldsymbol{c}^*(t_k) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) = \sum_{i=1}^{m} \frac{\partial u_i}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [f\theta_i(t_{k+h}) - \theta_i(t_k)]$$

Since the true utility function is unknown, one can freely define the partial derivatives of this unknown function. Given the utility functions defined for each of the metric values, $u_i(\theta_i)$, these can be taken as the marginal utilities:

$$\frac{\partial U}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} \overset{\text{def}}{=} \frac{\partial u_i}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))}$$

Thus, it enables us to model fully the unknown utility in the neighbourhood of the optimal configuration $\boldsymbol{c}^*(t_k)$ for $\boldsymbol{\theta}(t_k)$. It may seem surprising that the change in utility is not depending on $\boldsymbol{c}(t_{k+h})$. However, an optimal configuration is optimal until it must be changed as a result of a changed application execution context. The change in the application execution context is driven by the evolution of the metric values, and thus it is natural that $\boldsymbol{c}(t_{k+h}) = \boldsymbol{c}^*(t_k)$ as the *time* moves on until the change of the utility indicates that a new optimal application configuration must be found.

5.2.2 Algorithm for modelling user's utility function

The proposed method for retrieving DevOps engineer's preferences in accordance with the application's optimization goals is presented schematically presented algorithmically.

Algorithm 1 Modelling the utility function from marginal utility function templates

```
   Input: $\boldsymbol{f} = [f_1, f_2, \ldots, f_5]^T$
   Result: the utility function: $U\big(\boldsymbol{c}(t_{k+h}) \,\big|\, \boldsymbol{\theta}(t_{k+h})\big)$
1  $\boldsymbol{\theta}(t)$ ←The user defines metrics that are measured
2  $\boldsymbol{\theta}(t_k)$ ← The metric values are measured at time $t_k$
3  For ( $\theta_i \in \boldsymbol{\theta}$ )
```

```
4       $j \leftarrow$ The user chooses the index of the best $f$
5       $a, b \leftarrow$ The user defines $a, b$
6               $u_i(\theta_i) = f_j(\theta_i \mid a, b)$
7   $\boldsymbol{c}^*(t_k) = \max_{\boldsymbol{c} \in V} U(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k)) \leftarrow$ The user indicates optimal configuration
8   (Optional) $U_0 \leftarrow$ The user assigns the value of $\boldsymbol{c}^*(t_k)$
9   **If** ($U_0$ exists)
10      $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k)) = U_0$
11  **Else**
12      $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k)) = \left[\sum_{i=1}^{m} w_i \cdot u_i(\theta_i(t_k))\right] \leftarrow$The utility value of $\boldsymbol{c}^*(t_k)$ is calculated
13  **Return**
    $U(\boldsymbol{c}(t_{k+h}) \mid \boldsymbol{\theta}(t_{k+h})) = \left[\sum_{i=1}^{m} w_i \cdot u_i(\theta_i(t_k))\right] + \sum_{i=1}^{m} \frac{du_i}{d\theta_i}|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))}[f\theta_i(t_{k+h}) - \theta_i(t_k)]$
```

First, the DevOps engineer is asked to define the vector of metrics, $\boldsymbol{\theta}(t)$, that is considered important. Then a template utility function for each metric $\theta_i(t) \in \boldsymbol{\theta}(t)$ must be chosen. We describe the details of the five proposed template functions, $\boldsymbol{f} = [f_1, f_2, \dots, f_5]$ in Section 5.2.3 where we discuss their desirable properties. However, this approach is flexible, and any new template function can be added to $\boldsymbol{f}$ without any changes in Algorithm provided this new function is differentiable. A template function typically has some shape parameters that the DevOps engineer must tune and fix for the function to represent the univariate utility function for the metric value argument. Then, a time point t_k must be fixed as the origin of the linearised utility, and the measurements $\boldsymbol{\theta}(t_k)$ must be obtained for this time point. The DevOps engineer must state what is the optimal application configuration $\boldsymbol{c}^*(t_k)$ *for this particular time point* and, correspondingly, the perceived application utility value, $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) \in [0,1]$. It should be noted that fixing this utility value at one of the extremes will have a bearing on the template utility function forms and possibly the time of validity of the linearization because the $\sum_{i=1}^{m} \frac{\partial u_i}{\partial \theta_i}\Big|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [f\theta_i(t_{k+h}) - \theta_i(t_k)]$ must be strictly positive if $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) = 0$ or strictly negative if $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) = 1$. If the DevOps engineer can assign a weight of importance, $w_i \in [0,1]$, to the utility of measurement $\theta_i(t)$ such that the weights sum to unity, $\sum_i w_i = 1$, it is recommended to take the utility value as the affine combination of the utilities of the metric values, $U(\boldsymbol{c}^*(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) = \sum_{i=1}^{m} w_i \cdot u_i(\theta_i(t_k))$.

Note that $\sum_{i=1}^{m} w_i \cdot u_i(\theta_i(t_k))$ is the standard arithmetic mean if all the weights are equal, $w_i = 1/m$. The utility function can be calculated as

$$U(\boldsymbol{c}(t_{k+h}) \mid \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}) = \left[\sum_{i=1}^{m} w_i \cdot u_i(\theta_i(t_k))\right] + \sum_{i=1}^{m} \frac{du_i}{d\theta_i}|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))}[f\theta_i(t_{k+h}) - \theta_i(t_k)]$$

5.2.3 Function shapes creation

We propose to use three most popular function shapes from Chang [36], *i.e.*, a piecewise linear utility function [37], a constant function, and the 'S-shaped' function [38] that can be used in Algorithm presented in Section 5.2.2. The latter will be represented as the differentiable sigmoid function. On top of that, we propose to use two functions that we call the U-shaped utility function and reversed U-shaped utility function. To calculate the change in the utility, there is a need to know the derivatives of $u(\theta)$, and we also provide these derivatives for the proposed template functions.

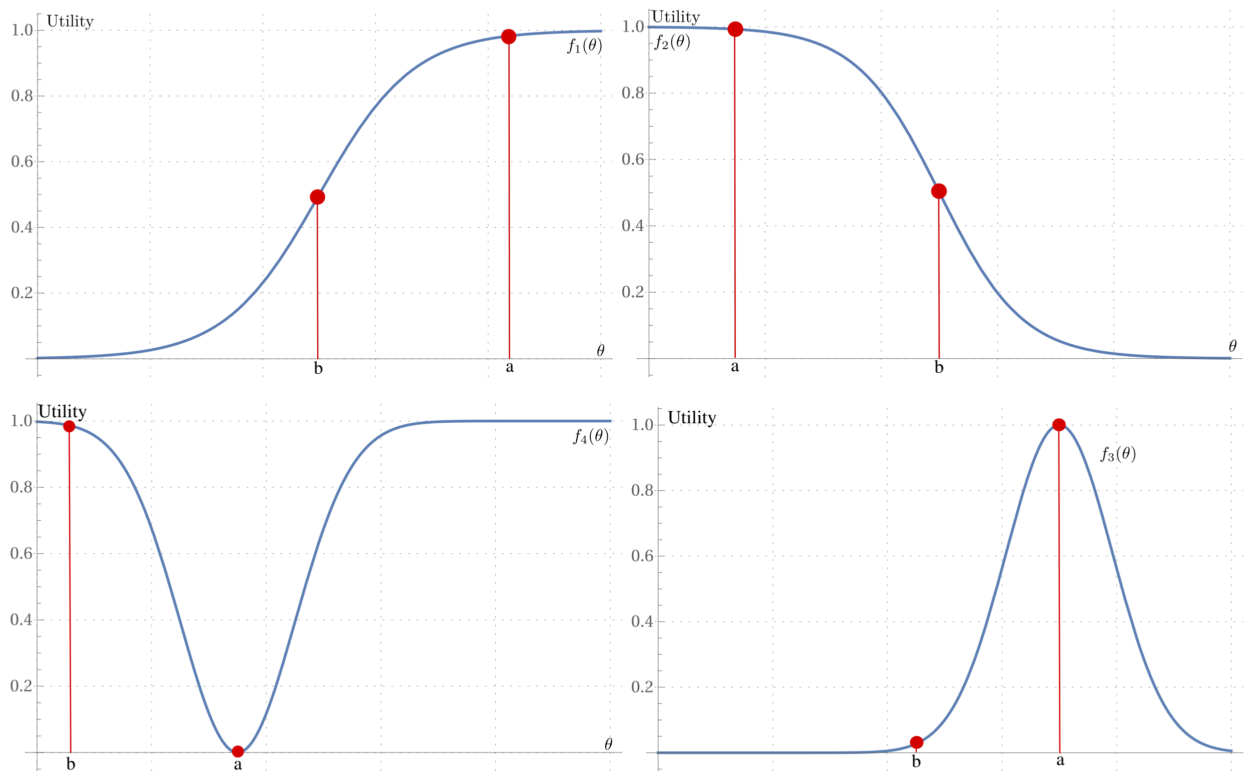


Figure 5 - Template S-shaped , reversed S-shaped, reversed U-shaped and U-shaped functions

5.2.3.1 S-shaped sigmoid function

The sigmoid function f_1 can be seen as a similar function to the standard 'S-shaped' utility function, which is one of the most popular shapes of utility functions [38]. It is just modified to be differentiable. This function has a shape presented on Figure 5 where θ is the chosen metric: $f_1(\theta) = 1 - \left[1 + e^{\phi_2(\theta-\phi_1)/\phi_1}\right]^{-1}$. When choosing this function, there is a need to indicate two points: a where the utility value is close to 1 and $b < a$ where the value should be 0.5: $f_1(a) = 1 - \epsilon$ and $f_1(b) = 0.5$. Given these two points: a and b, it is possible to calculate the function form and adjust the ϕ_1, ϕ_2 parameters by solving the following set of equations:

$$1 - \left[1 + e^{\phi_2(a-\phi_1)/\phi_1}\right]^{-1} = 1 - \varepsilon$$

and $1 - \left[1 + e^{\phi_2(b-\phi_1)/\phi_1}\right]^{-1} = 0.5$

which yields: $\phi_1 = b$ and $\phi_2 = \ln\left((1-\epsilon)/\epsilon\right) \cdot b/(a-b)$.

The overall function has a form:

$$f_1(\theta \mid \mathrm{a}, \mathrm{b}) = 1 - \left[1 + e^{\left((\theta-b)\ln[(1-\epsilon)/\epsilon]\right)/(a-b)}\right]^{-1}$$

while the derivative of $f_1(\theta \mid \mathrm{a}, \mathrm{b})$ is:

$$\frac{\partial f_1}{\partial \theta} = \frac{\ln\left(\frac{1-\epsilon}{\epsilon}\right)\left(\frac{1-\epsilon}{\epsilon}\right)^{(\theta-b)/(a-b)}}{(a-b)\left(\left(\frac{1-\epsilon}{\epsilon}\right)^{(\theta-b)/(a-b)} + 1\right)^2}$$

5.2.3.2 Reversed S-shaped function

As can be seen of Figure 5, it is a very similar function to S-shaped sigmoid function: $f_2(\theta \mid \phi_1, \phi_2) = \left[1 + e^{\phi_2(\theta-\phi_1)/\phi_1}\right]^{-1}$. The only difference is that for bigger value of θ, the function value decreases. When choosing this function, there is a need to indicate two points: a where the utility value is close to 1 and $b > a$ where the value should be 0.5: $f_2(a) = 1 - \epsilon$ and $f_2(b) = 0.5$. Given these two points: a and b, it is possible to calculate the function form and adjust the ϕ_1, ϕ_2 parameters by solving the set of equations:

$$\left[1 + e^{\phi_2(a-\phi_1)/\phi_1}\right]^{-1} = 1 - \varepsilon \text{ and } \left[1 + e^{\phi_2(b-\phi_1)/\phi_1}\right]^{-1} = 0.5$$

which yields $\phi_1 = b$ and $\phi_2 = \ln\left(\epsilon/(1-\epsilon)\right) \cdot b/(a-b)$. Therefore, the overall template function has the

form:

$$f_2(\theta \mid \mathrm{a}, \mathrm{b}) = \left[1 + e^{((\theta-b)\ln[\epsilon/(1-\epsilon)])/(a-b)}\right]^{-1}$$

with the derivative:

$$\frac{\partial f_2}{\partial \theta} = -\frac{\ln\left(\frac{\epsilon}{1-\epsilon}\right)\left(\frac{\epsilon}{1-\epsilon}\right)^{(\theta-b)/(a-b)}}{(a-b)\left(\left(\frac{\epsilon}{1-\epsilon}\right)^{(\theta-b)/(a-b)} + 1\right)^2}$$

5.2.3.3 Reversed U-shaped function

Reversed U-shaped function $f_3(\theta \mid \phi_1, \phi_2)$ is another proposition for the utility function template. The intuition behind it is simple: there is a metric value that the user expects to be achieved, and the more the measured values are distanced from the expected value, the lower the utility is for the application owner. This function has a shape which can be seen on Figure 5, where θ is the chosen metric: $f_3(\theta \mid \phi_1, \phi_2) = e^{-((\theta-\phi_1)^2)/\phi_2}$. Having this function chosen as a template, there is a need to indicate two points: a where the utility value is equal to one, and $b < a$ where the value is close to zero: $f_3(a) = 1$ and $f_3(b) = \varepsilon$.

For values equally distanced from point a, the value will be proportionally lower in both directions. Given these two points, a and b, it is possible to calculate the function form and adjust the ϕ_1, ϕ_2 parameters by solving the set of equations:

$$e^{-\left\{((a-\phi_1)\}^2\right)/\phi_2} = 1 \text{ and } e^{-\left\{((b-\phi_1)\}^2\right)/\phi_2} = \epsilon$$

which yields $\phi_1 = a$ and $\phi_2 = -(b-a)^2/\ln\epsilon$

The overall template function has the form:

$$f_3(\theta \mid \mathrm{a}, \mathrm{b}) = e^{\ln\epsilon(\theta-a)^2/(b-a)^2}$$

with the derivative:

$$\frac{\partial f_3}{\partial \theta} = \frac{2(\theta-a)(\ln\epsilon)\epsilon^{(\theta-a)^2/(b-a)^2}}{(b-a)^2}$$

5.2.3.4 U-shaped function

U-shaped function has a shape presented on Figure 5. It is a fourth new proposition for the utility function template. This function has the form: $f_4(\theta \mid \phi_1, \phi_2) = 1 - e^{-((\theta-\phi_1)^2)/\phi_2}$.

