D3.3 Optimized planning and adaptation approach

MORPHEMIC

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
D3.3

Date
7 December 2021

Responsible partner
UiO

Editor(s)
Geir Horn, Maunya Doroudi Moghadam

Reviewers
Yiannis Verginadis, Chris Kachris

Distribution
Public

Availability
https://www.morphemic.cloud/

Executive summary

The architecture of the polymorphic application optimisation design and adaptation planning will be presented. The integration with the workflow scheduler and the federated event system is described. The various algorithms for the solvers are discussed, and the state of the art presented for the various promising approaches will be given. The deliverable therefore forms a research programme for the releases to be produced during the last part of the project.

Author(s)

Marta Różańska, Geir Horn, Kyriakos Kritikos, Jean-Didier Totow, Ferath Kherif, Paweł Skrzypek, Maroun Koussaifi, Khalil Labidi, Aleksandra Skwara, Iyad Alshabani, Katarzyna Karnas

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 figures

1 Introduction: MORPHEMIC autonomic computing concepts

It is important to remember that MORPHEMIC is an extension to the autonomic Cloud application management pioneered in MELODIC. This implies that a DevOps engineer is supposed to describe the application in the *Cloud Application Modelling and Execution Language* (CAMEL[1]), and then hand it over to MORPHEMIC for deployment and management throughout the application's lifecycle. Hence the starting point for the optimisation in MORPHEMIC is the application's components, the data sets, how these are connected together, the requirements for executing each component, and the availability of a component as different binary artefacts to be executed on different hardware accelerators like *Graphical Processing Units* (GPUs), *Field Programmable Gate Arrays* (FPGAs), or *Tensor Processing Units* (TPUs).

MELODIC, the enactment layer of MORPHEMIC, deploys a CAMEL model and optimises the Cloud resources used by the application. Cloud resources can be virtual machines, containers, or serverless functions. The application components will be allocated to these resources according to the description in the CAMEL model. An optimization problem is solved to find the best possible allocation of the application's component, and the fundamentals of this problem and two example applications are described in Section 2.

To enact the polymorphism of the application deployment, *e.g.*, deploying a component on different hardware configurations, MORPHEMIC must optimise the CAMEL model *before* it is given to MELODIC for deployment. In other words, MORPHEMIC is an "outer optimisation loop" whereas MELODIC is an "inner optimisation loop". The optimisation loops and the integration with the federated event system are described in Section 3 together with the mathematical solvers and algorithms used for the optimisation.

Simplistically, one may say that MELODIC will allocate one virtual machine per application component. This approach wastes resources for workflow applications where there is a data dependency between components since the virtual machines for components executing late in the workflow will be idling until components earlier in the workflow have completed their execution; and similarly, the virtual machines for the earlier components will be idling while the later components execute. Thus, integrating a workflow scheduler will allow a sequence of dependent components to execute in the same virtual machine and reduce the need for Cloud resources. Ideas for understanding the intra-component dependencies and constructing the workflow and subsequent component grouping are discussed in Section 6. One can further optimise the Cloud resource allocation if one takes into consideration known good deployment 'templates' and maps the workflow onto the resource graphs of these templates ensuring that each Cloud resource has only the minimal capacities needed to execute the set of components it is supposed to host, *i.e.*, tuning the requirements of the virtual machines to host groups of application components. These concepts are described in Section 7.

Finally, it is important to remember that MORPHEMIC is not about only deploying an application first time. It is about managing the application over its lifecycle, and the above optimisation must be done continuously while the application is running. Hence, once MORPHEMIC has a better CAMEL model than the currently running model, it may be that this better model has overlapping Cloud resources with the currently running model. The currently deployed application should therefore not be stopped with its allocated Cloud resources being removed. The running configuration should rather be *adapted* into the new and better configuration proposed by MORPHEMIC. The two-level adaptation process will be described in Section 9.

[1] http://camel-dsl.org/

2 Utility based optimization

2.1 Fundamental concepts

MORPHEMIC will extend the autonomic Cloud application management provided by MELODIC [1]. MELODIC is implementing the reasoning loop of autonomic computing [2]. The architectural blueprint is the fundamental feedback loop to *Monitor*, *Analyse*, *Plan*, and *Execute* using the obtained *knowledge* (MAPE-K) [3]:

- The initially deployed application will be *monitored*, see the deliverable *D2.1 Design of a self-healing federated event processing management system at the edge* for a description of the event monitoring system. Unique to this system is the capability for the user to provide application sensors providing relevant information from the running application about metrics that can only be known by the application. An example is the number of application users. Arguably this may also be inferred from the *Central Processing Unit* (CPU) consumption. However, high CPU computation need not imply many users but equally that a single user has asked the application to do some complicated calculations. Knowing directly the number of users will avoid errors and allow better decisions to be made.
- The monitored data will collectively describe the application's *execution context*. In the sequel, the vector of metric values describing the application's execution context obtained at a given time point t_k is denoted $\boldsymbol{\theta}(t_k)$. This context must be *analysed* to see if it has changed significantly from the context at the time of last application configuration. It is necessary to validate that the application is still running within the *operational constraints* set by the application's DevOps when defining the application CAMEL model where these constraints are called *Service Level Objectives (SLOs)*. All SLOs must be evaluated on every update of the execution context, *i.e.*, whenever a metric involved in the optimisation is observed.
- Whenever it is detected that the current execution context is outside of the allowed variability space defined by the operational constraints, a *planning* operation will follow in two stages:
    1. The allowed variability space will be searched for a better application *configuration* $\boldsymbol{c}^*(t_k)$ optimized for the current execution context.
    2. If there is a better application configuration possible, the adaptation process defining how to reconfigure the running application based on the configuration $\boldsymbol{c}(t_k)$ to the new optimized configuration $\boldsymbol{c}^*(t_k)$. This involves planning the application resources and components to stop or to start, in what order, and how to reconnect these resources making them available to the application.
- Once the reconfiguration plan has been produced it will be *executed* by implementing the requested changes across the affected Cloud providers and connecting the new resources to the running application. The monitoring system will also be reconfigured with the metrics to obtain from new components and resources and start collecting these for future reconfigurations.
- The implementation of this feedback loop in MELODIC was reactive and triggered by detected violations of the operational constraints. The used *knowledge* was restricted to the instantaneous execution context observed at the point of triggering. MORPHEMIC will implement an outer loop to this reactive reconfiguration. In this outer loop the collected monitoring data will be used for forecasting future application context changes allowing the application reconfiguration to be *proactive* and the new resources to be available for the application use when needed without the application needing to wait for the Cloud provider reconfigurations to be executed. Proactive adaptation will be further elaborated in deliverable *D2.3 Proactive utility - Framework and approach*. The second novelty implemented by the outer MORPHEMIC loop is a continuous optimization of the application's architecture and requirements by identifying components that are performance bottlenecks, or identifying novel hardware accelerated deployment options. These optimisations will largely affect the size of the application's variability space and requires an understanding of the correlations between an application's running configuration parameters and its performance related metric values.

2.2 Optimization problem

Cloud applications are per definition *distributed* applications consisting of a set of communicating components or services. The components can be containerized microservices, software modules, stateless functions, or other types of services, even third-party services. The application components come in different *types*, $\mathbb{T} = \{T_1, \ldots, T_\tau\}$. Each

component type $T_i \in \mathbb{T}$ has an attribute vector $\boldsymbol{a}_i$ with $|\boldsymbol{a}_i|$ elements. Examples of application component attributes can be the number of cores assigned to the component, the amount of memory, or the Cloud provider to host instances of this component type. Each attribute $a_{i,j} \in \boldsymbol{a}_i$ has a domain $\mathbb{A}_{i,j}$ defining the possible values for this attribute. For instance, a component type may need a certain minimum number of cores to run and cannot benefit more than some maximum number of cores. Similarly, the number of instances to deploy of a particular component type is also an *attribute* for that component type. Should a component exist in different artefacts, *i.e.*, compiled binaries for various hardware accelerators, one may either chose to model these as different types allowing the instance attribute to be zero for non-used hardware variants, or one may model this as a single attribute with the hardware types as domain elements.

The variability space $\mathbb{V}_i$ for the component type T_i is the Cartesian product space of all the domains of its attributes: $\mathbb{V}_i = \mathbb{A}_{i,1} \times \mathbb{A}_{i,2} \times \cdots \times \mathbb{A}_{i,|\boldsymbol{a}_i|}$. To find the best deployment, the optimal application configuration $\boldsymbol{c}^*(t_k)$, one must assign values to all attributes of all component types from their respective domains. The search space to find the optimal configuration is therefore the Cartesian product space of the variability of all the component types: $\boldsymbol{c}^*(t_k) \in \mathbb{V} = \mathbb{V}_1 \times \mathbb{V}_2 \times \cdots \times \mathbb{V}_\tau$. The benefit of finding the optimal configuration at the type level is that even though the search space $\mathbb{V}$ can be large, it is independent of the number of instances of each component type and the number of VMs offered by the usable Cloud providers. Furthermore, constraints among the component type attributes will reduce the effective search space.

To find the *optimal* application configuration one needs to know what optimality means, and for whom. The classical economic concept to explain the choices made by rational economic actors is to say that they will make the choices that maximises their *utility* [4]. Utility maximisation has also been adopted as one of the key elements for realising autonomic computing systems [5], and this view is also adopted in MELODIC [1]. Therefore, each of the different application configurations represents a certain *utility* to the owner of the application. This utility can be expressed as a generalised function $U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) : \mathbb{V} \mapsto [0,1] \subset \mathbb{R}$ that takes the current configuration $\boldsymbol{c}(t_k)$ as argument given the *application's current execution context, i.e.*, the combined set of monitored metric values $\boldsymbol{\theta}(t_k)$, and a vector of static parameters, $\boldsymbol{\phi}$**.** This function may be a mathematical expression, but it does not have to be a mathematical expression, as past research has shown that it may be extremely hard for DevOps engineers to formulate this expression [6]. Any algorithm can be used to express the utility, *e.g.*, simulation, if it produces a real number in the interval [0, 1] for a given application configuration.