Having this function chosen as a template, there is a need to indicate two points: a where the utility value is equal to zero, and $b < a$ where the value is close to one: $f_4(a) = 0$ and $f_4(b) = 1 - \epsilon$. Given these two points: a and b, it is possible to calculate the function form and adjust the ϕ_1, ϕ_2 parameters by solving the set of equations:

$$1 - e^{-((a-\phi_1)^2)/\phi_2} = 0$$

and $1 - e^{-((b-\phi_1)^2)/\phi_2} = 1 - \epsilon$

that yield: $\phi_1 = a$ and $\phi_2 = (b-a)^2/\ln\epsilon$.

The overall template function has a form:

$$f_4(\theta \mid \mathrm{a}, \mathrm{b}) = 1 - e^{-\ln\epsilon(\theta-a)^2/(b-a)^2}$$

with a derivative:

$$\frac{\partial f_4}{\partial\theta} = \frac{2(\theta-a)(\ln\epsilon)\epsilon^{-(\theta-a)^2/(b-a)^2}}{(b-a)^2}$$

5.2.3.5 *Constant shaped function*

The constant function is a very simple function that can be used to indicate that the measured metric θ is not important for the user in terms of the utility. Such metrics could be useful to indicate the execution context, but not to evaluate the application's utility. The good example of such metric is the number of requests that are coming to the application in one minute. This value does not depend on the parameters of the deployment configuration, but it can be used to indicate the context for the optimization. The shape of the function is flat in a as utility value. If the user chooses this function as a template, there is a need to indicate one value: $a \in [0,1]$, which is achieved independently on the θ value. Therefore, the overall template function has a form: $f_5(\theta) = a$ with the derivative equal to zero.

5.2.4 Secure document storage example

The marginal metric utility function approach is illustrated with an example based on the software application to store documents in a secure way introduced in Horn and Skrzypek [6] and used by Horn and Rozanska [12].

5.2.4.1 *Description*

The software uses a two-level encryption, and its owner has the deployment goal to provide the best possible performance to the users. The performance is measured in terms of the average response time to the users, but the deployment should be done at the minimal price per served request. To maximize the utility of the user, a cloud optimization platform has two main decision variables: c_1, which is the number of instances, and c_2, which is the number of cores available at each instance. It is assumed that one document decoding thread runs on one core only. Hence, the number of 'servers' seen from the user's perspective is equal to the number of cores. Furthermore, let us assume the execution context of the application is defined by the following metrics:

1. $\theta_1(t)$, which is the average response time to the user,
2. $\theta_2(t)$, which is the number of requests that are coming to the application per second, and
3. $\theta_3(t)$, which is the fraction of requests per second that are unacceptably delayed.

The utility functions will be modelled using marginal metric utility function modelling approach. According to this approach, there is no need for direct modelling of the utility function formula. It is enough to specify functions for each $\theta_i(t) \in \boldsymbol{\theta}(t)$ for each dimension; in this example the performance and cost: $u_1(\theta_1(t)) \mapsto [0,1]$, $u_2(\theta_2(t)) \mapsto [0,1]$, and $u_3(\theta_3(t)) \mapsto [0,1]$. Given that, together with optimal configuration $\boldsymbol{c}^*(t_k)$ for time t_k, it is possible to calculate the change in the utility in terms of each dimension and combine it together to calculate the overall utility. There is a need to choose template functions that can be used as u_1, u_2, u_3 and indicate parameters as described in Algorithm 1.

5.2.4.2 Performance utility

Performance utility considers the average response time to the user and for this purpose only the average response time metric θ_1 is used. The marginal metric utility functions for other metrics should be constant.

A function that evaluates the average response time θ_1 should return the highest possible value if the average response time is equal to the best possible value which is $\overline{T}_S$, *i.e.,* the average response time of directly served requests. It should have the lowest value when the average response time is significantly greater than the response time T^+ that makes the request considered as delayed. These are the two points that the user should define to calculate the shape of function $u_1\big(\theta_1(t)\big)$. The template function f_3 is considered as the most suitable in this case:

$$u_1\big(\theta_1(t)\big) = f_3\big(\theta_1(t)\,|\,a = \overline{T}_S,\ b = T^+\big) = e^{\ln \epsilon \left(\theta_1(t)-\overline{T}_S\right)^2/\left(T^+ - \overline{T}_S\right)^2}$$

The number of requests $\theta_2(t)$ and the fraction of delayed requests $\theta_3(t)$ just give the execution context for the user in terms of the performance utility. It means that it is the objective information that should be considered, but every number of users gives the same performance utility for the user if they are properly served. Therefore, the marginal metrics utility functions can be modelled as constant functions f_5:

$$u_2\big(\theta_2(t)\big) = u_3\big(\theta_3(t)\big) = f_5(\theta_2(t)\,|\,a = 1) = 1\ .$$

The derivatives of all marginal metric utility functions for the performance dimension are:

$$\frac{\partial u_1}{\partial \theta_1}\big(\theta_1(t)\big) = \frac{2\big(\theta_1(t) - \overline{T}_S\big)\epsilon^{\left(\theta_1(t)-\overline{T}_S\right)^2/\left(T^+ - \overline{T}_S\right)^2}}{\big(T^+ - \overline{T}_S\big)}$$

while the derivatives of $u_2(\theta_2), u_3(\theta_3)$ are zero.

Consequently, we define the performance utility as:

$$U_{\mathrm{time}}\big(\boldsymbol{c}(t_{k+h})\,\big|\,\boldsymbol{\theta}(t_{k+h})\big) = u_1\big(\theta_1(t_k)\big) + \frac{\partial u_1}{\partial \theta_1}|(\theta_1(t_k)\ [\mathrm{f}\theta_1(t_{k+h}) - \theta_1(t_k)]$$

where the first part of the sum is $u_1\big(\theta_1(t_k)\big)$ with the assumption that weights w_2, w_3 are zero.

5.2.4.3 Cost utility

Cost utility should express the desire of the application's owner to minimize the cost for the served user. In this cost utility function, only two metrics are involved: $\theta_2(t)$, which is the number of requests that are coming to a server in one second, and $\theta_3(t)$, which is the fraction of requests that are delayed per second. It is possible to assume that the marginal metric utility function for the average response time $u_1\big(\theta_1(t)\big)$ in cost dimension is constant:

$$u_1\big(\theta_1(t)\big) = f_5(\theta_1(t)\,|\,a = 1) = 1.$$

A function that evaluates the number of users θ_2 gives the execution context for the user. Every number of users gives the same utility for the user if they are properly served. Therefore, the value can also be modelled as constant function f_5: $u_2\big(\theta_2(t)\big) = f_5(\theta_2(t)\,|\,a = 1) = 1$.

The evaluation of the fraction of delayed requests $\theta_3(t)$ should reflect the fact that there is some fraction R, which is expected, while any value above R^+ is not acceptable, because the profit from serving users is significantly lower. It is possible to use template function f_2:

$$u_3\big(\theta_3(t)\big) = f_2(\theta_3(t)\,|\,a = R,\ b = R^+) = \left[1 + e^{\left((\theta_3(t)-R^+)\ln[\epsilon/(1-\epsilon)]\right)/(R-R^+)}\right]^{-1.}$$

The derivatives of all marginal metric utility functions for cost dimension are zero for $u_1\big(\theta_1(t)\big)$ and $u_2\big(\theta_2(t)\big)$ and for $u_3\big(\theta_3(t)\big)$:

$$\frac{\partial u_3}{\partial \theta_3}(\theta_3(t)) = -\frac{\ln\left(\frac{\epsilon}{1-\epsilon}\right)\left(\frac{\epsilon}{1-\epsilon}\right)^{(\theta_3(t)-R^+)/(R-R^+)}}{(R-R^+)\left(\left(\frac{\epsilon}{1-\epsilon}\right)^{(\theta_3(t)-R^+)/(R-R^+)}+1\right)^2}$$

We define the cost utility can be defined as:

$$U_{\text{price}}\big(\boldsymbol{c}(t_{k+h})\,|\,\boldsymbol{\theta}(t_{k+h})\big) = u_3\big(\theta_3(t_k)\big) + \frac{\partial u_3}{\partial \theta_3}\big(\theta_3(t_k)\big)[\theta_3(t_{k+h}) - \theta_3(t_k)]$$

5.2.4.4 Evaluation

We present the evaluation of the marginal metric utility function approach. For evaluation purposes, we provide a numerical example based on the Queuing framework. We compare this approach with classical direct utility function modelling, which was used, among others, in the MELODIC project. The comparison is based on the utility function value derived from using two approaches and its accuracy in representing the changes in the execution context and the optimality of the deployed configuration.

Since the user requests enter a First-In-First-Out queue it is natural to represent the application as an open queuing system with one or several servers. The number of requests will certainly vary over the day, but for the sake of exposition here it will be understood that the daily variation can be split into periods with an approximately constant average level of requests per second, $\overline{\theta_2}$. Without further knowledge of a real application, one may model the number of requests per second as Poisson distributed with the given average, $\theta_2(t_{k+h}) \sim Poisson(\overline{\theta_2})$. Interpreting this as a counting process for the request arrivals, the time between two requests will be a random time exponentially distributed with the expected number of requests per seconds as parameter, $t_{k+1} - t_k \sim Exp(\overline{\theta_2})$. Similarly, it is reasonable to expect that the average time to serve one request measured in seconds can be estimated as $\overline{T}_S$, and that the time taken to serve one request is again exponentially distributed with the server capacity measured in requests per second, $Exp\big(1/\overline{T}_S\big)$.

With the above considerations the application can be modelled as an $M/M/s$ queue where s is the number of servers, allowing the reuse of some standard results for this queue model [39]. One may assume that the average requests per second, $\overline{\theta_2}$, will stay constant long enough for the queue to be considered to be in a *stationary* state. The expected utilization of each server is obviously the expected arrivals divided by the expected server handling taking into account that there are s servers available to handle the requests,

$$\rho = \frac{\overline{\theta_2}}{s/\overline{T}_S} = \frac{\overline{\theta_2}\,\overline{T}_S}{s} < 1$$

where the upper limit is necessary to ensure that the servers eventually are able to catch up with the requests, *i.e.,* the queue length will be finite. This implies that $s > \overline{\theta_2}\overline{T}_S$, which can be used to find the minimum number of servers needed. Furthermore, the average response time to the user is the expected waiting time in the queue plus the expected service time,

$$\overline{\theta_1} = \frac{(s\rho)^s p_0}{s!s(1/\overline{T}_S)(1-\rho)^2} + \overline{T}_S$$

where the probability of an empty queue is given by $p_0 = \left[\frac{(s\rho)^s}{s!(1-\rho)} + \sum_{n=0}^{s-1}\frac{(s\rho)^n}{n!}\right]^{-1}$.

A request that is not serviced by time threshold, T^+ will eventually be flagged as *delayed*. The time budget for a decoding can be split into two parts, $T^+ = T_Q^+ + T_S^+$ where T_Q^+ is the upper time limit allowed for queueing and T_S^+ is the maximum time allowed for the server decoding the document. The probability that the request is queued longer than T_Q^+ is given by

$$Pr\big[T_Q > T_Q^+\big] = 1 - Pr\big[T_Q \le T_Q^+\big] = \frac{(s\rho)^s p_0}{s!\,(1-\rho)}e^{-s(1-\rho)T_Q^+/\overline{T}_S}$$

and the server delay probability is given by the cumulative density function for the exponential distribution $Pr[T_S > T_S^+] = 1 - \Pr[T_S \leq T_S^+] = e^{-T_S^+/\overline{T}_S}$.

Since the service time is independent of how long the request has been queued, the probability of exceeding the delay threshold is the sum of these two probabilities. The fraction of delayed requests, $\theta_3(t)$ equals this joint probability.

A server is a single core that serves requests sequentially. When a server performs well, the average response time is $\overline{T}_S = 89\text{ms} = 89 \cdot 10^{-3}\text{s}$, which means that a single server saturates when the average number of requests per second is $\overline{\theta_2} = 10^3/89 = 11.23$. According to the common knowledge[16], the expected response time to the user should be around $\overline{T} = 0.1\text{s}$, while after 1 seconds a request should be considered *delayed*. Since the document decoding time is independent of the number of servers, the threshold for delayed requests is fully allocated to the queueing delay, $T_Q^+ = 10\text{s}$. The acceptable percentage of delayed requests can be assumed to be 1%, which is still acceptable for the user, while anything more that 70% is definitely not acceptable, so $R = 0.01, R^+ = 0.7$.

Consider the situation at t_1 with moderate load, $\overline{\theta_2} = 100$ requests per second. The minimum number of servers is then 9. However, the average response time $\overline{\theta_1}$ at 0.945s will then be almost 10 times the response time requirement. The optimal number of servers can then be found from setting $\overline{T} = \theta_1(t_1) = 0.1\text{s}$, and solve $\overline{\theta_1}$ for the number of servers, showing that this requirement can be met with $c_1^* = 12$ servers. The performance metric values with increasing load are given in Table 3.

The change in the utility can be expressed locally around the point consisting of the optimal solution $\boldsymbol{c}^*$ found for the set of measurements $\boldsymbol{\theta}(t_k)$ in terms of the measurements $\boldsymbol{\theta}(t_{k+h}) = [\theta_1(t_{k+h}), \theta_2(t_{k+h}), \theta_3(t_{k+h})]^T$ according to the previous theory as

$$U(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\varphi}) \; - \; U(\boldsymbol{c}^*(t_k) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\varphi}) = \sum_{i=1}^{m} \frac{\partial u_i}{\partial \theta_i}\bigg|_{(\boldsymbol{c}^*(t_k), \boldsymbol{\theta}(t_k))} [f\theta_i(t_{k+h}) - \theta_i(t_k)].$$

We calculate the change in the value of utility for the example modelled and using numerical values presented above. Results of all calculations are presented in Table 4.

[16] https://www.dnsstuff.com/response-time-monitoring

Table 3 - The performance metric values with increasing load

Time	Load $\overline{\theta_2}(t)$	Servers c_1^*	Utilization ρ	Average response $\theta_1(t)$	Delayed requests $\theta_3(t)$
t_1	100	12	0.741667	0.0962045	$1.87 \cdot 10^{-16}$
t_2	110	12	0.815833	0.105346	$6.67 \cdot 10^{-12}$
t_3	120	12	0.89	0.130056	$2.20 \cdot 10^{-7}$
t_4	130	12	0.964167	0.267191	$6.87 \cdot 10^{-3}$
t_5	131	12	0.971583	0.320972	0.0192672
t_6	132	12	0.979	0.412911	0.0540442
t_7	133	12	0.986417	0.605511	0.151525
t_8	134	12	0.993833	1.26196	0.424647

We compare our approach with the directly modelled utility function from Section 5.1. We calculate utility function values separately for the performance and for the cost utility dimensions. To calculate the utility in the performance dimension, there is a need to set parameters ϕ_1, ϕ_2 of the beta distribution function presented in Section 5.1.1. The highest possible utility value should be achieved for the expected response time $\overline{T} = 0.1s$. Therefore, we set $\phi_1 = 1.5, \phi_2 = 5$, then $\beta^+(1.5,5) = 2.81665$. We calculate:

$$\upsilon_{\text{time}}\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_1)\big) = \big(\beta(0.0962045\,|\,1.5,5)\big)/2.81665 = 0.994509$$

For the cost dimension, we put parameters $c_1{}^- = 1, c_1{}^+ = 15, P^- = 0.5, P^+ = 10, Price\big(c^*(t_1)\big) = 4 \cdot 12$. We assume that values of $\theta_3(t_{k+h})$ that are below 10^{-10} are zero. Values for $\upsilon_{\text{time}}\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_2)\big), \dots, \upsilon_{\text{time}}\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_8)\big)$ and $\upsilon_{\text{price-user}}\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_2)\big), \dots, \upsilon_{\text{price-user}}\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_8)\big)$ are presented in Table 4. It is possible to calculate the overall utility values calculated using affine combination of both functions. For the simplification of calculations, we assume that weights for both dimensions are equal, $w_1 = w_2 = 0.5$.

To summarize, numerical results clearly show that the approach presented in this deliverable gives promising results that are comparable with results derived from direct utility modelling. The modelled situation started at t_1 with highly evaluated utility values of metrics $\boldsymbol{\theta}(t_1)$, while for t_8 the measured metric values indicated that the utility of the application had significantly decreased and the drop in the utility of the optimal configuration for t_1 can be noticed in both approaches.

Even though the values of utility look similar for both approaches, it is important to notice that it is not necessary to calculate the difference between $U\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_{k+h})\big)$ and $\upsilon\big(\boldsymbol{c}^*(t_1)\,\big|\,\boldsymbol{\theta}(t_{k+h})\big)$. The direction of the change in the utility should be evaluated and should correctly indicate the need of an adaptation of the application configuration to a new configuration that is optimal for the current execution context. The proposed metric based utility is as good as the more complex direct utility modelling for this purpose.

Table 4 - The utility function values for time and price dimensions for direct utility function modelling and marginal metric utility function modelling

Time	Load	U_{time}	U_{price}	υ_{time}	$\upsilon_{\text{user-price}}$
t_1	100	0.9971276	0.9906382	0.994509	0.99683322777
t_2	110	0.9898405245	0.990638	0.999220	0.99712112488
t_3	120	0.970152832	0.99063798641	0.992572	0.99736103721
t_4	130	0.86089052075	0.99021369574	0.716318	0.99754702257
t_5	131	0.818040509	0.98944802112	0.578783	0.997534644
t_6	132	0.744478811075	0.98730012752	0.366839	0.9974624445
t_7	133	0.59133406075	0.9812795281	0.0905592	0.997189298
t_8	134	0.06830832	0.96441099445	0.0254211	0.99683322777

The utility value for the changed context $\boldsymbol{\theta}(t_{k+h})$ stay in the standard unit interval in the presented example. However, this property is not assured in general by our modelling approach. By definition, the linearisation method works in the neighbourhood of $c^*(t_1)$. A change in the utility that makes the new utility out of the interval $[0,1]$ is a strong indication that a new optimal configuration must be computed because the current configuration is no longer optimal. This is a feature that is not clearly defined when using direct utility function modelling, and an out-of-bound utility value can be used as a trigger for optimization. Moreover, the biggest advantage of this modelling is the fact that it can be extended to support this type of utility function also in the reasoning process. Hence, it is possible to make the reconfiguration decisions without having an explicit utility function formula. This work will be proceeded in the coming months and reported in Deliverable D2.4 *Proactive utility: Algorithms and evaluation*.

5.3 Template Cloud utility function construction

Measurement metrics can be freely aggregated forming new metric values. For instance, a windowed average over the last n samples of a metric is itself a metric whose value is updated every time a new value of the averaged metric is observed. In CAMEL terms, and aggregated metric is called a *composite metric*. Composite metrics can again be used in other composite metrics hierarchically. The metric values involved in the utility function can be simple 'raw' measurement values, or composite metrics. The marginal metric utility function modelling aimed at relating the utility value to the measurement values. However, often one will have a composite metric better expressing a concept to which the DevOps engineer can relate. Consider for instance the concept of 'server performance'. A server can be overloaded if the CPU load is high and at the same time the memory consumption is high, or the network is slow. The server load composite metric is then a functional relation of several other composite metric values. In the example, on may scale vertically to a larger virtual machine if the CPU or memory were the reasons for the observed performance degradation, but if it was network latency it may be better to move the virtual machine closer to a data source. The challenges addressed in this section is to model utility in terms of such composite metric values and then relating this kind of composite metric values used in the utility function to the changes in the configuration.

5.3.1 Utility metric

We use the term *utility metric* to indicate metrics $\boldsymbol{\nu}(t_k) \in \boldsymbol{\theta}(t_k)$ that are not only a part of the *execution context*, but also a part of the utility function. There is a need to predict or estimate what will be the value of the utility metric when the deployment configuration changes. This is the role of the metric estimator in MORPHEMIC called the Performance Module (see Section 6.2.3 and D3.*3* Optimized planning and adaptation approach for more details). It is important to notice that the metric estimator can gather feedback regarding estimated metric value when the proposed configuration is deployed which allows learning the correlation between decision variables, predicted execution context, and future utility metric value.

The standard utility function [12] as a weighted sum combination is called *affine* if the weights $\boldsymbol{w}$ sum to unity: $U(\boldsymbol{c}(t_{k+h}) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{w}) = \sum_{d=1}^{D} w_d u_d(\boldsymbol{c} \,|\, \boldsymbol{\theta}(t))$ where $\sum_{d=1}^{D} w_i = 1$. We propose a different form of a utility function that is able to express the same user preferences which takes into consideration only the performance metrics calculated as a function of decision variables and the execution context:

$$U(\boldsymbol{v}(t_{k+h}) \,|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\theta}(t_k), \boldsymbol{w}) = \sum_{d=1}^{D} w_d u_d\big(v_d(t_{k+h}) \,\big|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\theta}(t_k)\big)$$

where $\sum_{d=1}^{D} w_i = 1$.

Moreover, it should be still possible for the user to provide the utility metric formula directly in CAMEL, which makes these improvements compatible with the MELODIC platform. It makes this approach more flexible and general.

5.3.2 Utility function construction

There is a simple way to retrieve from the user the application optimization preferences. The steps needed to be performed are described as Algorithm 2 and they contain specifying the utility metrics, specifying the shape of the utility function by choosing from the template g for each metric together with a default utility metric formula that will be used for the initial deployment of application and also in case of not sufficient predictions from the metric estimator.

Algorithm 2 Steps for the user to define a utility function

```
   Input: $\boldsymbol{g} = [g_1, g_2, \ldots, g_8]^T$
   Result: the utility function: $U(\boldsymbol{v}(t_{k+h}) \,|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_k), \boldsymbol{w})$
1  $\boldsymbol{\theta}(t)$ ←The user defines metrics that are measured
2  $\boldsymbol{v}(t)$ ← The user defines utility metrics $\boldsymbol{v}(t_k) \in \boldsymbol{\theta}(t_k)$
3  For ( $v_d \in \boldsymbol{v}$ )
4      $j$ ← The user chooses the index of the best $g$
5      $a, b$ ← The user defines $a, b$
6      $v_d(t_{k+h}) = f\big(\boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\theta}(t_k)\big)$ ← The user defines how $v_d$ is calculated
7      $u_d(v_d) = g_d(v_d \,|\, a, b)$
8      $w_d$ ← The user defines the proportional importance of $v_d$ in the overall utility
9  return $U(\boldsymbol{v}(t_{k+h}) \,|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_k), \boldsymbol{w}) = \sum_{d=1}^{D} w_d u_d\big(v_d(t_{k+h}) \,\big|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_k)\big)$
```

As the input to Algorithm 2, we propose eight template function shapes but this list can be easily extended with new template function forms. Four of them have been already described in Section 5.4.3 as f_1, f_2, f_3, f_4, but for the purpose of the utility metric utility function construction, they will take as argument ν instead of the predicted execution context

$metric$ θ. The next four are the linear versions of similar functions such as linear function f_5, f_6, V-shaped function f_7, f_8. Utility metric utility function construction does not require any restriction on the shape of the function, the only restriction relates to the domain of the function values, so all templates can be freely combined, and it is also possible to use non-differentiable functions.

5.3.2.1 S-shaped sigmoid function templates

The S-shaped sigmoid function can have the same formula as defined in Section 5.2.3.1 and Section 5.2.3.2. The only difference is the fact that instead of using metrics from the execution context, only the utility metrics of the application can be the argument. Therefore, we define g_1 and g_2 as:

$$g_1(\nu) = f_1(\theta \mid \mathrm{a,b}) \; = \; 1 - \left[1 + e^{\left((\nu-b)\ln[(1-\epsilon)/\epsilon]\right)/(a-b)}\right]^{-1}$$

$$g_2(\nu) = \; f_2(\theta) = \left[1 + e^{\left((\nu-b)\ln[\epsilon/(1-\epsilon)]\right)/(a-b)}\right]^{-1}$$

5.3.2.2 U-shaped function templates

The U-shaped function templates have the formula presented in Section 5.2.3.3 and 5.2.3.4. However, the argument for the function is changed to be utility metric, not the general metrics and therefore we define them as g_3 and g_4

$$\begin{aligned} g_3(\nu) &= f_3(\theta) = e^{\ln\epsilon(\nu-a)^2/(b-a)^2} \\ g_4(\nu) &= f_4(\theta) = 1 - e^{-\ln\epsilon(\nu-a)^2/(b-a)^2} \end{aligned}$$

5.3.2.3 Linear function templates

The linear function is the simplest possible version of the utility function. It has a form $g_5 = \phi_1\nu + \phi_2$ or for reversed function: $g_6 = -\phi_1\nu + \phi_2$. This function may be used to indicate that the lower or higher the value is, the better for the user. When choosing this function, there is a need to indicate two points: a when the value is zero and b when the value is one for the standard linear utility function, and a with utility value one, b with utility value equal to zero for the reversed linear function. It is important to notice that for all values behind $[a, b]$, the utility value will be constantly equal to zero or one. Given these two points, it is possible to calculate the function form and adjust the ϕ_1, ϕ_2 parameters by solving the following set of equations for g_5:

$\phi_1 a + \phi_2 = 0$ and $\phi_1 b + \phi_2 = 1$ that leads to $\phi_1 = \frac{1}{b-a}$ and $\phi_2 = \frac{-a}{b-a}$. The linear utility function has therefore a form: $g_5 = \frac{1}{b-a}(\nu - a)$.

$$g_5(\nu) = \begin{cases} 0\,, & \nu \le a \\ 1, & \nu \ge b \\ \dfrac{1}{b-a}(\nu - a), & a < \nu < b \end{cases}$$

Analogically, it is possible to solve the set of equations for g_6, where the user specifies the points: a where the utility value is one, and b where the utility is zero: $-\phi_1 a + \phi_2 = 1$ and $-\phi_1 b + \phi_2 = 0$ that leads to $\phi_1 = \frac{1}{b-a}$ and $\phi_2 = \frac{b}{b-a}$.

$$g_6(\nu) = \begin{cases} 1\,, & \nu \le a \\ 0, & \nu \ge b \\ \dfrac{1}{b-a}(b - \nu), & a < \nu < b \end{cases}$$

5.3.2.4 V-shaped function templates

The V-shaped function is the alternative for the U-shaped and it can be used if the user would like to express more strict boundaries for acceptable performance metric values, for instance, for V-shaped function there is only one utility metric value ν where the utility is zero, and for reversed V-shaped, only one value of ν where the utility is one.

By choosing this template of V-shaped function, there is a need to specify three values $a < b < c$: a where the utility is one, b where the utility is zero, and c where the utility is one. For the reversed V-shaped function, the values in points $a < b < c$ is different: for a and c the utility is zero, for b it is one. The shape of the function between values a, b, c can be calculated by solving similar set of equations as for g_5 and g_6. The overall formulas are:

$$g_7(\mathrm{v}) = \begin{cases} 1, & \mathrm{v} \leq a \vee \mathrm{v} \geq c \\ \frac{1}{b-a}(b-\mathrm{v}), & a > \mathrm{v} \leq b \\ \frac{1}{c-b}(\mathrm{v}-b), & b < \mathrm{v} < c \end{cases}$$

$$g_8(\mathrm{v}) = \begin{cases} 0, & \mathrm{v} \leq a \vee \mathrm{v} \geq c \\ \frac{1}{b-a}(\mathrm{v}-a), & a > \mathrm{v} \leq b \\ \frac{1}{c-b}(c-\mathrm{v}), & b < \mathrm{v} < c \end{cases}$$

5.3.3 Utility metric in reasoning

To optimize Cloud application resources, there is a need to know how the change in the application deployment configuration will influence the future metric measurements. In other words, there is a need to know what the correlation between the execution context is, the decision variables, and future measurements. It is a big challenge and a complex problem because there may be no data about the performance of the application running on a specific configuration. For the current utility function modelling used in the MELODIC platform, this correlation has to be specified by the DevOps engineer. In the utility metric approach, this difficulty is a part of the optimization platform responsibility, so the part of the MORPHEMIC optimization loop.