One will normally face a situation where the utility has various dimensions, $U_d(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi})$, where all dimensions are measured in different units. For instance, one may want a deployment that minimises cost, measured in a currency, and minimise the response time experienced by the application users, measured in seconds. Hence, finding the configuration that maximizes the utility is generally a *Multi-Criteria Decision Making (MCDM)* problem, which is normally difficult to solve, and many methods have been proposed in the literature over the years [7]. The classical approach to multi-criteria decision making is based on a kind of *'scalarization'*: the utility vector is mapped to a representative single, scalar utility value, which is then optimized [8]. A commonly used approach is first to normalize the utility value in each dimension to a quantity over the unit interval $[0,1]$ to remove each utility dimension's measurement unit and scale. Then express the utility as a weighted sum of the normalized dimensional components. The weighted sum combination is called *affine* if the weights sum to unity:

$$\mathrm{U}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k),\ \boldsymbol{\phi},\ \boldsymbol{w}) = \sum_{d=1}^{D} w_d\, U_d(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) \tag{1}$$

In the case of two dimensions only, like one often encounter the dimensions 'cost' and 'performance', there is a systematic methodology to derive robust weights for the affine utility scalarization based on explicitly solving optimality in both dimensions [9]. However, no particular functional form is required for the utility function in MORPHEMIC, and finding the optimal configuration $\boldsymbol{c}^*(t_k)$ requires solving a non-linear constraint mathematical programme [10].

$$\mathbf{c}^*(\mathrm{t_k}) = \underset{\boldsymbol{c}(t_k)}{\operatorname{argmax}}\ \mathrm{U}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k),\ \boldsymbol{\phi}) \tag{2}$$

subject to

$$\boldsymbol{g}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) \leq \mathbf{0} \tag{3}$$

$$\boldsymbol{h}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) = \mathbf{0} \tag{4}$$

$$\boldsymbol{c}(t_k) \in \mathbb{F}\big(\boldsymbol{\theta}(t_k)\big) \subseteq \mathbb{V} \tag{5}$$

Here $\mathbb{F}\big(\boldsymbol{\theta}(t_k)\big)$ is the *feasible* subspace of the variability space based on the application's current execution context. It works by restricting the domain of the configuration attributes, and this feasible region is a result of evaluating the other constraints, both the inequality constraints and the equality constraints. Consider, as an example, the attribute variable 'number of cores' a given component can use. This will be confined by the number of cores offered by the virtual machine types currently available for the deployment, and a lower limit of the number of cores needed may result from the necessity to serve the current number of application users. Thus, both constraints on the 'number of cores' are dependent on measurements of the current application context.

Given that this is a standard, non-linear mathematical programme, it is possible to draw on a wide range of solvers. The ones currently under evaluation or development are described below. The major difficulty comes from the fact that many of the component attributes may only take on discrete values, and therefore the problem is *combinatorial* and solving the problem to optimality will therefore in general take exponential time. In MORPHEMIC we will approach this problem in various ways: One may use an *anytime* algorithm which can be stopped when the time to find a solution expires with the best solution until the time out as a result. This solution is obviously optimized, but not the optimal one. Alternatively, one may apply transformations on the application model to reduce the number of attributes and the size of the domains since solving a small combinatorial optimization problem does not take much longer than solving the same problem with polynomial algorithms or approximations. Finally, one may run a stateful solver that will optimize the problem continuously in the background and have an optimized solution available when needed. However, this will typically require learning based solvers, and the solution returned will be the best solution on average given the observed variation in the application's execution context [11].

2.3 Example: Big data farming application ("Genome")

Consider an application from the medical domain where a set of *Machine Learning (ML)* models needs to be trained on complex data sets with a soft deadline $\phi_1 = \mathrm{T}^+$ on completing the training of the full set of models. The training is *data parallel*, and the application may therefore be implemented as a training task scheduler dispatching individual training jobs onto a set of worker cores. This model was first introduced in deliverable *D2.3: Proactive utility: Framework and approach* and repeated here to ensure that this document is self-contained.

2.3.1 Performance utility

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, $\theta_1(t_k) = \theta_1(t_{k-1}) - 1$, will be a change in the application's execution context.

Saving cost means that one should start off with a limited number of cores, then see how the computation progresses, trying to forecast the overall completion time based on the execution times of the already completed tasks. Since the training times $\Delta t_k = t_k - t_{k-1}$ are random variates with unknown distribution, one must estimate the upper $1 - \alpha/2$ 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

$$\frac{\theta_1(t_k) \cdot \theta_2(t_k)}{c_1 \cdot c_2} \tag{6}$$

where the numerator 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 numerator represents the number of available cores for the job as the number of worker virtual machines c_1 times the number of cores per worker virtual machine c_2. To know the predicted overall completion time for all calculations one must add to (6) and 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 worker virtual machines, 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

$$U_T(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) = \frac{1}{1+\exp\left(-\phi_2\left[\phi_1 - \left(\frac{\theta_1(t_k)\cdot\theta_2(t_k)}{c_1\cdot c_2} + \theta_3(t_k)\right)\right]\right)} \tag{7}$$

where $\boldsymbol{\phi}$ 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 $\phi_1 = T^+$.

2.3.2 Cost utility

It is obvious in this example that one could get the fastest training by allocating one task per worker virtual machine. Then the make span of the computation would equal the longest training for a single job. This would also be the costliest deployment, and it is therefore desirable to complete the training with the least possible worker machines. A negative exponential is a good candidate for the cost dimension utility. 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 worker virtual machine possible. Furthermore, it is reasonable to assume that the price of a worker virtual machine is decided by the number of cores it offers. A reasonable cost utility function is therefore

$$U_C(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) = \exp\left(\frac{\phi_5}{[\phi_3 - \phi_4]^{\phi_6}} - \frac{\phi_5}{[\phi_3 - c_1\cdot \mathrm{Price}(c_2)]^{\phi_6}}\right) \tag{8}$$

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 worker virtual machine instances multiplied by the price of each worker virtual machine instance, *i.e.*, the total cost of the deployment configuration.

2.3.3 Total utility

It is clear from the above discussion that maximising the two utility dimensions independently gives a conflicting result where one would like to use many worker virtual machines for the best timeliness utility, and as few worker virtual machines as possible for maximizing the cost utility. It is therefore desirable to have a joint utility function that can balance the two utility dimensions, and if an affine combination (1) is used the total utility for this problem becomes

$$U(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) = w_T U_T(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) + (1-w_T) U_C(\boldsymbol{c} \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) \tag{9}$$

for some weight $0 \le w_T \le 1$ given to the application performance utility.

2.3.4 Constraints

There are two constraints for this problem: The chosen configuration must respect the soft deadline, and the cost of the deployment must not exceed the available budget. The expected completion time is the expected time left given by (6) plus the time already spent, $\theta_3(t_k)$. One may add a tolerance parameter $\phi_7 \ge 0$ to allow the deadline $\phi_1 = T^+$ to be exceeded, giving the timeliness constraint.

$$\frac{\theta_1(t_k)\cdot\theta_2(t_k)}{c_1\cdot c_2} + \theta_3(t_k) \le (1+\phi_7)\phi_1 \tag{10}$$

The budget constraint is easier

$$\phi_3 - c_1\cdot \mathrm{Price}(c_2) \ge 0 \tag{11}$$

These constraints are called SLO in the CAMEL model and they will be evaluated for every new time point t_h when a new monitoring metric vector $\boldsymbol{\theta}(t_h)$ is measured. If the constraints are satisfied for the current execution context, $\boldsymbol{\theta}(t_h)$,

the current application configuration will be kept. However, if either of the constants are violated, the solver of the autonomic application management platform must find the configuration $\boldsymbol{c}^*(t_h)$ that maximizes (9), and then appropriately reconfigure the running application to use the available number of worker virtual machines $c_1^*(t_h)$.

2.4 Example: Brain science application

In recent years, *Magnetic Resonance Imaging (MRI)* has become a powerful solution for studying brain tissue and function, being deployed in both research and clinical settings. As a result, the amount of data collected involving brain imaging has exploded dramatically. Thousands of high-resolution brain scans of patients are acquired every day in hospitals. Brain scans of healthy individuals and patients participating in groups studies are also collected every year by research labs worldwide. A number of new large-scale studies have been launched, a prime example being the UK Biobank, through which unique data are collected on a large sample of individuals from the general population. In order to benefit the most from the information the scanners produce, we must be able to identify relevant features in order to gain a greater understanding of human anatomy and function. As a result, in addition to the data acquisition protocol, the research community has also developed sophisticated tools for pre-processing the data and analysing it. These tools are optimized for various types of MRI data and allow the extraction of information about brain activity, the morphological characteristics of the brain's white and grey matter, and the identification of the white matter fibre bundles connecting the different brain regions. In summary, organizations that conduct clinical research have been collecting a growing amount and variety of data. A parallel phenomenon is occurring in the development of brain disease models, which are becoming increasingly data-driven, as well as the expectations of users.

2.4.1 Brain data processing workflow

Morphemic solves these problems; presented below are the major challenges and solutions for running workflows across different types of tasks and their specific requirements.

With the increasing amount of data and sophisticated tools for pre-processing and analysing neuroimaging data, clinical researchers now face the difficulty of managing increasingly complex workflows. In typical neuroimaging applications, a data set is processed within a short period, usually within five minutes, though in rare cases a process may continue to run for days, even weeks. Even with smaller tasks, the large number of data sets makes the entire process require greater computational resources and more storage capacity for storing the intermediate results.

Figure 1 illustrates a typical neuroimaging workflow. The workflows include a variety of inputs that require different pre-processing steps. The intermediate results are combined to produce the final outputs that provide clinically relevant information about a person's brain. This process is repeated for each participant's dataset, where the number of individual data sets may vary from ten to several thousand.

Users defined constraints such as time and cost must be taken into consideration to determine the most effective way of running each workflow. Morphemic enables to optimally configure the neuroimaging application based on utility functions that are every time tailored to the user's workflow. The choice of the optimal configuration and the corresponding utility function are determined by the real time collection of the following metric values.

- The characteristics of each task in term of CPU, *Random Access Memory (RAM)* needs
- The dependencies between the tasks within the workflow. The *Directly Acyclic Graph (DAG)* indicates the type of processes included in the workflow.