The utility metric utility function $U(\boldsymbol{v}(t_{k+h}) \,|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_k), \boldsymbol{w})$ can be used in the reasoning to guide the decision about the reconfiguration of the Cloud application. The reasoning is done for the given execution context that can be measured $\boldsymbol{\theta}(t_k)$ or predicted for t_{k+h}. A solver is solving the Constraint Problem and each proposed candidate solution $\boldsymbol{c}(t_{k+h})$ can be passed to the metric estimator that will predict utility metric values $\left(v_d(t_{k+h}) \,\middle|\, \boldsymbol{c}(t_{k+h}), \boldsymbol{\theta}(t_k)\right)$. After that, the utility function value for the proposed candidate solution can be calculated and return to the solver.

It is important to notice that the metric estimator (in MORPHEMIC called the Performance module, described in Section 6.2.3) can learn the correlation between the predicted execution context, configuration, and future utility metrics values. It is possible because when the optimization platform makes the decision and performs the reconfiguration, the real measurements of utility metrics are collected, so it is possible to calculate the error between the estimated and real value and continuously gather new knowledge. Therefore, for the initial deployment, the default utility metric formula can be used but the formula calculated by the metric estimator can improve this default formula over time.

Consider, for example, the utility metric that estimates the time required to complete the remaining trainings proportional to the number of trainings left to do, *i.e.,* it is a linear function of remaining tasks. When a training is completed, one has the observation of the event time and the corresponding total time for all the trainings completed. Hence, the relations among the involved metric values can be inferred from historical data from the past time points. A benefit of learning the model is that it can be used also for time points where only some of the metric values are measured. Every time point corresponds to the measurement event for at least one metric value, but it could be only one value measured at the event and the remaining metrics will keep their values until next time points when their values will be measured.

Another approach for the usage of the utility metric in the reasoning process is to create a new proactive solver that will both solve the Constraint Problem and estimate utility metrics values to learn the correlations internally. This idea and the solver are described in D3.3 *Optimized planning and adaptation approach.*

5.3.4 Evaluation

The evaluation of the utility metric concept has proceeded with the data gathered from simulations on the MELODIC use case: Secure Documents and Big Data Genome application. The data is available in the repository OW2 GitLab repository[17]. Note that the same data is being used to train the Forecasting module that will be reported in D2.2 *Implementation of a holistic application monitoring system with QoS prediction capabilities*, and therefore, the detailed description of the dataset will be provided in that deliverable.

The Secure Document application, as it was described in section 5.2.4, uses a two-level encryption, and its owner has the deployment goal to provide the best possible performance to the users. The performance is measured in terms of the average response time to the users, but the deployment should be done at the minimal price per served request. To maximize the utility of the user, a cloud optimization platform has two main decision variables: c_1, which is the number of instances, and c_2, which is the number of cores available at each instance. It is assumed that one document decoding thread runs on one core only. Hence, the number of 'servers' seen from the user's perspective is equal to the number of cores. Furthermore, two parameters were measured:

- $\theta_1(t)$, which is the average response time to the user,
- $\theta_2(t)$, which is the number of requests that are coming to the application per minute.

The simulation parameters of coming users were defined once and repeated for various configurations described in Table 5.

Table 5 - Simulation parameters of coming users

Number of users	**New number of users**	**Reconfiguration**	**Instances**
0 to 10	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	No, 1 instance	2 cores and 8 GB of memory
0 to 40	Random interval of time according to Poisson distribution with lambda 1500 milliseconds and constant delay offset 15000	No, 1 instance	2 cores and 8 GB of memory.
0 to 40	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	No, 1 instance	2 cores and 15 GB of memory
0 to 150	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	No, 1 instance	4 cores and 30 GB

[17] https://gitlab.ow2.org/melodic/time-series-data/-/tree/time-series-experiments/time-series-data

Number of users	**New number of users**	**Reconfiguration**	**Instances**
0 to 10	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	No, 2 instances	2 cores and 8 GB of memory
0 to 150	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	Yes, min number of machines was 1, maximum 20	4 cores and 30 GB of memory
0 to 150	Random interval of time according to Poisson distribution with lambda 3000 milliseconds and constant delay offset 30000	Yes, allowed, min number of machines was 1, maximum 10	4 cores and 30 GB of memory

One can easily notice that the number of requests $\theta_2(t)$ cannot be seen as the optimization goal because it is independent from the application deployment configuration. Therefore, $\theta_2(t)$ should be a part of the execution context and it should be predicted as a simple time series. However, $\theta_1(t)$ is exactly the application owner's goal and it is a performance metric of the application. It is a utility metric and the utility function described in Section 5.1.1 can be easily transformed into the following utility metric function:

$$\upsilon_{\text{time}}\big(\boldsymbol{c}(t_{k+h})\,\big|\,\boldsymbol{\theta}(t_{k+h})\big) = \beta\left(\frac{\theta_1(t_k)}{T^+}\cdot\frac{c^*{}_1(t_k)\cdot c^*{}_2(t_k)}{c_1(t_{k+h})\cdot c_2(t_{k+h})}\,\middle|\,\phi_1,\phi_2\right)/\beta^+(\phi_1,\phi_2)$$

$$\rightarrow\ U\big(\theta_1(t_{k+h})\big|\,\boldsymbol{c}(t_{k+h}),\boldsymbol{\theta}(t_k),\theta_2(t_{k+h})\big) = \beta\left(\frac{\theta_1(t_{k+h})}{T^+}\,\middle|\,\phi_1,\phi_2\right)/\beta^+(\phi_1,\phi_2)$$

In this case, the main challenge is to provide a prediction for $\theta_1(t_{k+h})$ for given $\boldsymbol{c}(t_{k+h}),\boldsymbol{\theta}(t_k),\theta_2(t_{k+h})$. For this initial evaluation, we made predictions for horizon 3 which means 3 time points into the future on data from experiments [18] and compare them with real measured values. The part of the experiment dataset is shown in Table 6.

Table 6 - Initial evaluation of predictions

Time	Number of servers $c_1(t_k)$	Number of requests per minute $\theta_2(t_k)$	Average response time to the user $\theta_1(t_k)$	Predicted number of requests per minute $\theta_2(t_{k+h})$	Proposed number of servers $c_1(t_{k+h})$	Average response time to the user $\theta_1(t_{k+h})$	Result: Predicted average response time to the user $\theta_1(t_{k+h})$	Result: Calculate average response time to the user (Marta)
92	2	106	6	366	2	1072	1017	6

[18] https://gitlab.ow2.org/melodic/time-series-data/-/tree/master/time-series-data/secure-document%E2%80%8B/deployment-reconfiguration-range-1-to-20/2021-02-05%20to%202021-02-05/v1.0%20raw%20data

93	2	143	5	494	12	955	998	0.83
95	2	470	1072	531	20	540	18	107
96	12	565	955	422	20	10	14	573

For the requests prediction 3 models were tested: N-BEATS [40], ETS (exponential smoothing) [41] and SARIMA (Seasonal Autoregressive Integrated Moving Average) [42]. Methods were evaluated using rolling window strategy, grid search hyperparameter optimization and MAE (mean absolute error) metric on validation set. The best model (N-BEATS) requests predictions were then used for average response time to the user.

For average response time to the user the training was performed using TFT [43] model with Optuna hyperparameters optimization. The best set of hyperparameters was chosen basing on MAE metric. Horizon was equal to 3 minutes. Train, validation, test (values 0: 12) were used. Dataset was prepared with the given columns: Average response time to the user (target), number of instances, predicted requests, cores, memory. Final score was reported on the test set. The accuracy measured as MAE:12. The results that can be seen in the last two columns in Table 6, clearly show that the calculated average response time based on manually defined correlation, that the average response time scales linearly, gives worse results than the result derived by the metric estimator.

5.4 Utility function modelling in CAMEL

5.4.1 Background

CAMEL is multi-domain-specific-language (multi-DSL) able to cover the modelling of all relevant domains related to multi-cloud application management. This covers also the requirement and metric domains where CAMEL is able to specify SLOs and utility functions as well as their constituting parts, such as metrics and (metric) variables. CAMEL is described in more details in D1.1 *Data, Cloud Application & Resource Modelling*. Focusing on utility functions, these can be expressed as metric variables with (mathematical) formulas that involve both metrics as well as other metric variables, where such formulas can include any kind of mathematical function. A metric variable can be considered as a dynamic variable, which needs to be computed during the optimization of application deployment. In contrast to metrics, which need to be computed during application execution. Metric variables can be discerned into two main categories: (a) dynamic node candidate variables; (b) utility functions and their parts.

Dynamic node candidate variables map to computations over characteristics of node candidates (like number of cores) with respect to application components or the whole application. They can be single or composite. When they are single, they directly map the value of a specific characteristic of a currently selected node candidate. For instance, the number of cores for the node candidate that has been selected in the current solution for a specific application component. Composite node candidate variables are computed from other node candidate variables by applying a certain mathematical formula. For instance, we might need to compute the minimum over the number of cores across all node candidates for a specific application component.

Utility functions (or their parts) can be considered as composite, dynamic variables which are computed via formulas over other dynamic (metric) variables as well as metrics. They play the most crucial role in application deployment optimization as they enable us to compute the utility for a certain candidate solution (i.e., a solution comprising a particular node candidate per each application component).

In order to raise the level of understanding of the reader, we supply a specific example of a utility function for computing the utility of an application based on cost:

$$U = exp\left(\alpha/(B-P)^{\beta} - \alpha/(B - c_1 * c_2)^{\beta}\right)$$

where α, β, B are constants with B being the user budget, P is the price of the least expensive node candidate (let's assume a single-component application or an application with only one component having to be deployed in the cloud), c_1 is the number of instances for the component (according to the current candidate solution being examined) and c_2 is the price of the selected node candidate.

This utility function would be expressed as a metric variable with the above formula. Such a function does not need to be split into specific function parts but does include some node candidate variables, which need to be separately expressed. These are P, $c_1$and c_2 . P is a composite node candidate variable (with minimum as its formula over the price of each node candidate) while $c_1$and c_2 are single node candidate variables. Thus, as it can be easily seen, CAMEL is able to easily address the modelling of such a utility function.

5.4.2 The Problem

The main issue involved in utility function specification comes with the reliance on metrics, which map to different measurements depending on the current examined configuration/candidate solution. Thus, metrics that depend on the candidate solution. For such metrics, we need to either have a workaround formula for their computation or rely on prediction techniques in order to compute their expected measurement. In other words, we do need to somehow compute the values of these metrics such that a different utility value can be computed for each candidate solution when such a solution differs from the current application configuration. Without this ability, there will be no possibility to actually conduct the utility-based optimization of the application deployment, especially as such metrics are related to major performance aspects of an application.

In order to better comprehend the above issue, we provide an example of a new utility function, which is performance-oriented. The formula of this function is computed as follows:

$$U = 1/\left(1 + exp(\alpha * (CT - T_{max})/T_{max})\right)$$

where α, T_{max} are constants with the second representing a deadline given by the user and CT is the completion time as a metric. As it can be easily seen, when we need to conduct the deployment reasoning for the respective application, we rely on a specific measurement for CT. As there is nothing in the utility function that depends on the node candidates and/or the current candidate solution, the utility value will be always constant. So, there is no possibility to find an alternative application configuration that might be better from the current one. Even worse, the first application configuration will be randomly selected as there will be no measurement for T_{max}initially.

Based on the above situation, there is a need to find a mechanism such that we can differentiate the utility in the presence of metrics in the utility function that depend on the actual application configuration.

5.4.3 The Solution

In the past, PaaSage[19] and MELODIC relied on a workaround where the metric appearance in utility functions was replaced with specific formulas that have clear dependency on the current candidate configuration through the use of metric variables. The main idea was to have a way to direct the evaluation of the utility function towards more powerful configurations when the current performance was not good and the less powerful configurations when the current performance was too good. In this respect, in the previous, performance-oriented utility function, CT could be replaced with $(\theta_1 \cdot \theta_2 / c_1 \cdot c_2) + \theta_3$, where θ_i are metrics (θ_1 represents number of trainings left to do, θ_2 the percentile bound on task execution and θ_3 the elapsed time, as it was described in the Section 5.1.2) while ci are metric variables (c1 represents the number of instances and c2 the number of cores).

The problem with the above workaround is that leads to an imprecise utility function, which can lead to non-optimal solutions with the inevitable negative impact of continuous application reconfiguration to continuously adapt them. Fortunately, MORPHEMIC now supplies the ability to estimate the value of metrics based on the current candidate configuration/solution through the use of prediction techniques. This gives the possibility to utilise such predictions in deployment reasoning for more precisely computing the respective application (deployment) utility. However, as predictions are not always possible, especially when the number of past measurements is insufficient, there is still the need to apply the workaround, mainly in the initial application execution period until a sufficient measurement number is obtained.

Based on the above need, CAMEL has been extended in order to enable the modelling of another category of metric variables which we can call prediction variables. Such metric variables still incorporate the formula attribute in order to

[19] https://paasage.ercim.eu/

have the ability to apply workarounds when the number of past metric measurements is insufficient but also refer to the respective metric that needs to be predicted through being associated with the corresponding metric context (that covers all necessary details for the computation of the metric, such as schedule, window and object context).

As such, by considering the original utility function, we could introduce a metric variable called CTV which incorporates the formula $(\theta_1 \cdot \theta_2 / c_1 \cdot c_2) + \theta_3$ while it also refers to the CT metric. Initially, when the number of past CT measurements is not adequate, the formula will be utilised for the computation of the CTV value for each candidate solution. Then, when such a number is adequate, the prediction of CT will be incorporated during the utility function evaluation as the value of CTV.

5.4.4 Solution Realisation

In order to realize such a solution CAMEL has been slightly updated through one extension and one modification. The extension concerns incorporating into the MetricVariable class an optional property referring to a MetricContext, thus enabling to associate metric variables with metrics whose value can be predicted by the Melodic platform, especially when such values depend on the current application configuration being examined during application deployment reasoning. By enhancing the MetricVariable class, then, it is up to the modeller to decide for a metric variable whether it should utilise no formula (so as to model single node candidate variable), only the formula (so as to model composite node candidate variables and utility functions) or both the formula and the metric context reference (so as to model prediction variables).

The CAMEL modification refers to removing the currentConfiguration attribute from the MetricVariable class as anything that refers to the current application configuration (i.e., the one currently being applied) should be considered as a metric. This was actually an ambiguity in previous versions of CAMEL as it enabled to specify metrics that map to the current application configuration in two ways, via normal metrics and metric variables.

Both CAMEL updates are visualised in the following Figure 6.

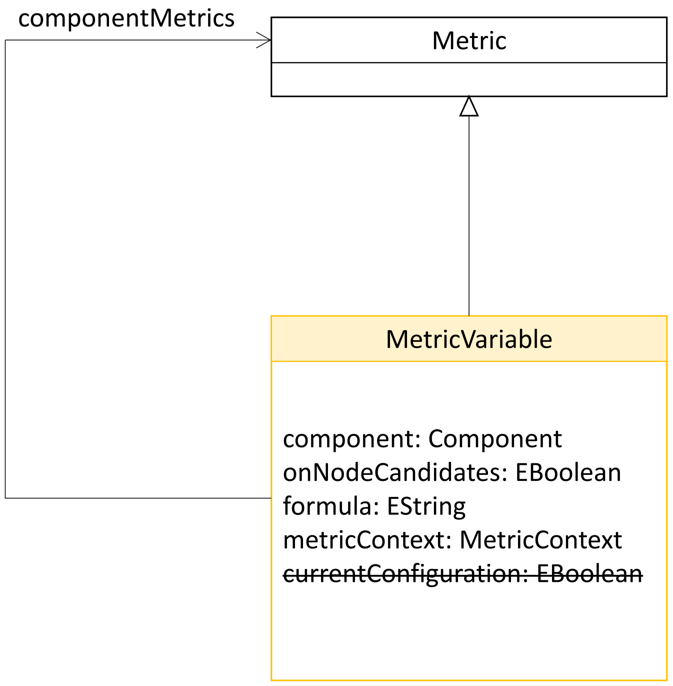
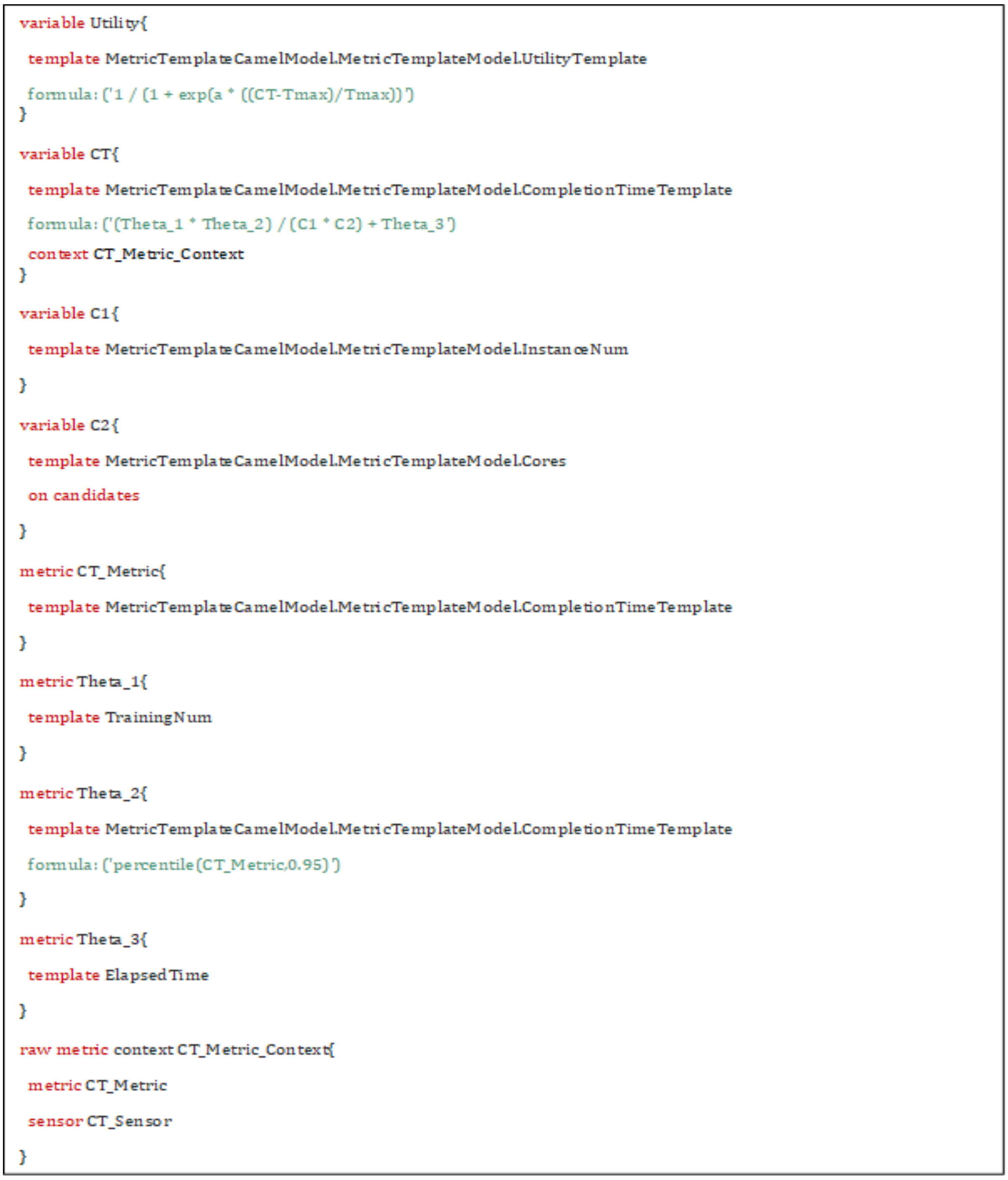

componentMetrics
Metric
MetricVariable
component: Component
onNodeCandidates: EBoolean
formula: EString
metricContext: MetricContext
~~currentConfiguration: EBoolean~~

Figure 6 - The two CAMEL updates on MetricVariable class

We now conclude this section by demonstrating how easily CAMEL can completely express a utility function, the performance-oriented aforementioned one, through the visualization of a small CAMEL specification fragment in CAMEL's textual syntax shown on Figure 7 below.



```
variable Utility{
  template MetricTemplateCamelModel.MetricTemplateModel.UtilityTemplate
  formula: ('1 / (1 + exp(a * ((CT-Tmax)/Tmax))')
}
variable CT{
  template MetricTemplateCamelModel.MetricTemplateModel.CompletionTimeTemplate
  formula: ('(Theta_1 * Theta_2) / (C1 * C2) + Theta_3')
  context CT_Metric_Context
}
variable C1{
  template MetricTemplateCamelModel.MetricTemplateModel.InstanceNum
}
variable C2{
  template MetricTemplateCamelModel.MetricTemplateModel.Cores
  on candidates
}
metric CT_Metric{
  template MetricTemplateCamelModel.MetricTemplateModel.CompletionTimeTemplate
}
metric Theta_1{
  template TrainingNum
}
metric Theta_2{
  template MetricTemplateCamelModel.MetricTemplateModel.CompletionTimeTemplate
  formula: ('percentile(CT_Metric,0.95)')
}
metric Theta_3{
  template ElapsedTime
}
raw metric context CT_Metric_Context{
  metric CT_Metric
  sensor CT_Sensor
}
```

Figure 7 - CAMEL description fragment focusing on expressing a specific utility function

6 Architecture: Proactive adaptation approach

6.1 General component architecture and process flow

The goal of the proactive adaptation feature is to provide the ability to optimize the application's deployment taking into account a future window of time. This means that the application will be optimized for future runtime conditions. This also allows the MORPHEMIC platform to be proactive, and to make the necessary adaptations before any SLO violation occurs, instead of simply reacting to these violations once they occur.

This goal is particularly important from a practical point of view. The reconfiguration of an application takes time, usually a couple of minutes. After that, the runtime requirements may change, which can lead to another reconfiguration. There are some critical applications, for instance, from the medical or networking fields that must run with minimal downtime. Thanks to this feature, the MORPHEMIC platform can proactively adapt application resources to prevent situations that are critical and ensure the expected performance of the application. The proactive adaptation feature is the goal of Work Package 2, and it is realized by the process which includes the following activities: utility function modelling and creation, forecasting of the execution context for the running application and finally, the proactive optimization that leads to a reconfiguration of Cloud application resources. The logical components architecture for proactive adaptation is presented in Figure 8.

The implementation follows the fundamental MORPHEMIC architecture principles that each framework should be separated by interfaces and in case of component reuse, dedicated instances of the component will be created (see D4.1 *Architecture of pre-processor and proactive reconfiguration*). Forecasters and Solvers can be easily plugged in and involved in the adaptation loop. New components are marked in yellow colour, the components that are already present in the MELODIC platform are marked in blue colour. Other artifacts such as the CAMEL Model and the deployed application have a green colour.

It is important to notice that the proactive adaptation can be seen as an extension of the current reasoning process. That is the reason why MELODIC components are being extended to handle predicted metric values in addition to real-time metric values. Further, there are new components that handle the persistent collection of metric data, forecasting, and metric estimation. Another new component that is developed in the scope of this feature is the Utility Function Creator, which produces a utility function formula from high-level utility policies defined by a user, as described in sections 5.3 and 5.4, where this formula can then be incorporated in the CAMEL model of the respective application. The detailed description of both extended MELODIC components and new MORPHEMIC components is provided in section 6.2.