The optimal configuration of the application for a given workflow and the current state of execution of the tasks within this workflow are:

- How many worker virtual machines nodes need to be deployed
  Where each task need to be run

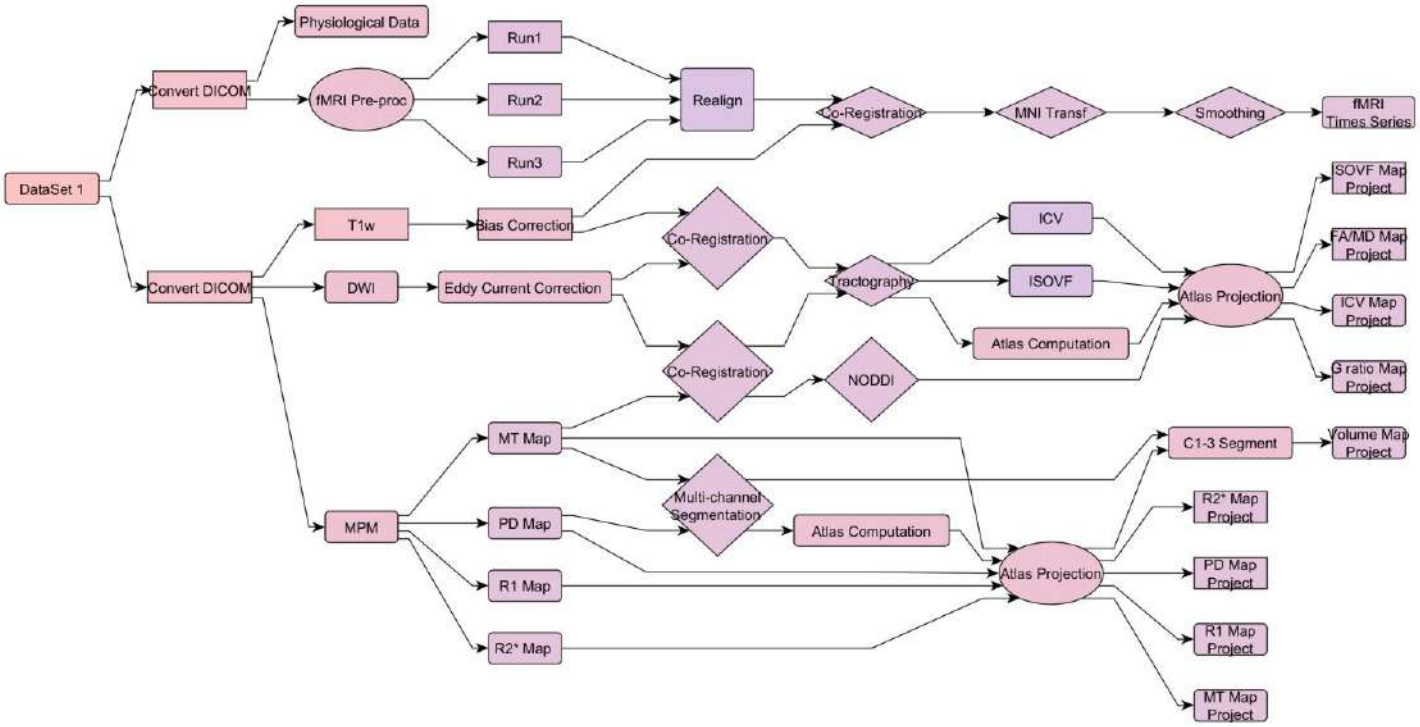

Figure 1 Typical Neuroimaging workflow

2.4.2 Performance utility

A brain data processing workflow represents a set of tasks aimed on pre-processing of different types of neuro images. Execution of some tasks are consecutive, e.g., task rely on input from the previous step, and some can be run in parallel. It is expected that the user can choose to run complete workflow or select a set of tasks, depending on aims and data availability. Typically, the workflow is applied on several subjects that can be run in parallel.

2.4.3 Task grouping

In determining the running time of a task, we have to consider the number of input images, *e.g.*, image conversion will take longer with more raw images provided; task complexity, *e.g.*, diffusion image pre-processing is more complex and time consuming than image segmentation; and amount of RAM available as the less RAM available, the slower the task. We can tune these parameters to optimize the performance and separate the tasks into groups, for example, tasks aimed at different types of images are entirely independent of each other. It is best to run the group of tasks on one VM to minimize data transfer costs. Task running times and their dependencies and whether they are executed sequentially or in parallel will determine the running time of the group.

Figure 2 shows how to optimize the workflow in term of budget, speed and time a grouping approach is proposed, where tasks are assigned optimally to different workers with different resources available so that sub-sequences of tasks are executed on the same worker node if they must be executed sequentially, and on different worker instances if they can be executed in parallel.

The number of subjects' datasets and the total number of tasks are determined at the beginning of the computation. The next question is: how do we decide how many workers and how many nodes to assign from the performance point? Can a Group with lower complexity be run alongside a group with higher complexity (to save RAM)? It is possible that there could be free unused nodes (cost problem) if some parts of the workflow has to wait for other parts to complete? Figure 3 shows an example of task grouping for the “typical neuroimaging workflow” group with low complexity finishes sooner.

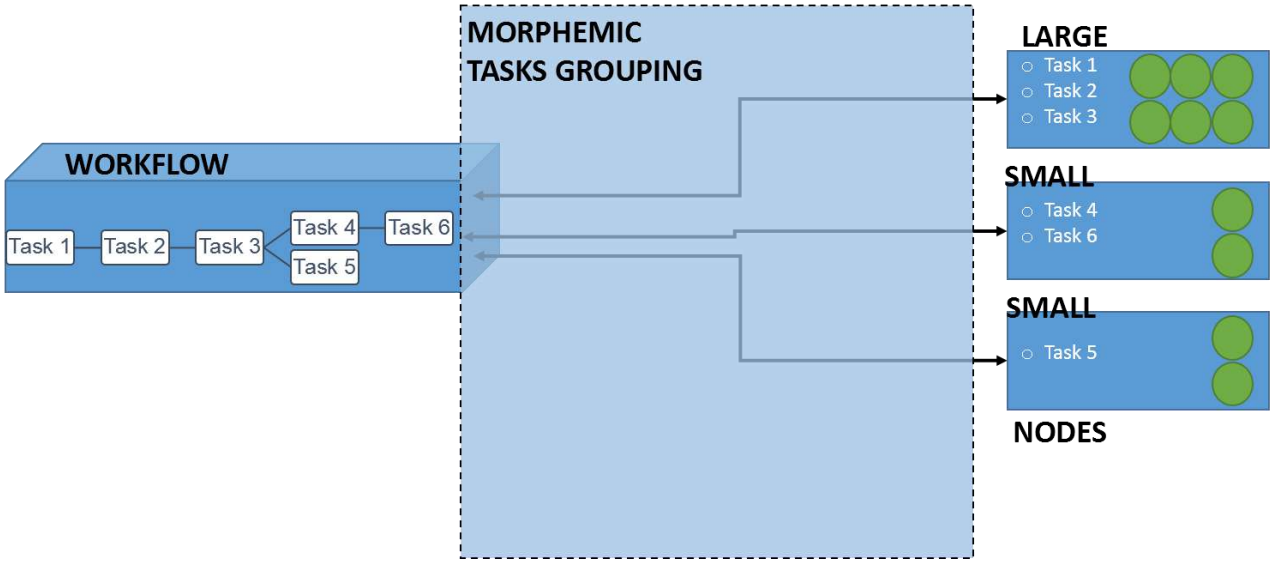

Figure 2 MORPHEMIC forkflow Optimization

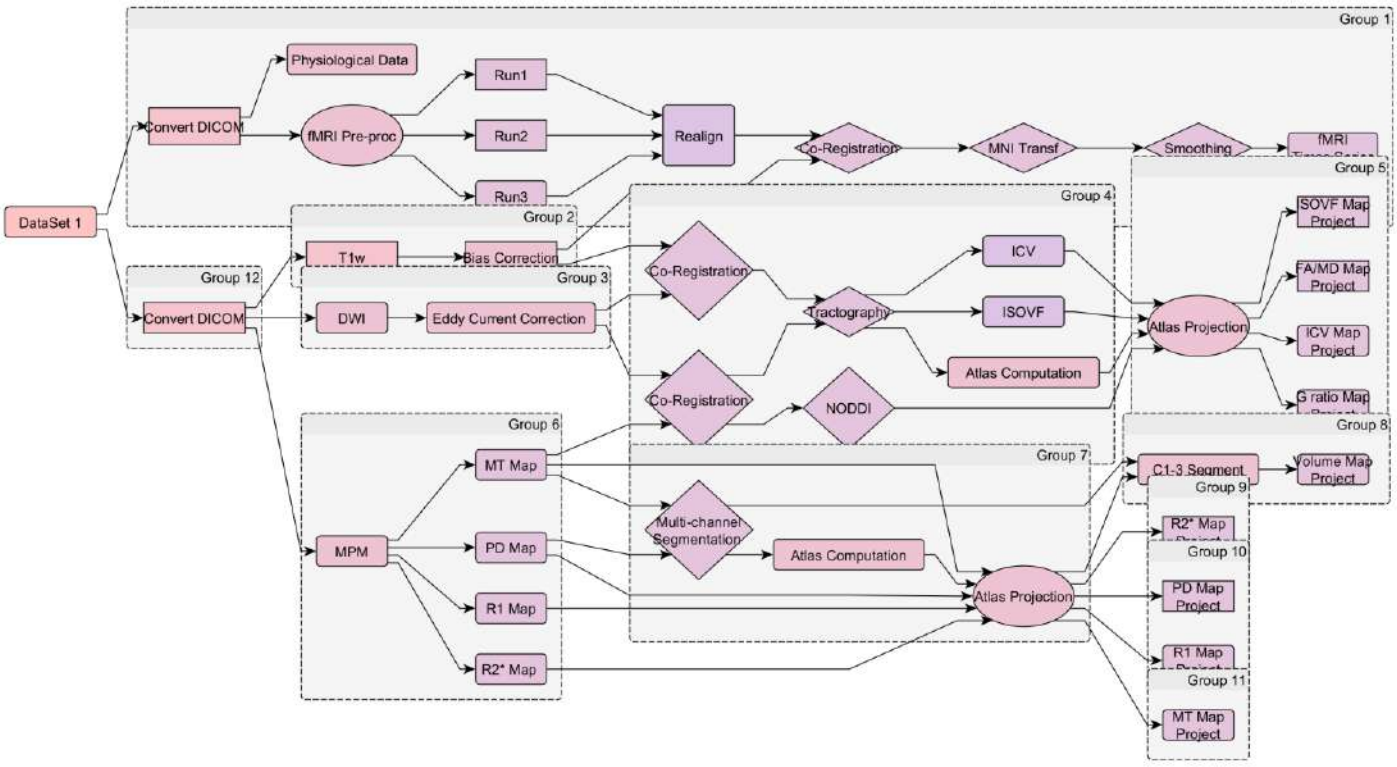

Figure 3 Task grouping for the "typical neuroimaging workflow" of Figure 1

2.4.4 Utility

The utility function for this application is still in development as there are mainly two different approaches for modelling the application's resources and resource scaling:

1. The application can be modelled like the Genome example with a scheduler dispatching over a set of workers. The question is then to scale the number of instances of each worker node type for a timely execution of the whole workflow at the minimal cost.