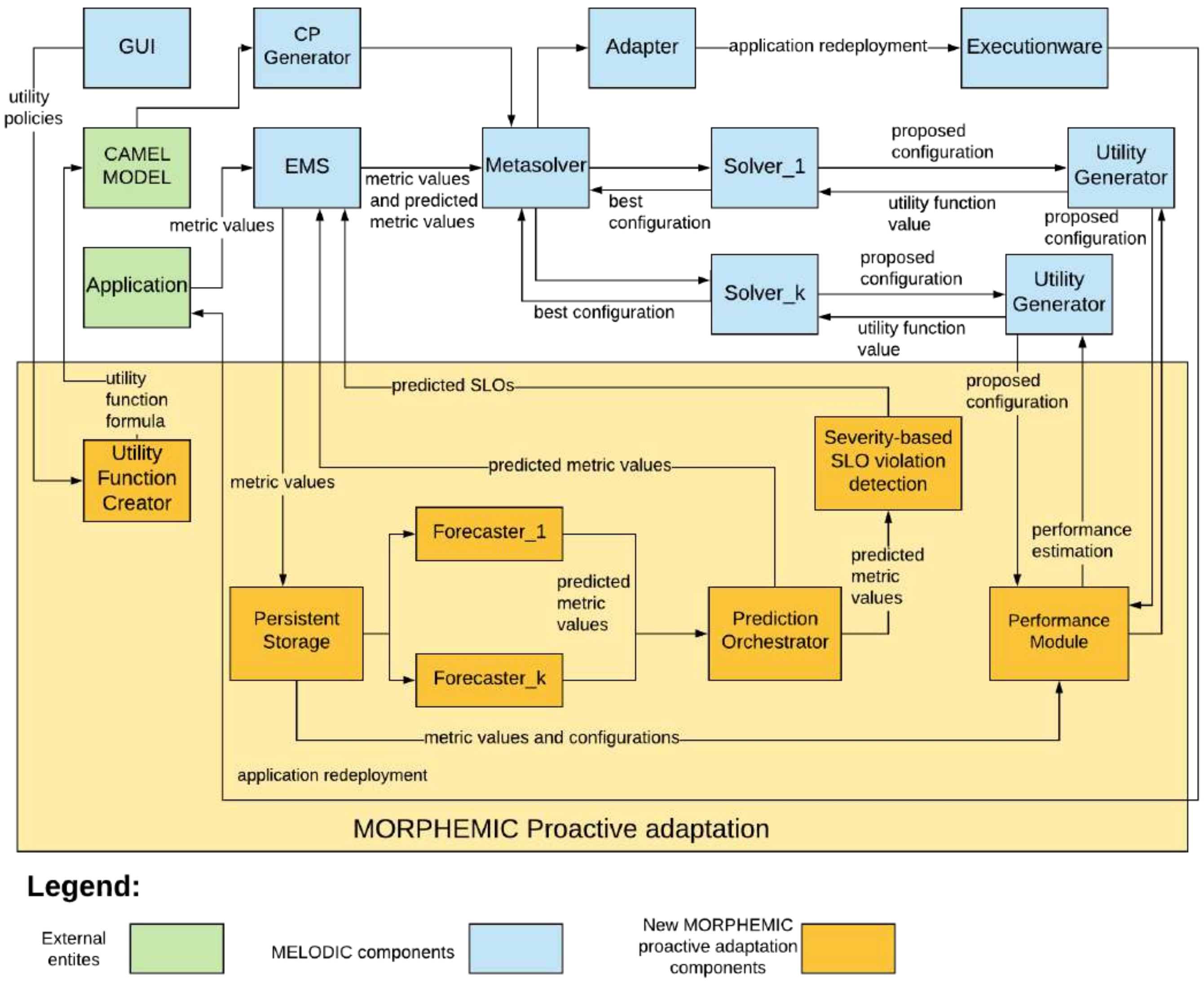


Figure 8 - Proactive adaptation

6.1.1 Utility function creation sequence

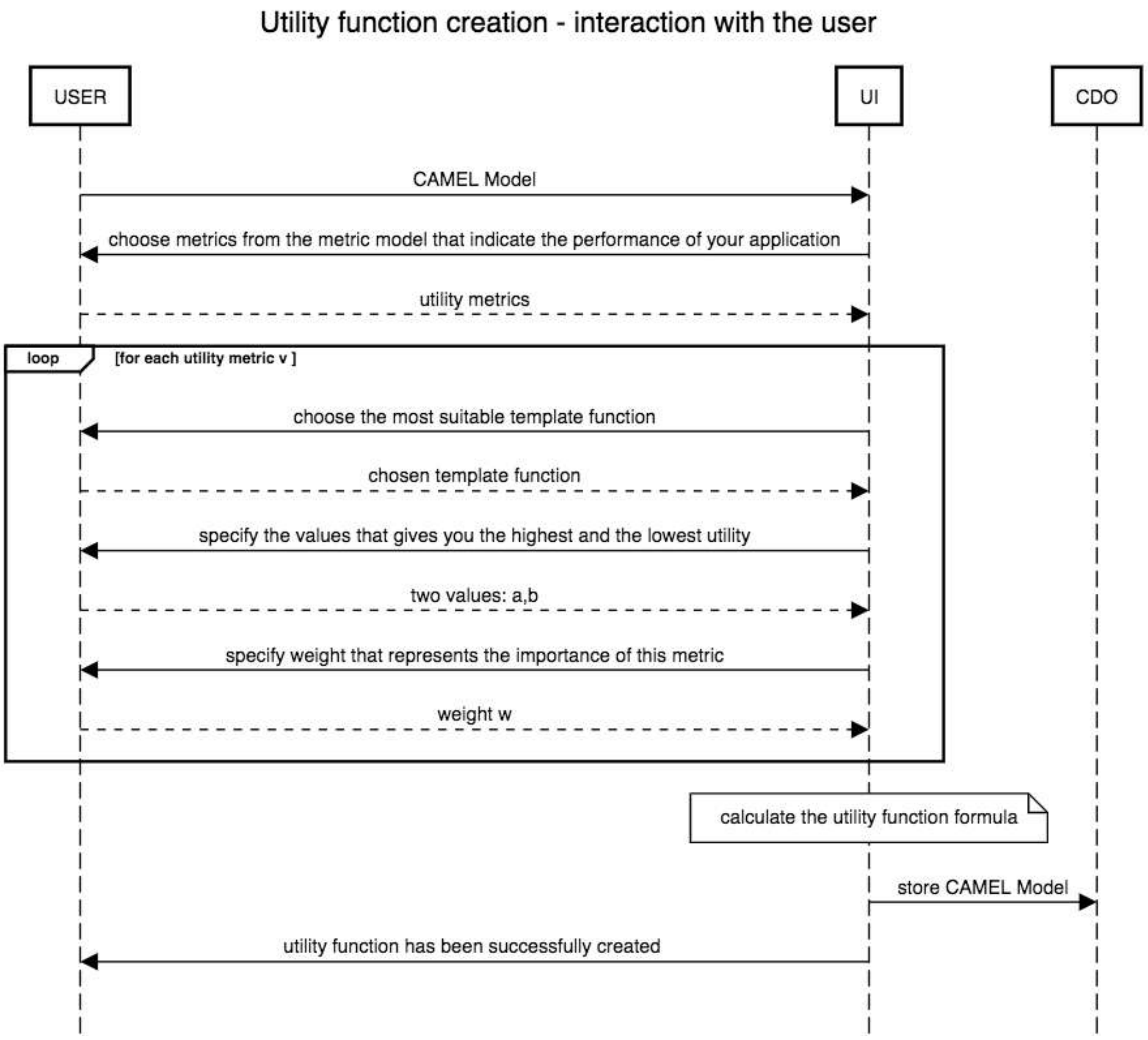

Utility function creation - interaction with the user
USER
UI
CDO
CAMEL Model
choose metrics from the metric model that indicate the performance of your application
utility metrics
loop
[for each utility metric v]
choose the most suitable template function
chosen template function
specify the values that gives you the highest and the lowest utility
two values: a,b
specify weight that represents the importance of this metric
weight w
calculate the utility function formula
store CAMEL Model
utility function has been successfully created

Figure 9 - Sequence of utility function creation

The sequence of utility function creation can be seen on Figure 9 and it is proceeded as follows:

1. The user defines a CAMEL Model for his/her application, without specifying the utility function, using the CAMEL Designer or any textual editor.
2. Then, using the MORPHEMIC GUI, the user chooses metrics that are utility metrics, see detailed description of this concept in sections 5.3 and 5.4.
3. For each utility metric the user specifies:
   a. The shape of the utility function
   b. Two metric values where the utility function value should be equal or close to 1 and where it should be equal or close to 0
   c. Optionally: a default utility metric formula that will be used for the initial deployment and in case of not sufficient predictions; some examples of default formulas can be seen in Section 5.1.

4. Then, the user specifies the weights of each utility metrics. These weights should represent the importance of each utility metric and the sum of weights is equal to 1.
5. After that, Utility Function Creator calculates and creates the overall utility function formula and stores it in the CAMEL Model.

6.1.2 Proactive optimization sequence

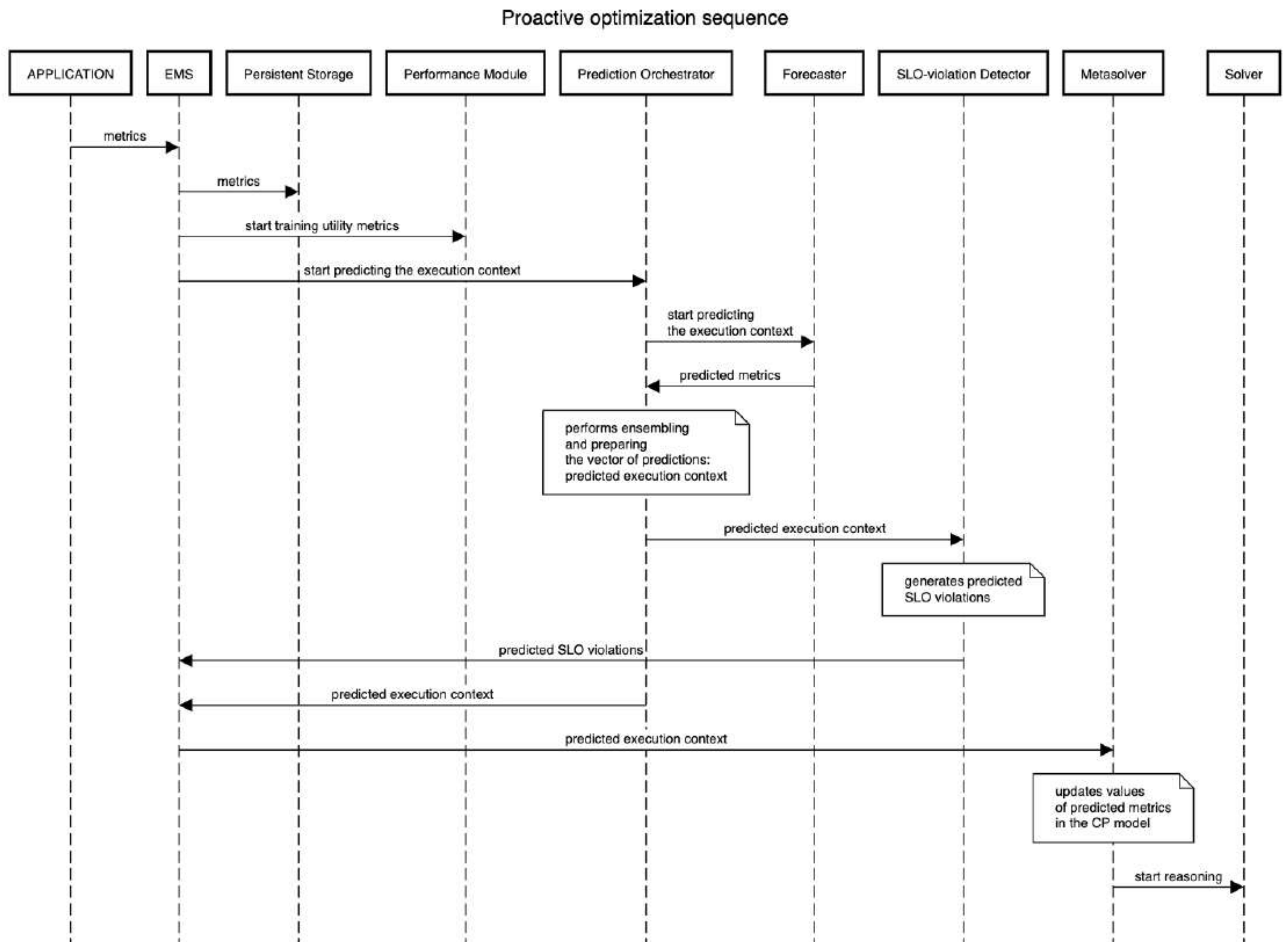

Proactive optimization sequence
APPLICATION
EMS
Persistent Storage
Performance Module
Prediction Orchestrator
Forecaster
SLO-violation Detector
Metasolver
Solver
metrics
metrics
start training utility metrics
start predicting the execution context
start predicting the execution context
predicted metrics
performs ensembling and preparing the vector of predictions: predicted execution context
predicted execution context
generates predicted SLO violations
predicted SLO violations
predicted execution context
predicted execution context
updates values of predicted metrics in the CP model
start reasoning

Figure 10 - Sequence of proactive optimization

The sequence of proactive optimization can be seen on Figure 10 and is proceeded as follows:

1. The CP Generator creates a CP Model, the initial deployment is processed normally (as in MELODIC) with the usage of default metric values and default formulas for utility metrics. To simplify, this step is not seen on Figure.
2. After the successful deployment of the application, measurements are started to be collected.
3. Persistent Storage stores time-series and prepares datasets (different for each Forecasting Method)
4. Once a minimum volume of datasets is available the Prediction Orchestrator is informed and certain or all forecasting methods plus the Performance Module are triggered
5. Prediction Orchestrator ensembles predicted metrics from all Forecasting Methods and publishes orchestrated results as a vector of predictions and also communicates forecasting results to the Severity-based SLO-violation Detector.

6. Severity-based SLO-violation Detector generates predicted SLO violations and publishes it to EMS.
7. Metasolver receives predicted metric values and updates the metric values in the Constraint Problem using predicted metric values and starts the reconfiguration.
8. The reasoning and the rest of the reconfiguration process are proceeded as in MELODIC.

It must be noted that the detailed description of this sequence and of the respective components will be provided in *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities.*

6.1.3 Utility value calculation

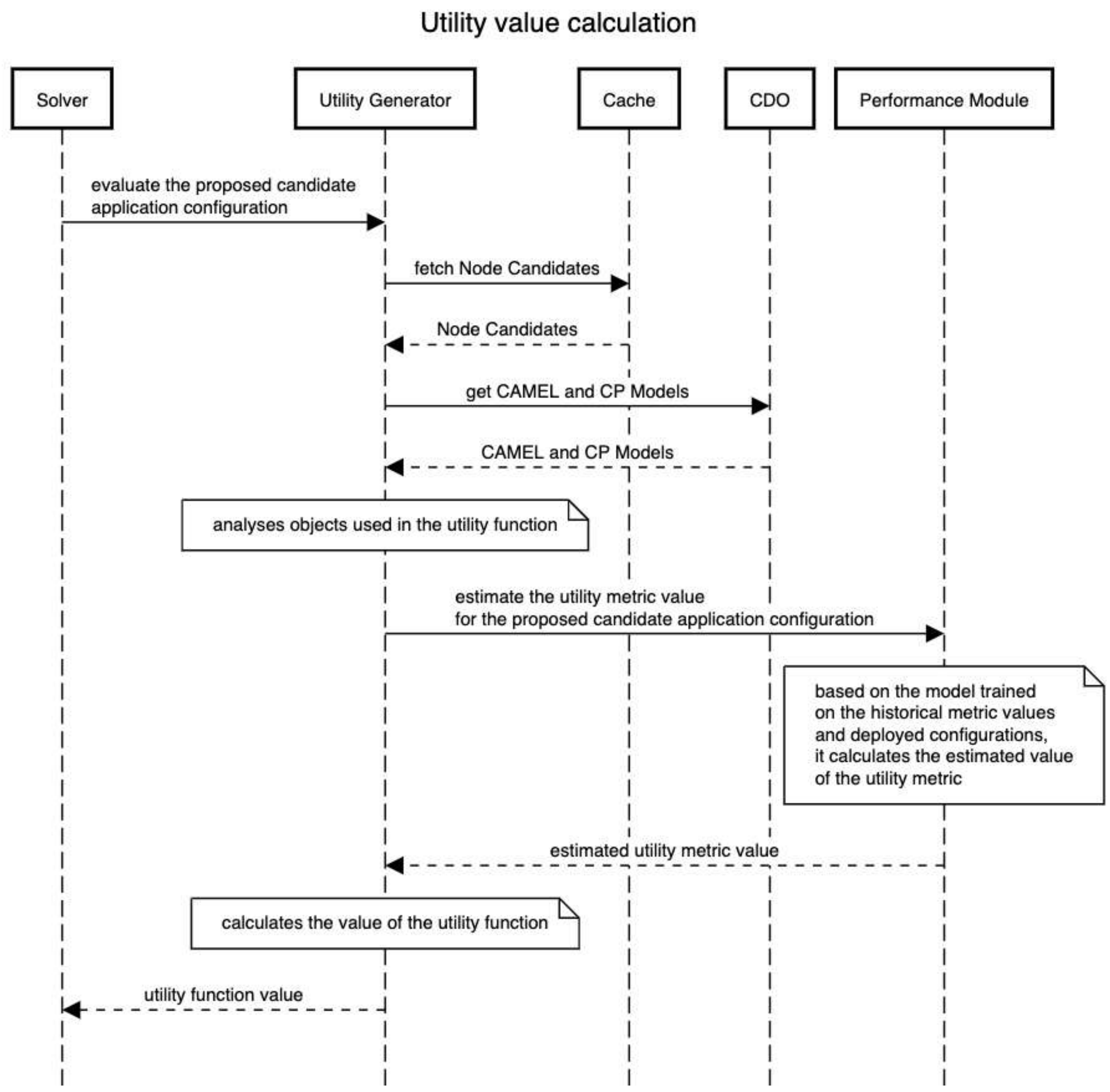

Utility value calculation
Solver
Utility Generator
Cache
CDO
Performance Module
evaluate the proposed candidate application configuration
fetch Node Candidates
Node Candidates
get CAMEL and CP Models
CAMEL and CP Models
analyses objects used in the utility function
estimate the utility metric value for the proposed candidate application configuration
based on the model trained on the historical metric values and deployed configurations, it calculates the estimated value of the utility metric
estimated utility metric value
calculates the value of the utility function
utility function value

Figure 11 - Sequence of the utility function calculation

The sequence of utility function calculation can be seen on Figure 11 and is executed as follows:

1. Solver starts solving using predicted metric values as the execution context.
2. Each proposed candidate application configuration is evaluated by the Utility Generator.
3. The Utility Generator fetches the Node Candidates from Cache and the utility function from the application's CAMEL Model.
4. If the utility function formula does not include any utility metric, the utility function is calculated and returned to the Solver.