2. Since the workflow is static, it can be pre-grouped as illustrated in Figure 3. Even though a user can chose to run only a partial workflow, the picture remains the same with some parts deleted. One may take that each task type is a component type in the application and separately scale the parallel sections.

The utility function will most likely have the same two balancing utility dimensions as seen for the Genome example with one *utility dimension* trying to maximize the performance and total makespan of the whole workflow, whereas the other dimension aims at minimizing the deployment cost.

2.4.5 Constraints

Except for the obvious budget constraint for the deployment, the set of constraints for this problem depends on the modelling approach taken and outlined in the previous Section 2.4.4. By having sequential parts in addition to the parallel parts, Amdahl's law gives a bound on the total speed up possible [12]. This should be taken into account when allocating the resources versus cost as it is not necessary to extend the parallelism fully as there will be no additional performance gained with respect to faster computations, and there should definitely be a constraint on the parallelism given by Amdahl's law.

There are no constraints on data location for the use case demonstration in MORPHEMIC as it uses anonymous research data for testing. However, when this application is used on data from real patients, privacy and security issues will force the execution of many of these components to private infrastructures where the data is located. Then there should be constraints for the data locality considering the computational capacity offered by the private infrastructures storing the data. It should be noted that data set components are not included in the above application workflows of Figure 1 and Figure 2.

3 Optimization

3.1 Inner loop: MELODIC optimization architecture

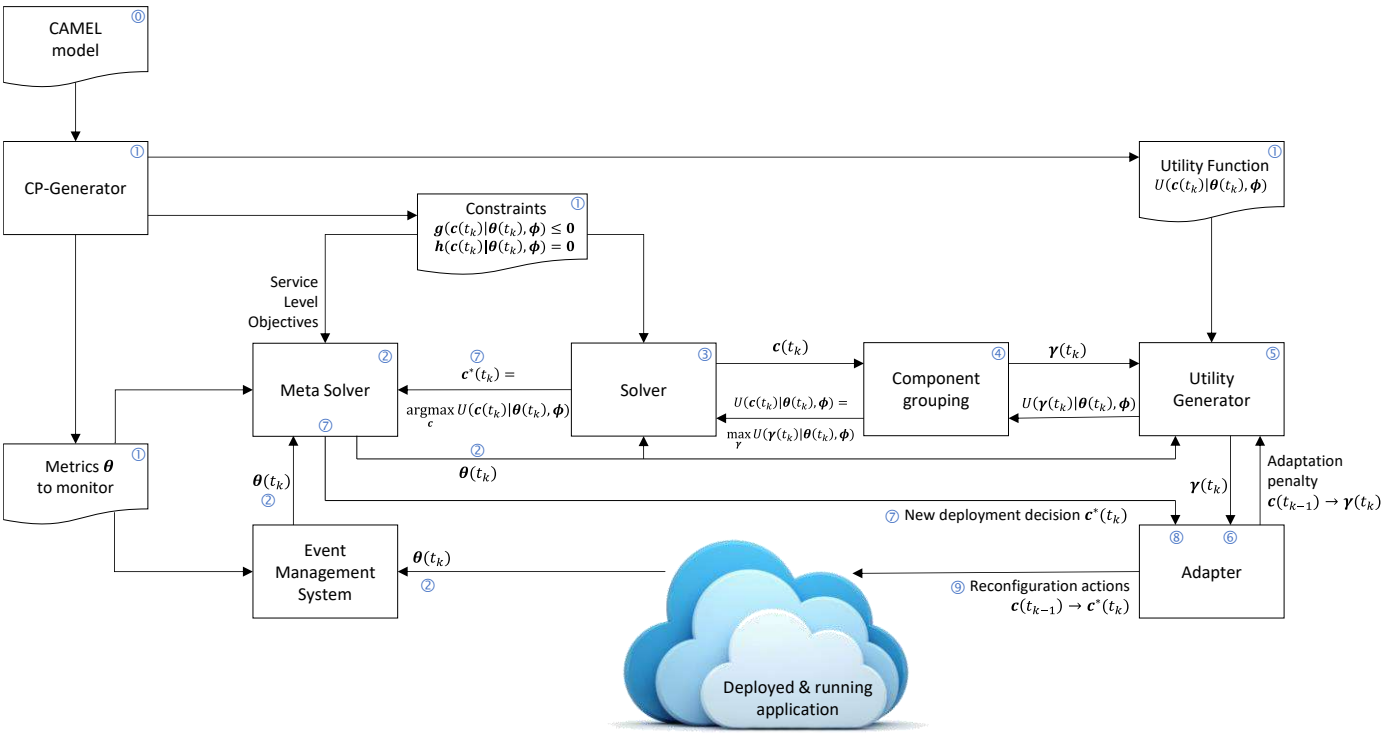

Figure 4 The inner optimization loop of MELODIC

Figure 3 shows the inner optimization loop of MELODIC which reacts to the monitored metrics solving the constraint programming problem for the configuration that maximizes the utility function for the current application execution contexts and reconfigures the application for this better configuration. The inner optimization loop is implemented in the MELODIC platform [1]. The goal of the optimisation process is to find the Cloud application configuration that maximises the utility value. The involved components are shown in Figure 4. The process starts with the DevOps engineer providing the CAMEL model of the application to be managed. This is then translated to a constraint programming (CP) model by the CP-Generator. After the initial application deployment, the monitored metric values defining the application's current execution context are collected by the Event Management System (EMS). Strictly speaking, each new measurement event should produce an improved application configuration. This implies that the constraint programming problem must be solved before the next measurement event will happen. Given that the measurement events are driven by external factors the time point of the next event is not known in advance, and it is likely that there will be too little time to solve the optimization problem over the time interval available up to the next measurement event. The pragmatic solution adopted is to let the Meta Solver compare the measurements representing the application's current execution context with the feasible region of the constraint optimization problem. If any of the constraints, called Service Level Objectives (SLOs), is violated, an improved deployment configuration is needed. This corresponds to the indicated steps 0 – 2 in Figure 4.

The search for the optimized configuration for the current execution context is started by the solver iteratively proposing new feasible configurations. The components of the proposed may then be grouped together as super-components, in various ways, and the utility is calculated for each of the ways to group the configuration proposed by the solver. The utility calculation is done by the Utility Generator considering an adaptation penalty computed by the Adapter as an indication of the complexity of deploying the grouped configuration given the currently running application. The utility will typically be dependent on the actual virtual machines (VMs) selected for the deployment since the VM selection will decide the cost of the deployment. There is a sub-module of the Utility Generator responsible for selecting the cheapest virtual machine among the known *node candidates* satisfying the combined requirements of the component(s) to be hosted by the VM. The utility value assigned to the proposed configuration will be the best utility value for any of its allowed groupings. Based on this utility value, the Solver will update its proposal and the utility evaluation starts for this updated configuration proposal. Eventually, the Solver will converge on the best configuration for the application's current execution context. This reasoning process corresponds to the indicated steps 3 – 6 in Figure 4; with many sub-iterations for the steps 4 – 6 as there may be many groupings possible per proposed application configuration.

The best application configuration found by the Solver in the reasoning process will be returned to the Meta Solver, which will then pass it to the Adapter for actual deployment. The Adapter will calculate the sequence of actions needed in order to transform the currently running application configuration to the new optimized application configuration. Eventually, the adapter requests the changes to be made by the MELODIC Executionware. This corresponds to the steps 7 – 9 in Figure 4. The whole sequence of the optimization process is shown in Figure 5.

3.2 Outer loop: MORPHEMIC optimization

3.2.1 Architecture

The reactive optimization of the MELODIC platform will be extended in MORPHEMIC with a continuous long-term optimization loop that constantly will seek to optimise the *application model* deployed by MELODIC. This architectural optimization can be seen transparently as a *pre-processor* for MELODIC, or as an *outer optimization loop* for MELODIC. The key components of the MORPHEMIC optimization loop are shown in Figure 6 below. Some of these components are identical to the ones found in the MELODIC architecture, and they can even be the same implementation, but it is important to remember that there are separate instances of these components being executed in MORPHEMIC. To illustrate: The MORPHEMIC outer optimization can use the same solver as used by MELODIC, but then there will be two independent instances of that solver running concurrently: one for optimizing inside MELODIC and another to optimize the architecture in MORPHEMIC. The components covered by other deliverables will be briefly presented in the subsections below, whereas the components covered by this deliverable have separated sections below in this deliverable with references given in both cases.

The starting point for the MORPHEMIC optimization loop is the *application CAMEL model*. This has been designed by the DevOps engineer using the CAMEL Designer. This model is first processed by the workflow analyser. The idea

is to identify the temporal data dependencies among the application's component so that if one component has to finish execution before another component can start, this will be noted as a dependency between the two components, and both of the components will have a dependency on the same shared *data set component*. Dependent components can then be grouped into "super components" that can be hosted by the same virtual machine. The idea is that starting and stopping a virtual machine is costly with respect to performance, and there will be beneficial to re-use an already running virtual machine for running a component further down the workflow. The workflow analyser and the component grouping are covered in Section 6 below.

The application code will also be *profiled*, which means that it will be analysed, and the various components will be classified according to how well they match open-source projects with a known profile. This process is described in Deliverable *D3.1 Software, tools, and repositories for code mining*. The result will be a *template model* that is assumed to be the better model for deploying the application containing the various hardware accelerator options available for the different application components. The requirements of the template model and the grouped model will then be tuned to find the best requirements for the "super components" to be deployed. It is obvious that a virtual machine to execute two components that have been grouped must minimally have resources in all resource dimensions based on the maximum requirements of either of the two groped components. The location of the corresponding grouped "super component" may be forced to the same location of the data set the two grouped components are working on, hence either integrating a data component in the super group or binding the location of the super group to be the location of the data component. Finally, the profiling may indicate that this application probably is a *High-Performance Computing (HPC)* application, and that they have significant communication with other components using the *Message Passing Interface (MPI)*, and so the "super component" at hand should be co-located to the same data centre as the other "super components" taking part of the application's MPI communication group. The parameter tuning and model optimization is covered in Section 7 below.

Once the requirements of the resulting "super components" of the application have been fixed, the best possible deployment configuration for this application is found based on the forecasted application metric values. The *solver* tries to maximise the utility by iteratively requesting the utility value for a candidate configuration from the *utility generator*, which in turn will invoke the *performance module* trying to obtain the better values for selected parts of the utility function. The solvers are described in Section 4 and Section 5 below, and the performance module is described in Section 8. The utility function has been elaborated by the DevOps engineer initially using the *utility function creator* and subsequently the utility function will be optimized based on the performance module's predictions of the utility metric value. Utility modelling is described in Deliverable *D2.3 Proactive utility - Framework and approach*, and the forecasting module will be described in Deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*.

The described process until now will be sufficient for creating an optimized *initial deployment* model. However, if the application is running concurrently with this optimization process, then the cost of adaptation must be taken into consideration. The cost is computed by the *lazy adapter*. It is called "lazy" because it will try to keep as much as possible of the current deployment and only make the most minimal reconfiguration. The reconfiguration cost is considered by the utility generator as a *reconfiguration penalty* when it computes the utility value of a proposed application configuration candidate, and some reconfigurations may be considered too costly to perform at all and then the lazy adapter will generate a set of *prohibition constraints* to the solver so that its proposed configuration candidates will never suggest such impossible reconfigurations. The adaptation process is described in Section 9 below.

Finally, once the application model has been optimized with the currently best configuration for a future *application execution context, i.e.,* represented by the predicted involved application monitoring metric values, then the optimized deployment model with a set of minimal reconfiguration actions to be applied to the running application will be passed over to MELODIC for enactment and managed execution. Necessary run-time adaptations of this model will then be managed by MELODIC while the outer optimization loop of MORPHEMIC will continue the search for a more optimal application *model* to deploy.

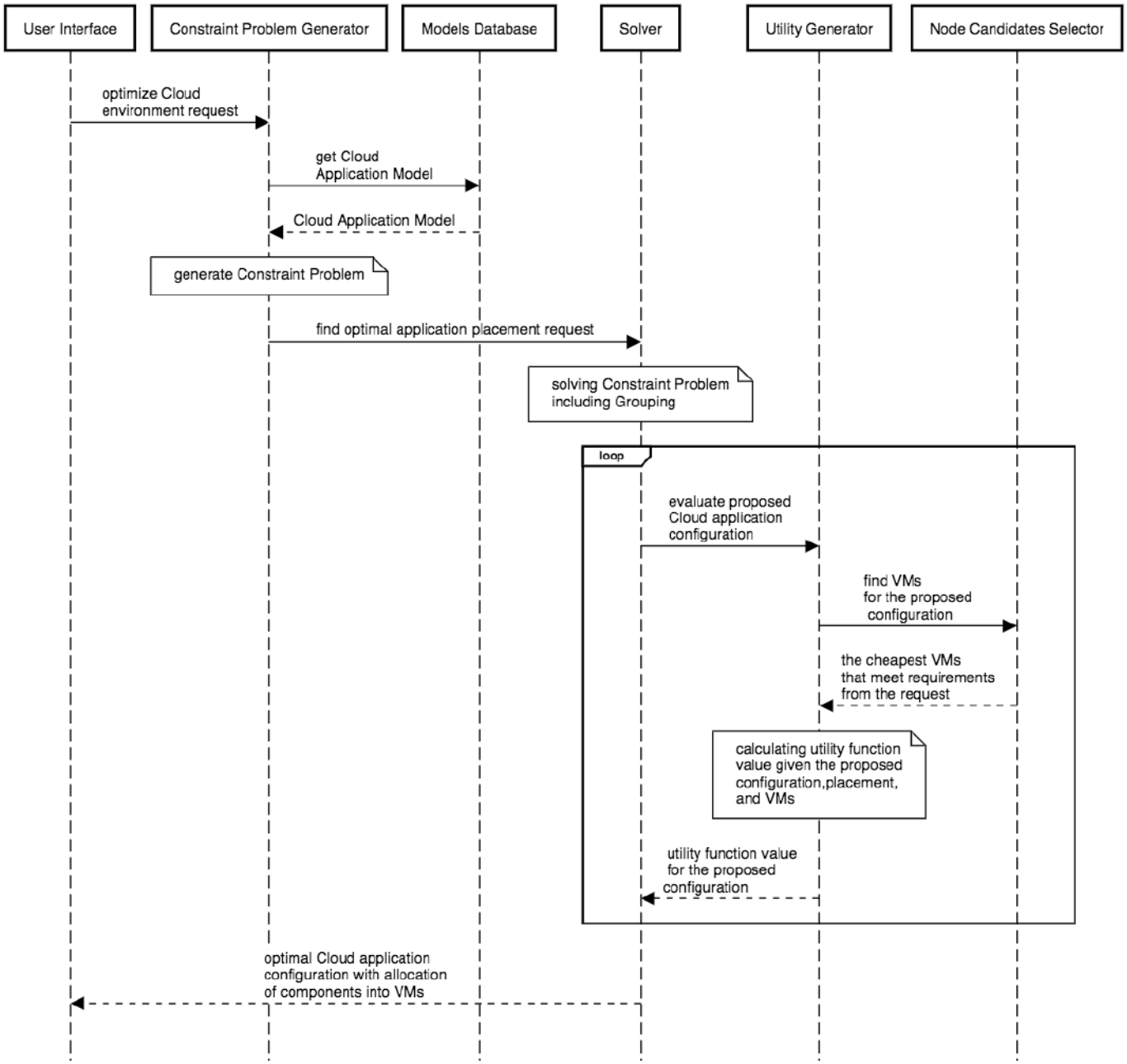

Figure 5 Simplified sequence of Cloud application optimisation in MELODIC

3.2.2 Application Model

Application model is the starting point of the optimization loop. Expressed in CAMEL Language, this application model touches mainly three domains: (a) the application domain to describe the application architecture and its variants; (b) the requirement domain to explicate the main requirements of the application and its components; (c) the metric and constraint domains to cover the specification of relevant metrics and attributes as well as constraint for supporting application deployment reasoning and monitoring (more details are available in *D1.1 Data, Cloud Application and Resource Modelling)*.

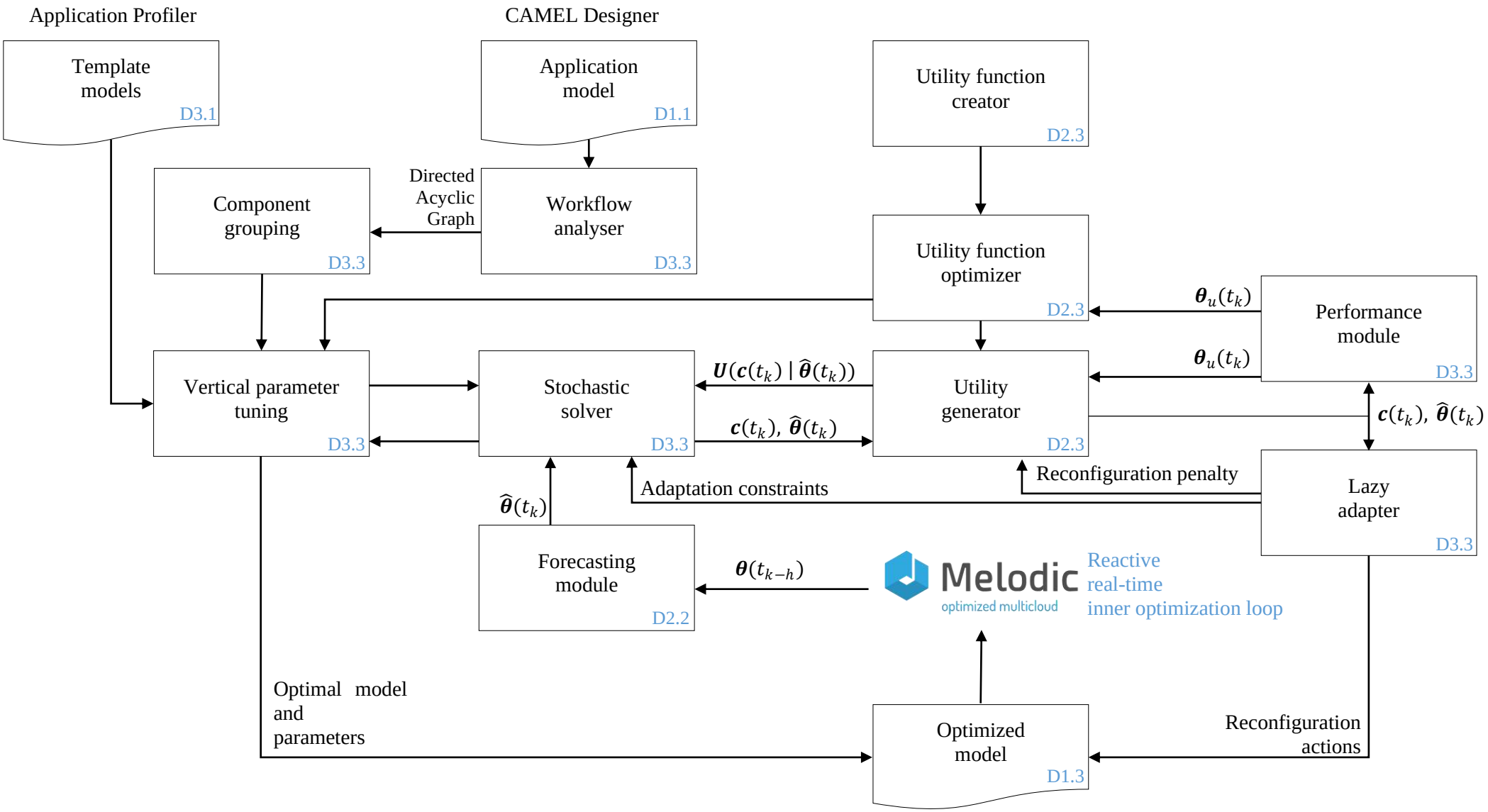

Figure 6 The MORPHEMIC outer optimization loop

3.2.3 Template models

As a cloud management system for polymorphic applications, MORPHEMIC platform must have the ability to move from one application form/configuration to another in order if the change improves the utility value. There is a need to discover different forms under which a cloud application can run and also possible hardware accelerators supported. While the forms and configurations must be included in the CAMEL model for being taken into account when triggering architecture adaptation, the Profiler, one of MORPHEMIC must suggest (complete) others forms and configuration after analysing the application's source code as developed in D2.3.