5. If the utility function formula includes utility metrics, then for each utility metric the Performance Module is invoked, and steps 6-7 are repeated.
6. The Utility Generator invokes the Performance Module to estimate the utility metric value for the candidate application configuration.
7. The Performance Module fetches the historical metric values and deployed configurations from the Persistent Storage, estimates the utility metric value and then returns the estimated value to the Utility Generator.
8. The Utility Generator calculates the value of the utility function for a given candidate application configuration and returns it to the Solver.
9. Solver compares the value of the utility function for given candidate application configuration to the current best one. If the current candidate application configuration has better utility, then it replaces previous best one.
10. Steps from 2 to 9, that can be seen on Figure 11, are repeated until the best candidate application configuration is found.
11. The solution of the Constraint Problem is passed from the Solver to the Metasolver.

6.2 Key components

The proactive adaptation feature is the goal of Work Package 2 and a part of goal of Work Package 3. It is realized by the processes of utility function modelling and creation, forecasting of the execution context for running application and finally, the proactive optimization that leads to a reconfiguration of Cloud application resources. This section provides the description of all components involved in the Proactive utility framework, where for some components it includes references to other deliverables with a more detailed description.

6.2.1 Utility Function Creator

The Utility Function Creator is a new module; its main role is to create the utility function formula and store it into the application's CAMEL model. The part of this module is integrated with the MORPHEMIC GUI, and it allows the user to specify preferences regarding application optimization visually. This process is described in Section 6.1.1. The internal architecture for this module is presented in Figure 12. It contains a UI component that is integrated with MORPHEMIC UI and that is responsible for the interactions with the user. It also communicates with another UI Component, the Template Shapes Provider, which provides the visualization of template functions. The core component is called Controller; it orchestrates the process of utility function creation by invoking:

- Metric CAMEL analyser – responsible for analysing the CAMEL Model to retrieve the defined metrics
- Weights Calculator – responsible for calculating the weights of each utility dimension. It may use a simple weighted sum method, but it can also be extended to use various methods to retrieve and calculate weights
- Formula Creator – responsible for creating the utility function formula for the given utility metrics, templates, parameters, and weights
- Function Optimizer – responsible for optimizing the parameters of the utility function. It aims for improving the initial utility function created by the user by making it more accurate
- CDO Service – responsible for storing the created utility function formula in the CAMEL Model and then this updated model to the CDO repository

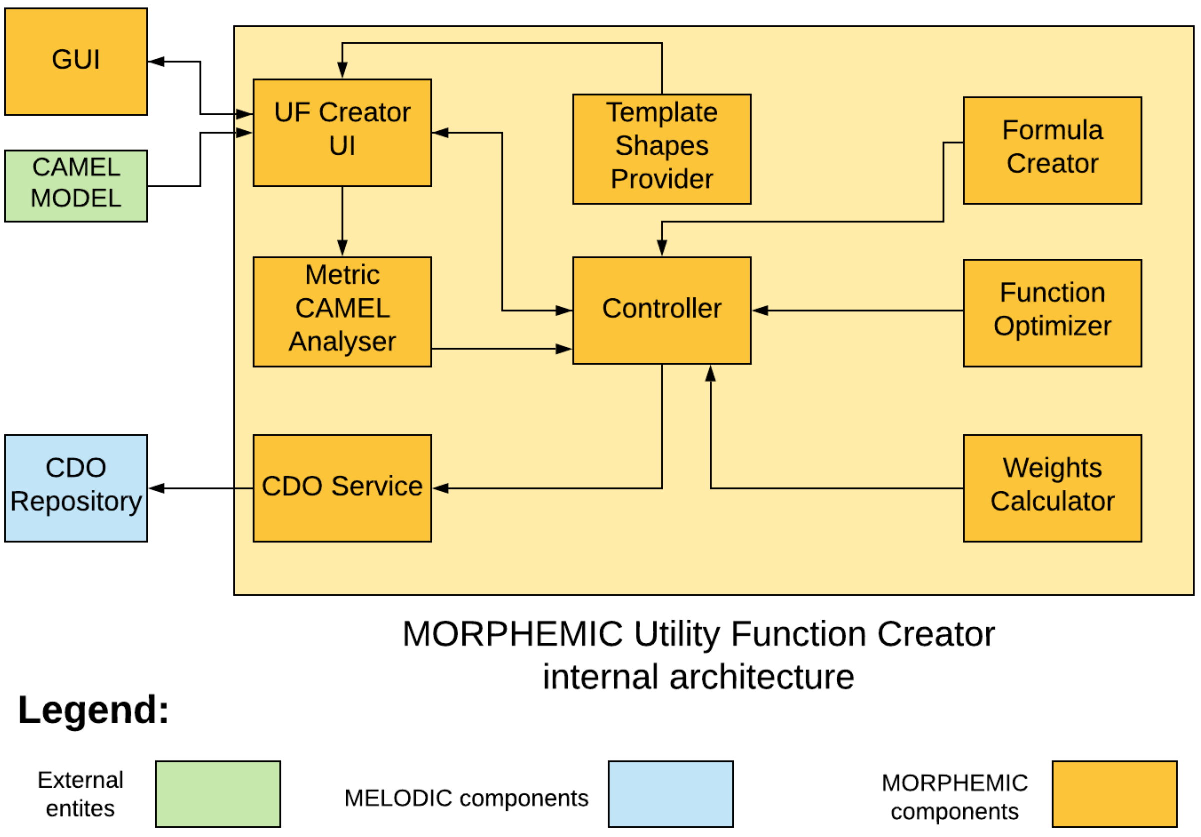


Figure 12 - MORPHEMIC UFC internal architecture

6.2.2 The Utility Generator

The Utility Generator is a MELODIC component that is responsible for calculating the utility function value. It was described in detail in MELODIC Deliverable *D3.5 MELODIC Upperware* [45]. The Utility Generator receives the available Node Candidates offers, the Constraint Problem and utility function, and calculates the utility for the proposed Constraint Problem solution candidate (i.e., the candidate configuration of the application deduced by the Solver during deployment reasoning). For proactive optimization purposes, the Utility Generator will be extended to communicate with the Performance Module to retrieve utility metrics estimations.

6.2.3 Performance Module

Improving the utility value requires the adjustment of variables involved in the utility function such that the application presents a satisfying performance and an acceptable cost. When the costs are determined by the Cloud provider according to the resource allocated to the application, the performance of the application is not only determined by the resource allocated but also, among others, by the workload handled, represented as the execution context in MORPHEMIC. Therefore, optimizing the application proactively consists of exploiting the relationship between the predicted application workload, the resources allocated, and the performance. Two approaches can be used for establishing the relationship referred lately. The first approach can be based on a mathematical model called a utility function as described in Section 5.1. The second approach consists of mimicking the application behaviours by correlating the application workload indicators, the resource allocated to the application (configuration), and the application performance. This concept, described in Section 5.3, is used and implemented in the MORPHEMIC project exploiting machine learning methods. The performance module outputs an application performance metric.

6.2.4 Persistent Storage

6.2.3The persistent storage stores and provides a retrieval mechanism for all metrics (including performance indicators) exposed by managed applications and execution platforms (infrastructure running applications) as shown in Figure 13.

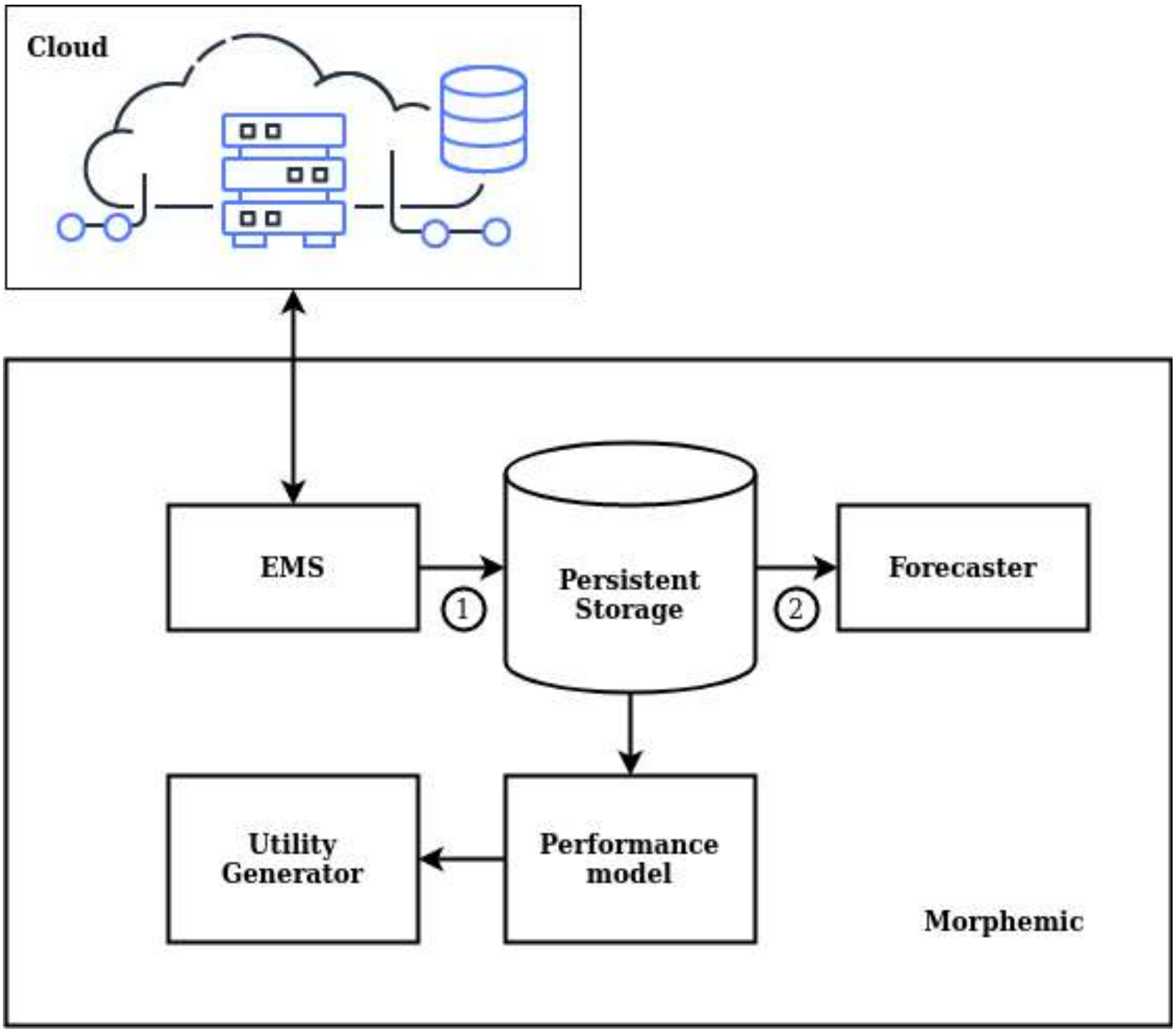


Figure 13 - Persistent storage interaction

As can be seen in the above figure, the persistent storage maintains a connection to EMS for consuming all metrics involved into the utility function and constraints of the managed application. These metrics (1) are transformed and pre-processed for being exposed as a dataset (2). The internal architecture and design details are elaborated in deliverable *D2.1 Design of a self-healing federated event processing management system at the edge*. Metrics stored in the persistent storage can be used for creating datasets for machine learning operations by using an API (Application Programming Interface) implementing all required functionalities. This API known as dataset maker can be imported as a library by all forecasters and other MORPHEMIC's components requiring the creation of datasets.

6.2.5 Forecasters

MORPHEMIC will allow the plugging in of different forecasting methods for issuing useful metrics' predictions that will drive the proactive reconfiguration of applications. Each forecasting method will be introduced in the architecture as a distinct forecaster. These forecasters, as subcomponents of the forecasting module, will handle the publishing of predicted values regarding any monitored metric that the platform will require. The consortium is currently investigating several statistical, machine learning (ML)-based and hybrid time-series forecasting methods (i.e., based on a combination of ML and statistical methods) in order to deliver a minimum set of forecasters as part of the forecasting module. The details of this work will be provided in the upcoming WP2 deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*.

6.2.6 Prediction Orchestrator

The Prediction Orchestrator, created specifically for the MORPHEMIC platform, orchestrates the entire workflow of gathering and processing predicted metric values. Based on the metric's list, the Prediction Orchestrator specifies the details of the prediction process and informs the forecasters to start working. It later continuously gathers values forecasted by Forecasters, and ensembles the values using various machine learning and statistical algorithms. The created vector of predictions is then distributed to further (MORPHEMIC) components. The Prediction Orchestrator will have multiple further functionalities that will allow it to self-improve over time, i.e., it will be able to start or stop different forecasting methods based on their performance. The detailed description of the internal architecture and ensemble algorithms will be provided in the upcoming WP2 deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*.

6.2.7 Severity-based SLO Violation Detector

The Severity-based SLO Violation Detector is a subcomponent of the forecasting module that undertakes the important task of indicating a potential imminent SLO violation. The outcome of this tool will essentially trigger the proactive reconfiguration of the deployed application. This subcomponent will apply techniques that target at the identification of the severity of a potential SLO violation, using as input the predictions published by the MORPHEMIC Forecasters. Aspects such as perceived rate of metric(s) change, probability confidence and distance from SLO threshold will be used for driving the output of this component which will be detailed in the upcoming WP2 deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*.