Consider the genome application where a complex dataset must be trained, and the training time should stay in an accepted range. According to section 3.3, the utility value is essentially composed of the training time and the cost of different worker virtual machines. Profiling the genome application will consist of discovering different variants (VM, docker, HPC) and different supported hardware such as : *Graphics Processing Units (GPU), Fields Programmable Gate Array (FPGA)* that could improve the application utility by a static analysis of the genome source code. From the description of the genome in camel model, the profiler will retrieve the git repository link for extracting code characteristics for finding the probability of the genome to be compatible with available forms and configuration. A *Machine learning (ML)* model is constructed by collecting a certain number of repositories and classifying them according to their label. The label refers to the forms and configurations.

Figure 7 presents the architecture used for constructing the experimental profiler. Github[2] API has been used for retrieving repositories given certain keywords (labels).

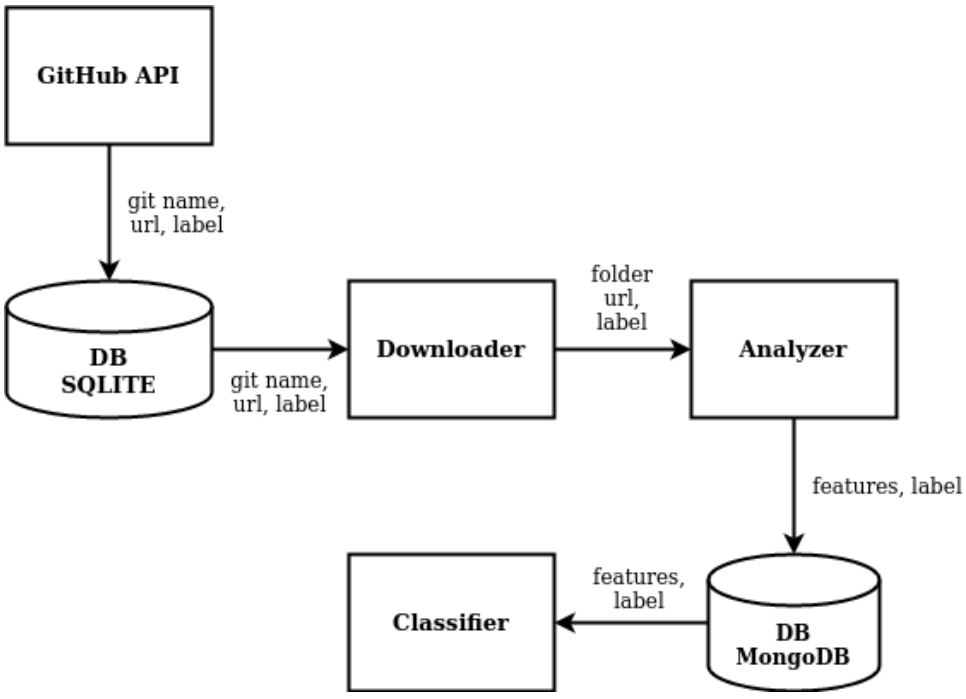

Figure 7 Profiler experiment architecture

Each repository was downloaded for extracting relevant features for discovering common characteristics. The extraction of the features is performed by the analyser which targets libraries included in source code and configuration files. These

[2] https://github.com

elements are stored as separated words (bag of worlds) in MongoDB where a collection is dedicated for storing the list of words and labels associated to them. The last step consisted of passing the list of words and labels to the classifier for creating clusters of forms and configurations.

We applied neural networks Convolutional Neural Network (CNN) and K-Nearest Neighbour (KNN) with respectively 97% and 96% accuracy. The KNN model is easier and quicker to train. For the KNN we reduced vectors' dimension for improving training time and cluster representation are presented in the above picture. The number of repositories by forms and configuration are presented in Table 1.

Table 1 Repositories by labels

Label	Number of projects
docker	2023
serverless	1940
edge	1996
gpu	3975
fpga	2134
hpc	2036

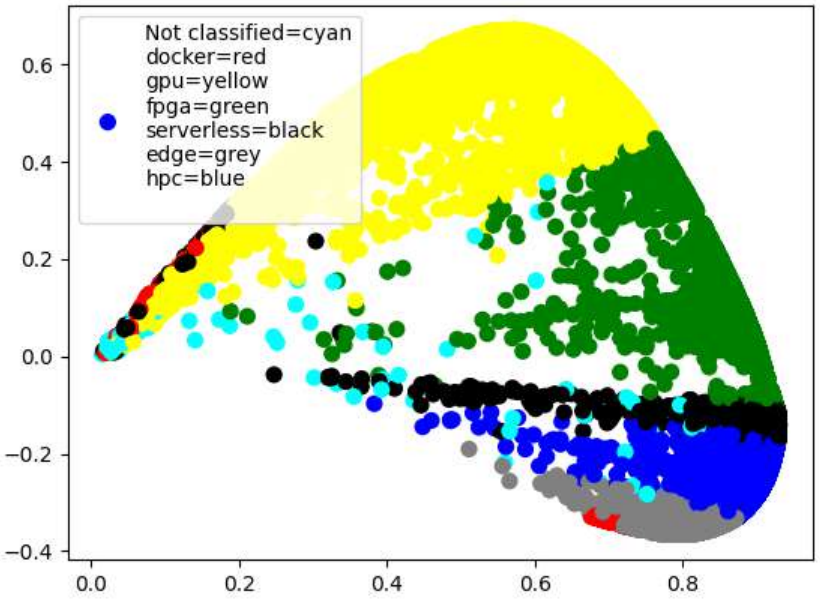

Figure 8 Cluster representation of forms and configurations

3.2.4 Utility function creator

The Utility Function Creator is a new module which the 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 specifying preferences regarding application optimisation in the visual way. The architecture and functionality of the Utility Function Creator will be described in detail in Deliverable *D2.3 Proactive utility - Framework and approach*.

3.2.5 Utility function optimizer

This component aims at improving the initial utility function created by the user. The point of departure is that there are various functional parts of the utility function that can be measured as independent metric values. Consider for example Equation (6) estimates the time required to complete the remaining trainings proportional to the number of trainings left to do, $\theta_1(t_k)$, *i.e.* it is a linear function of $\theta_1(t_k)$. When a training is completed, one has the observation of the event time t_{k+h} and the corresponding total time for all the $k+h$ trainings completed, $\theta_3(t_{k+h})$. 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. In general, it is 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.

Five approaches for inferring and model the relationships among metric values for sub-utility function expressions will be investigated and tested:

1. A regression approach [13]. For the example, in Equation (6) the measured metric values can be used to fit *better* the linear relation of number of trainings remaining as a function of total computation time. Given that not all metrics of the application execution context equally explain the observed behaviour, it is proposed to approach the regression problem with a generalized *Principal Component Analysis* (PCA) [14]. However, the relationship does not have to be linear, and adaptive fractional polynomial modelling seems to be a promising approach to capture linear and non-linear relationships [15].
2. A Bayesian regression approach where one may start with an *a priori* idea of the regression function, like the linear relation between computation time and the number of trainings left to do. This functional form will then be updated as new training events are completed producing a possibly non-linear *a posteriori* regression model [16]. Alternatively, one may use *support vector regression* (SVR) with a suitable kernel to learn the possibly nonlinear correlations involved in a utility relation like Equation (6) among the metric values and the configuration parameters [17].
3. A black-box approach for utility function sub-expressions: It is possible to consider a relation like Equation (6) as a black box and use a machine learning algorithm to learn directly the involved relations from the correlations detected in the incoming measurements. Any kind of neural network applicable for non-linear regression can be used in principle, although only a few neural networks have been designed with numerical regression in mind. The Self-Normalizing Neural Network seems like a good starting point for the model free regression [18].
4. A marginal utility function approach to model the overall utility function. Multi-attribute utility theory [19] is based on the idea of multidimensional probabilistic copulas from statistics [20] where one models the marginal univariate utilities, and then systematically combine these into a multidimensional expression. There are recent approaches using polynomial characteristic functions for Archimedean utility copulas allowing the construction of the global utility (copula) from assessing the utility at its corner values [21].
5. A sequential a *surrogate problem* approach as known from multi-objective optimisation with costly objective function evaluations [22]. As described in Section 2.1, the utility problems are multi-dimensional, and the DevOps engineer's utility function must find a reasonable compromise between the different utility dimensions. These dimensions can be captured as the DevOps engineer's high-level deployment goals. The DevOps engineer should be able to provide an *aspiration level* for the normalized utility in each dimension, and the *reference point* of the multi-dimensional utility is the vector of these aspiration levels. The idea is then to model the utility in each dimension from the measurement data according to one of the three first modelling approaches, with the goal to minimize the difference between the *modelled utility point*, obtained by evaluating the models for

the given configuration and measured application context, and the reference point provided by the DevOps engineer. The surrogate problem consisting of the modelled dimensional utility functions can then be used to identify possibly better configurations, and the deployment of these possibly better configuration may again provide new measurements helping to improve sequentially the modelled dimensional utility functions and thereby the surrogate problem.

These approaches are further discussed in Deliverable *D2.3 Proactive utility - Framework and approach*.

3.2.6 Vertical Parameter Tuning

As application requirements and especially resource constraints can be imprecise, in particular, when users do not have the right skills to go beyond a certain precision level, there is a need for a novel functionality, which is able to improve the resource constraints of applications in order to make them more precise and thus increase the changes for achieving true application deployment optimality. Such a functionality is realised through the Vertical Parameter Tuning component, which is able to produce precise upper bounds on resources for each configuration type of application components based on the modified CAMEL model of the application as given by the Component Grouping and the up-to-date utility function content as given by the Utility Function Optimizer. This functionality is realised through a simulation-based approach, as analysed in section 5.1, which is continuously executed in order to identify more optimal bounds as well as anticipate changes on utility functions (as delivered by the Utility Function Optimizer) as well as deployment feedbacks (as given by the Stochastic Solver). Such an adaptive, continuously running approach then produces new optimal bounds on the application component resources and thus updates the application's CAMEL model before it is handed over to MELODIC in order to start the classical (application deployment) optimisation loop.

3.2.7 Utility generator

The Utility Generator is the component responsible for calculating the utility function value. It was described in detail in MELODIC Deliverable D3.5. The Utility Generator receives the available Node Candidates offers, Constraint Problem and utility function, and calculates the utility for the proposed Constraint Problem solution candidate. The Utility Generator will be extended to handle more complex utility functions, it will be integrated with the Performance module and the Lazy Adapter. Furthermore, the extension of the Utility Generator to support Proactive adaptation will be described in Deliverable *D2.3 Proactive utility - Framework and approach.*

3.2.8 Forecasting module

The Forecasting Module is the component responsible for forecasting future values of the metrics, in the given time horizon. It has pluggable architecture and allow for adding new forecasting methods. It also ensembles the forecasted predictions to achieve the highest possible level of accuracy and robustness. The Forecasting Module receives real values of metrics from EMS component and based on that creates training, validation, and test sets to train forecasting methods. Then the forecasting methods, based on the incoming in real-time metrics are forecasting the future values of metrics which are used in Proactive adaptation. Detailed architecture of Forecasting Module will be described in Deliverable *D2.3 Proactive utility - Framework and approach*. The research process and evaluation results of the used time series forecasting methods will be described in detail in Deliverable *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities.*

3.2.9 Example optimization flow

The outer optimization loop always starts with the application model. The first iteration of the outer loop is just to prepare this model for the initial application deployment. However, the steps involved will continue to run alongside with the application in MELODIC trying to improve the application model. The following aims to describe the operation of the outer loop in steps that can run in parallel, hence the columns.

1	*Application profiling*	*Workflow analysis*
	The application code is profiled and matched against the corpus of available open-source software to identify if there is a typical template application model of the type of application at hand.	The user provided CAMEL model is statically analysed to identify workflows and sequences among the application components. Typically, this will be done based on the data dependencies. This module may also insert new metrics to monitor when a component starts and stops and thus automatically gather the information about the workflow for subsequent optimizations.
2	*Utility function creator*	*Grouping component*
	This is a tool used to capture the high-level goals and preferences of the DevOps engineer with respect to what is a good application deployment. The result is the initial utility function of the provided CAMEL model.	The application component execution sequence identified by the workflow, the grouping annotations in the CAMEL model, and the use of the data set components will be used to form groups of computational components with the corresponding data set components. The result will be a set of 'super components' (groups) to be deployed in separate virtual machines, and the requirement attributes for these super components.
3	*Performance module*	
	If there is historical data for the execution of the application, it is possible to derive the mapping (regression) between certain metric values, the configuration, and metric values used in the utility function. A typical example is an application's response time which can be seen as a function of the number of application users, a metric value, and the number of servers, a configuration variable value set by the solver. Thus, instead of the DevOps engineer directly expressing this mapping in the utility function, it can be learned as the application runs.	
4	*Utility function optimizer*	
	This module will compare the different *utility metrics* directly expressed in the utility function by the DevOps engineer with the ones being provided from the performance module. If the learned utility metrics are better than the functional sub-expressions in the utility function, the utility function will be rewritten to use the learned utility metrics.	
5	*Vertical parameter tuning*	
	The requirement attributes for each (super) component will be optimized and tuned. For instance, the DevOps engineer may have specified that the upper bound of the cores needed for a component is larger than actually observed as used in practice. Thus, the effect of the tuning is to reduce the search space.	
6	*Reasoning: Solver, utility generator, performance module, lazy adapter*	
	This reasoning is fundamentally the same as the one done by the MELODIC real-time inner loop. The Solver proposes a configuration for the predicted application execution context, and the Utility Generator evaluates this proposal assigning a utility value to the proposed configuration. In assigning the utility value the Utility Generator draws on the Performance Module to calculate the values of the *utility metrics*. Furthermore, the Utility Generator	

	also receives reconfiguration penalties calculated by the lazy adapter as an indication of the complexity of changing the application's configuration from the running configuration to the new configuration proposal made by the solver. The result of the reasoning will be an optimized application configuration that is *feasible* for the predicted future application execution context for the running application. Thus, the reasoning ensures that the optimized CAMEL model is deployable, with an initial application configuration for this model for a time point sufficiently into the future to allow the reconfiguration actions needed by MELODIC to change the current application configuration. The components used in this step may be the same as used by MELODIC, they can be the same implementations instantiated as separate copies not to infer with the inner loop of MELODIC, or they can be completely new and experimental versions of the components.
7	*Lazy adapter*
	Once the optimized configuration has been decided, it will be deployed by the lazy adapter aiming to make the minimal changes to the currently running application managed by MELODIC. Once the new configuration has been deployed, the inner loop of MELODIC described in Section 3.1 will take over the application management.
8	*Application monitoring and forecasting*
	MELODIC will monitor the metric values that jointly constitutes the application execution context and check the feasibility of this execution context by evaluating the Service Level Objectives (SLOs), *i.e.*, the constraints of the optimization problem. MORPHEMIC will use the monitoring information to further improve the application's CAMEL model by forecasting the future evolution of the application's execution context.
9	*Continuous improvements*
	MORPHEMIC will *continuously* evaluate possible improvements to the CAMEL model in parallel with the running application. This implies running through Step 1-5 above to see if a better CAMEL model can be constructed. If the model is changed, then Step 6-7 will also be executed to verify the feasibility of the CAMEL model, and then deploy the better feasible model. Step 8 may then produce different forecasts if some monitoring metrics have been added or removed from the model, and then the continuous improvements of this Step 9 continue.

4 Stateless solvers

In this section, the description of MORPHEMIC stateless solvers is provided. The stateless solvers are implemented using various optimisation algorithms such as naive search, genetic algorithm, parallel tempering, and Monte Carlo Tree Search. All of these solvers do not have any memory about the previous reasoning process: they are able to solve the given Constrain Problem anytime without the need of gathering the data or learning process which makes them *stateless*.

4.1 Choco solver

Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, operations research, algorithms, graph theory and elsewhere. The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them. Constraints are just relations, and a C*onstraint Satisfaction Problem (CSP)* states which relations should hold among the given decision variables. More formally, a constraint satisfaction problem consists of a set of variables, each with some domain of values, and a set of relations on subsets of these variables. For example, in scheduling exams at an university, the decision variables might be the times and locations of the different exams, and the constraints might be on the capacity of each examination room (e.g., we cannot schedule more students to sit exams in a given room at any one time than the room's capacity) and on the exams scheduled at the same time (e.g., we cannot schedule two exams at the same time if they share students in common). Constraint solvers take a real-world problem like this represented in terms of decision variables and constraints and find an assignment to all the variables that satisfies the constraints. Extensions of this framework may involve, for example, finding optimal solutions according to one or more optimization criterion (e.g., minimizing the number of days over which exams need to be scheduled), finding all solutions, replacing (some or all) constraints with preferences, and considering a distributed setting where constraints

are distributed among several agents. Constraint solvers search the solution space systematically, as with backtracking or branch and bound algorithms, or use forms of local search which may be incomplete. Systematic method often interleaves search and inference, where inference consists of propagating the information contained in one constraint to the neighbouring constraints. Such inference reduces the parts of the search space that need to be visited. Special propagation procedures can be devised to suit specific constraints called global constraints, which occur often in real life [23].

The CP Solver is a deterministic solver based on Choco-solver[3] library [24]. Such a solver can be used both for performing initial application deployment reasoning as well as application redeployment reasoning. Once the CP model is received, it is transformed into an internal representation which is fed into the CP solving engine. During CP model solving, the CP Solver cooperates with the Utility Generator in order to compute the utility of the currently examined candidate solution. Once the CP model solution is produced, it is incarnated inside the CP model in a certain specialised part.

The CP Solver has been implemented in Java as a Spring-boot[4] application. It can be built and bundled via Maven[5] in a form of either a fat Java Archive (JAR) file or a Docker[6] image. The latter form facilitates its integration into the MELODIC's platform swarm. Internally, the CP Solver exploits the Choco Constraint Programming solving engine as well as the CDOClient[7] in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions.

4.2 Parallel Tempering

Parallel Tempering is a popular method of nondeterministic approximation of the function optimum. It is a generic Markov chain Monte Carlo sampling method which allows good mixing with multimodal target distributions, where conventional Metropolis-Hastings algorithms often fail [25]. The mixing properties of the sampler depend strongly on the choice of tuning parameters, such as the temperature schedule and the proposal distribution used for local exploration. The origins of the parallel tempering, or replica exchange, simulation technique can be traced to a paper by Swendsen and Wang [26]. The general idea of parallel tempering is to simulate M replicas of the original system of interest, each replica typically in the canonical ensemble, and usually each replica at a different temperature. The high temperature systems are generally able to sample large volumes of phase space, whereas low temperature systems, whilst having precise sampling in a local region of phase space, may become trapped in local energy minima during the timescale of a typical computer simulation. Parallel tempering achieves good sampling by allowing the systems at different temperatures to exchange complete configurations. Thus, the inclusion of higher temperature systems ensures that the lower temperature systems can access a representative set of low-temperature regions of phase space [27].