6.2.8 EMS

Event Management System (EMS) was introduced in the H2020 MELODIC project [46] as a distributed application monitoring system, which is used for monitoring the operation of the deployed cross-cloud applications. It is able to efficiently collect, process and deliver, monitoring information pertaining to a distributed, cross-cloud application, according to CAMEL model specifications, especially considering the defined SLOs. Therefore, EMS is used as a performance monitoring system that aggregates and propagates information that may trigger application reconfigurations. EMS has been significantly enhanced as part of the MORPHEMIC work [47]. It is now called Federated EMS and it has the capability to deploy and maintain a distributed network of self-sustained centralized subnetworks that serve as gateways to different monitoring and event processing agents, which are coupled in different levels across multi-clouds and edge resources. The self-sustainability refers first to the automatic and independent, from a central server, allocation of aggregation and event processing responsibilities to certain nodes of the monitoring network. Secondly, it offers resilience against failures, without the need of manual intervention or orchestration from a centralized network entity. Hence, EMS in the context of MORPHEMIC offers the means to enhance the resilience of the monitoring service across both dispersed cloud and edge resources. Last, we note that the events that EMS processes and propagates enable both the reactive and proactive triggering of the application reconfiguration process and especially its reasoning sub-process part.

6.2.9 Solvers

All, MELODIC and MORPHEMIC solvers, described in detail in *D3.3 Optimized planning and adaptation approach* can solve the Constraint Problem for the current execution context and also for the predicted execution context. In other words, all stateless solvers can make the reasoning as it would be in the future such that they can be used in proactive adaptation. However, the current solvers are not able to consider the fact that the predicted execution context is given with some confidence interval. It should be noted that in proactive adaptation it will be beneficial to use a newly developed solver, described in *D3.3 Optimized planning and adaptation approach*, that considers both predictions, the confidence interval, and real-time measurements during the look up for the best application deployment configuration.

7 Conclusions

This deliverable discussed the ongoing work on an advanced proactive utility framework and the approach that enables the application resources optimization based on forecasted future needs.

We presented the methodology for the application's owner to model the functional form of the utility, including the way of describing this function form in the CAMEL language. Additionally, we provided high-level template function forms that can be incorporated as is in the application's CAMEL model or combined via the use of the Utility Function Creator.

Furthermore, we provided the description of various approaches for utility function modelling that leads to Cloud application resources optimization together with the discussion on fundamental concerns about forecasting in control loops. We designed and initially implemented the proactive adaptation feature of the MORPHEMIC platform which enables modelling the application utility in a proactive way, forecasting the execution context of the application, predicting the future performance metrics of the application, as well as conducting the reasoning based on the forecasted context and predicted performance for the proposed application deployment configuration. It should be noted that for the utility function modelling used in the MELODIC platform, the correlation between proposed configuration, execution context and the future performance of the application has to be specified manually by a DevOps; this is surely a difficult task to perform. In the utility metric approach, such a difficulty is a part of the optimization platform responsibility, so the part of the MORPHEMIC optimization loop.

The next steps of this work involve the implementation and evaluation of the previously discussed and selected approaches for the proactive utility framework based on the data and applications of the MORPHEMIC use case partners. Furthermore, we will investigate additional methods, such as the utility copula and surrogate problem for inferring and modelling the relationships among metric values for proactive utility functions including time series expansion. This work will be proceeded in the coming months and reported in Deliverable D2.4 *Proactive utility: Algorithms and evaluation*.

8 References

[1] Itzhak Gilboa, Rational choice. Cambridge, MA, USA: MIT Press, 2010.

[2] Jeffrey O. Kephart and David M. Chess, "The vision of autonomic computing, "Computer, vol. 36, no. 1,pp. 41–50, 2003,ISSN: 0018-9162.DOI: 10.1109/MC.2003.1160055.

[3] Huang D, He B, Miao C. A Survey of Resource Management in Multi-Tier Web Applications. IEEE Commun Surv Tutorials. 2014;16(3):1574–90.

[4] Lorido-Botran T, Miguel-Alonso J, Lozano JA. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. J Grid Computing. 2014 Dec;12(4):559–92.

[5] Qu C, Calheiros RN, Buyya R. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. ACM Comput Surv. 2018 Sep 6;51(4):1–33.

[6] Geir Horn and Paweł Skrzypek, "MELODIC: Utility-based cross cloud deployment optimisation," in Proceedings of the 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), Conference Location: Krakow, Poland: IEEE Computer Society, May 16, 2018, pp. 360–367.DOI: 10.1109/WAINA.2018.00112

[7] Jeffrey O. Kephart and Rajarshi Das, "Achieving self-management via utility functions," IEEE Internet Computing, vol. 11, no. 1, pp. 40–48, Jan. 2007, ISSN: 1089-7801.DOI: 10.1109/MIC.2007.2.

[8] IBM, "An architectural blueprint for autonomic computing," IBM, 17 Skyline Drive, Hawthorne, NY 10532, U.S.A., White Paper Third Edition, Jun. 2005, p. 34

[9] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjørven, et al. A comprehensive solution for application-level adaptation. Software: Practice and Experience. 2009;39(4):385–422.

[10] Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, et al. A development framework and methodology for self-adapting applications in ubiquitous computing environments. Journal of Systems and Software. 2012 Dec;85(12):2840–59.

[11] Jacqueline Floch, Cristina. Frà, Rolf Fricke, Kurt Geihs, Michael Wagner, Jorge Lorenzo, et al. Playing MUSIC --- building context-aware and self-adaptive mobile applications. Softw Pract Exper. 2013 Mar;43(3):359–88.

[12] Geir Horn, Marta Różańska. Affine Scalarization of Two-Dimensional Utility Using the Pareto Front. In: Proceedings of the IEEE International Conference on Autonomic Computing (ICAC 2019). Conference location: Umeå, Sweden: IEEE; 2019. p. 147–56.

[13] Ralph L. Keeney. Multiplicative Utility Functions. Operations Research. 1974 Feb;22(1):22–34.

[14] E. Jacquet-Lagreze, J. Siskos. Assessing a set of additive utility functions for multicriteria decision-making, the UTA method. European Journal of Operational Research. 1982 Jun;10(2):151–64.

[15] Jian-Bo Yang, Pratyush Sen. Preference modelling by estimating local utility functions for multi-objective optimization. European Journal of Operational Research. 1996 Nov;95(1):115–38.

[16] Silvia Angilella, Salvatore Greco, Fabio Lamantia, Benedetto Matarazzo. Assessing non-additive utility for multicriteria decision aid. European Journal of Operational Research. 2004 Nov;158(3):734–44.

[17] Luciana R. Pedro, Ricardo H. C. Takahashi. Modeling Decision-Maker Preferences through Utility Function Level Sets. In: Ricardo H. C. Takahashi, Kalyanmoy Deb, Elizabeth F. Wanner, Salvatore Greco, editors. Proceedings of the 6th International Conference on Evolutionary Multi-Criterion Optimization (EMO 2011). Conference Location: Ouro Preto, Brazil: Springer; 2011. p. 550–63. (Lecture Notes in Computer Science; vol. 6576).

[18] James E. Matheson and Ali E. Abbas, "Utility transversality: A value-based approach, "Journal of Multi-Criteria Decision Analysis, vol. 13, no. 5-6, pp. 229–238, 2005,ISSN: 1099-1360.DOI: 10.1002/mcda.395.

[19] Ali E. Abbas. Decomposing the Cross Derivatives of a Multi-attribute Utility Function into Risk Attitude and Value. Decision Analysis. 2011 Jun 1;8(2):103–16.

[20] Moawia Alghalith, "New methods of modelling and estimating preferences, „Studies in Economics and Finance, vol. 36, no. 1, pp. 83–88, May 2019,ISSN: 1086-7376.DOI: 10.1108/SEF-12-2017-0354.

[21] Ali E. Abbas and Ronald A. Howard, "Attribute Dominance Utility, "Decision Analysis, vol. 2, no. 4, pp. 185–206, Dec. 2005,ISSN: 1545-8490.DOI: 10.1287/deca.1050.0046.

[22] Roger B. Nelsen. An Introduction to Copulas [Internet]. 2nd ed. New York: Springer-Verlag; 2006 [cited 2021 Apr 20]. (Springer Series in Statistics). Available from: https://www.springer.com/gp/book/9780387286594

[23] Ali E. Abbas, "Multi-attribute Utility Copulas, "Operations Research, vol. 57, no. 6, pp. 1367–1383, Aug.2009,ISSN: 0030-364X.DOI: 10.1287/opre.1080.0687.

[24] Christian Genest and R. Jock Mackay, "Copulesarchim´ediennes et families de lois bidimensionnell esdont les marges sont donn´ees,"Canadian Journal of Statistics, vol. 14, no. 2, pp. 145–159, 1986,ISSN: 1708-945X.DOI: 10.2307/3314660.

[25] Ali E. Abbas, Zhengwei Sun. Multiattribute Utility Functions Satisfying Mutual Preferential Independence. Operations Research. 2015 Mar;63(2):378–93.

[26] Ali E. Abbas, Zhengwei Sun. Archimedean Utility Copulas with Polynomial Generating Functions. Decision Analysis. 2019 Aug;16(3):218–37.

[27] Kritikos K, Pernici B, Plebani P, Cappiello C, Comuzzi M, Benbernou S, et al. A survey on service quality description. ACM Comput Surv. 2013;46(1):1.

[28] Simmons B, Ghanbari H, Litoiu M, Iszlai G. Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In: 7th International Conference on Network and Service Management, CNSM 2011, Paris, France, October 24-28, 2011 [Internet]. IEEE; 2011. p. 1–5. Available from: http://ieeexplore.ieee.org/document/6103960/

[29] Casalicchio E, Silvestri L. Autonomic Management of Cloud-Based Systems: The Service Provider Perspective. In: Gelenbe E, Lent R, editors. Computer and Information Sciences III [Internet]. London: Springer London; 2013 [cited 2021 May 28]. p. 39–47. Available from: http://link.springer.com/10.1007/978-1-4471-4594-3_5

[30] orido-Botran T, Miguel-Alonso, J., Lozano, J.A. Comparison of Auto-scaling Techniques for Cloud Environments. Jornadas de Paralelismo, Servicio de Publicaciones. 2013;(XXIV).

[31] Charles K. Chui and Guanrong Chen, Kalman Filtering: with Real-Time Applications, 5th ed. Springer International Publishing, 2017. doi: 10.1007/978-3-319-47612-4.

[32] Jeffrey O. Kephart and David M. Chess, "The vision of autonomic computing, "Computer, vol. 36, no. 1,pp. 41–50, 2003,ISSN: 0018-9162.DOI: 10.1109/MC.2003.1160055.

[33] Vadim Alimguzhin, Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci, 'Linearizing Discrete-Time Hybrid Systems', IEEE Transactions on Automatic Control, vol. 62, no. 10, pp. 5357–5364, Oct. 2017, doi: 10.1109/TAC.2017.2694559.

[34] Renato Zanetti. "Recursive Update Filtering for Nonlinear Estimation." IEEE Transactions on Automatic Control, vol. 57, no. 6, June 2012, pp. 1481–90, doi:10.1109/TAC.2011.2178334

[35] Xuehai Wang, et al. "The Modified Extended Kalman Filter Based Recursive Estimation for Wiener Nonlinear Systems with Process Noise and Measurement Noise." International Journal of Adaptive Control and Signal Processing, vol. 34, no. 10, 2020, pp. 1321–40, doi:10.1002/acs.3148

[36] Ching-Ter Chang. Multi-choice goal programming with utility functions. European Journal of Operational Research. 2011 Dec;215(2):439–45

[37] Lai Y-J, Hwang C-L. Fuzzy Multiple Objective Decision Making. In: Lai Y-J, Hwang C-L, editors. Fuzzy Multiple Objective Decision Making: Methods and Applications [Internet]. Berlin, Heidelberg: Springer; 1994 [cited 2021 Apr 6]. p. 139–262. (Lecture Notes in Economics and Mathematical Systems). Available from: https://doi.org/10.1007/978-3-642-57949-3_3

[38] LiCalzi M, Sorato A. The Pearson system of utility functions. European Journal of Operational Research. 2006 Jul 16;172(2):560–73.

[39] U. Narayan Bhat ,An Introduction to Queueing Theory: Modelling and Analysis in Applications, 2nd ed.,ser. Statistics for Industry and Technology. BirkhauserBasel, 2015,ISBN: 978-0-8176-8420-4.DOI: 10.1007/978-0-8176-8421-1.

[40] Oreshkin BN, Carpov D, Chapados N, Bengio Y. N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. arXiv:190510437 [cs, stat] [Internet]. 2020 Feb 20 [cited 2021 Sep 14]; Available from: http://arxiv.org/abs/1905.10437

[41] Brown, Robert G. (1956). Exponential Smoothing for Predicting Demand. Cambridge, Massachusetts: Arthur D. Little Inc. p. 15.

[42] Forecasting: Principles and Practice (2nd ed) [Internet]. [cited 2021 Sep 6]. Available from: https://Otexts.com/fpp2/

[43] Lim B, Arık SÖ, Loeff N, Pfister T. Temporal Fusion Transformers for interpretable multi-horizon time series forecasting. International Journal of Forecasting [Internet]. 2021 Jun 16

[44] Claude E. Shannon, 'Communication in the Presence of Noise', Proceedings of the Institute of Radio Engineers, vol. 37, no. 1, pp. 10–21, Jan. 1949, doi: 10.1109/JRPROC.1949.232969.

[45] Yiannis Verginadis, Ioannis Patiniotakis, Vasilis Stefanidis, Fotis Paraskevopoulos, Evagelia Anagnostopoulou, Kyriakos Kritikos, Paweł Skrzypek, Marcin Prusiński, Marta Różańska, D3.5 MELODIC Upper Ware

[46] Y. Verginadis, C. Chalaris, I. Patiniotakis, V. Stefanidis, F. Paraskevopoulos, E. Psarra, B. Magoutas, E. Bothos, E. Anagnostopoulou, K. Kritikos, P. Skrzypek, M. Prusiński, M. Różańska (2019) D3.4: Workload optimisation recommendation and adaptation enactment. H2020 Melodic deliverable.

[47] Verginadis Y., Patiniotakis I., Tsagkaropoulos A., Totow J.-D., Raikos A., Mentzas G., Apostolou D., Tzormpaki D., Magoutas Ch., Bothos E., Stefanidis V. (2021). D2.1: Design of a self-healing federated event processing management system at the edge. H2020 Morphemic deliverable.

[48] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang, 'Model-driven optimal resource scaling in cloud', Softw Syst Model, vol. 17, no. 2, pp. 509–526, May 2018, doi: 10.1007/s10270-017-0584-y.