Given the finite, directed graph $G = \langle \mathbb{V}, \mathbb{E} \rangle$ which can be seen as a space of states, and function $f: \mathbb{V} \to \mathbb{R}$. The goal is to find a vertex $V^* \in \mathbb{V}$ where $f(V^*) = \sup(f)$.

There are two modules with different approaches to applying PT scheme to the Constraint Problem. The first approach is the solver unaware of the existence of Node Candidates and the second PT Solver, called also Node Candidates Solver, uses Node Candidates to move through the search space. The basic problem definition of Parallel Tempering and its mapping to the Constrained optimisation of Cloud application resources is as follows:

1. The objective function $f: \mathbb{V} \to \mathbb{R}$ is the utility function $U(\boldsymbol{c})$

[3] https://choco-solver.org

[4] https://spring.io/projects/spring-boot

[5] https://maven.apache.org/

[6] https://www.docker.com/

[7] https://wiki.eclipse.org/CDO/Client

2. The solution candidate which is an assignment of values to all decision variables, ***c***, with the restriction that this configuration may not be feasible in terms of fulfilling constraints. Solution candidate that does not fulfil constraints has the utility value 0 which excludes it from being considered as a solution in the reasoning process.
3. The neighbourhood is defined separately for each of the two solver approaches implemented:
   a. Two solution candidates are neighbours when they differ by value of exactly one variable, and the absolute difference between those values is 1 for the solver unaware of the node candidates.
   b. For solver with node candidates: Two solution candidates are neighbours when their values differ for only one component and only one of the following rules is fulfilled: the only difference is the component cardinality and this difference is equal to 1; the provider is the same and either locations are neighbours or configurations are neighbours, and cardinality is equal; the provider is different and either locations are neighbours and configurations are neighbours) or (locations are neighbours and configurations are the same) or (locations are the same and configurations are neighbours) or (locations are the same and configurations are the same) and cardinalities remain the same.

There are two modules with different approaches to applying PT scheme to the Constraint Problem. The PT Solver has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. Internally, the PT Solver exploits the JAMES[8] library as well as the CDOCLIENT in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions.

4.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) algorithms are very popular for solving complex multi-criteria decision problems. This method has been designed for a sequential problem such as games or decision making for a long-term period. Although the Constraint Problem does not have a sequential nature, MCTS approach can be used because of possibility of removing unpromising configurations. The benefit of this method, comparing to PT and GA approaches is the fact that for smaller problems it can simply search the whole solution space. MCTS algorithm is based on a playout, which are the random series that are used to calculate the best move.

MCTS is the optimization algorithm which construct the solution space in the form of tree and search this space for optimal solution using Monte Carlo simulations. MCTS consists of two strongly coupled parts: a relatively shallow tree structure and deep simulated evaluation of the given configuration. The tree structure determines the selected parameters for the given configuration. The results of these evaluated configurations shape the tree. MCTS uses four main steps:

1. In the selection step the tree is traversed from the root node until the end of the tree.
2. Next, in the expansion step a node is added to the tree.
3. Subsequently, during the simulation step moves are evaluated until the end of the tree is reached.
4. Finally, in the backpropagation step, the result of a evaluated configuration is propagated backwards, through the previously traversed nodes.

Key objects in MCTS algorithm are:

- A Node represents node in a search tree. It allows retrieval of parent and children, and it stores its value.
- Node Statistics - each instance of this class is owned by exactly one Node. It keeps Monte Carlo search specific data about node: depth which denotes owning Node depth in the search tree - this value does not change once the Node is inserted into tree and *visit count* that accumulates number of times when Monte Carlo search has traversed through owning node. Node Statistics calculate owning Node's evaluation - value which is used by Move Provider during node expansion.

[8] http://www.jamesframework.org/

- The Path is from a leaf to the root of the search tree. The path is represented by list of node values encountered during traversal of the path from root to leaf.
- The Move Provider traverses subtree rooted at the node until a leaf has been encountered. Once a leaf has been found at least one new child is added to it. It is implemented as
- Move Provider that performs standard expansions procedure - if expanded node has been encountered, its child with biggest value of *Node.Node Statistics.get Evaluation()* is chosen. Once a leaf has been found, all its children are added to the tree and one of them is chosen randomly
- The Policy is responsible for extending Path class instances to full solutions. Two policies are implemented:
- The Cheapest Policy: if at least provider and cardinality variables have been set for all components, the Utility Generator module provides functionality of extending the solution. To this end it searches for cheapest virtual machines which will match the partial solution. The Cheapest Policy assigns random values to missing provider and cardinality and uses the aforementioned functionality to complete the solution. It may happen that it is not possible to extend the solution. In such case the Cheapest Policy returns empty solution.
- The Random Policy assigns random values to variables
- The Solution is the full solution of a modelled problem.
- The Tree coordinates the work of all other classes. One tree iteration is defined to be full traversal of the tree - from root to some chosen leaf - with subsequent rollout performed by the Policy interface. Each iteration results in an instance of Solution class. Tree class provides only one method - run - which performs given number of iterations and returns best solution.
- The Memory Limiter is responsible for controlling the size of tree. It controls node count inside tree. If limit of nodes is exceeded, then Memory Limiter suggests pruning tree. It is implemented as:
- Queue and Queue Linker where Queue is a FIFO queue in which all nodes are present. Recently visited nodes are put at the back of queue. Nodes suggested to prune by the Memory Limiter are taken from the front of queue. Queue Linker is a set of methods enabling manipulation of nodes in Queue. In order for Queue insertion and deletion to work in O(1), Node contains reference to Queue Linker
- The MCTS Wrapper adds random generation of variable assignment. Serves as a factory for Policy implementations. Owns Node Candidates Provider class instance.
- The MCTS Wrapper Factory fixes a number of MCTS searches that will be conducted in parallel. Each Tree instance will be given its own MCTS Wrapper, utility Provider instances - to enable synchronization free execution. This class facilitates the creation of new MCTS Wrapper instances.
- The Node Candidates Provider receives Node Candidates for the problem. It removes nodes which are outside of variables' domains. The output is later used by Cheapest Policy during rollouts. Removal of nodes which do not belong to variables' domains ensures that every solution suggested by Cheapest Policy will be conformant of the domains
- The MCTS Solver wraps Policy, Tree, MCTS Wrapper classes together

MCTS Solver is composed of fixed number of MCTS Solvers working in parallel. After fixed number of iterations, threads synchronize and MCTS Coordinator sets new coefficient for each of the solver.

- The MCTS Coordinator is responsible for starting and coordinating MCTS searches. Communication between worker virtual machine threads and MCTS Coordinator is done entirely through One To Many Channel class.
- Worker virtual machine thread encapsulates work done by thread responsible for one MCTS search.
- One To Many Channel implements a blocking message channel which is used for communication between worker virtual machine threads and MCTS Coordinator ("one" corresponds to MCTS Coordinator).

The MCTS Solver has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. Internally, the Solver exploits the CDOClient in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions.

4.4 Evolutionary algorithms (GA)

Nearly three decades of research and applications have clearly demonstrated that modelling the search process of natural evolution can yield very robust, direct computer algorithms, although these models are crude simplifications of biological reality. The resulting evolutionary algorithms are based on the collective learning process within a population of individuals, each of which represents a search point in the space of potential solutions to a given problem. The population is arbitrarily initialized, and it evolves toward better and better regions of the search space by means of randomized processes of selection (which is deterministic in some algorithms), mutation, and recombination (which is completely omitted in some algorithmic realizations). The environment delivers quality information (fitness value) about the search points, and the selection process favours those individuals of higher fitness to reproduce more often than those of lower fitness. The recombination mechanism allows the mixing of parental information while passing it to their descendants, and mutation introduces innovation into the population [28].

Solver implementation is based on the Jenetics[9] library, which is the most popular open-source Java library for genetic algorithms. Thanks to use Jenetics library it was possible to quickly build the prototype to validate the efficiency of these algorithms for cloud computing resource optimization. As the results are promising we plan to research and use the more advanced evolutionary algorithms like NSGA-II [29]. Interfaces required by Jenetics that are implemented in Genetic Solver:

- *Gene* is a representation of variable assignment. Genes hold integers that represent either number or list index.
- *Chromosome* contains list of genes. Chromosome is practically a population individual.
- *Mutator* mutates single genes. Mutation is uniform.
- Selector Using default selector.
- *Eval Function* that for each genotype (chromosome's wrapper) returns its fitness.
- *Crossover* Using the *Single Point Crossover* implemented in Jenetics.

The algorithm is designed according to classical Genetic algorithm and it includes following steps:

1. Population is initialised using random values.
2. During the evolution process
    a. Population is being cross overed.
    b. Genes are muted.
    c. The best genes are chosen.

The Population's individual is represented by the Phenotype. The Phenotype consists of one Genotype. Each Genotype has exactly one Chromosome. Every Chromosome contains sequence of Genes. Chromosome is an assignment of values for all variables. Gene holds integer value that represents variable assignment.

It is important to notice that the translation of the constraint problem into a genetic algorithm is not trivial, but this field has been already researched. The most difficult part is connected with the representation of the assignment of values on variables that does not fulfil constraints. The first approach can be to simply delete such gene, but it can be inefficient. The implemented method uses stochastic ranking. It prefers feasible assignments but if all are not feasible, it prefers this with lower number of broken constraints. A Stochastic Rank Comparator is used in methods where chromosomes that survived too many iterations are deleted and methods where top chromosomes are chosen to be reproduced.

The GA Solver has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. Internally, the GA Solver exploits the Genetics library as well as the CDOClient in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions.

[9] https://jenetics.io/

5 Stateful solvers

5.1 The purpose of stateful solvers

It is important to note that the application's execution context is represented by the vector of monitoring metric measurements, $\boldsymbol{\theta}(t_k)$. Each metric value in this vector is updated when the monitored *event* happens, which is, in general, on an irregular basis. Hence, the event times $t_k, t_{k+1}, \ldots$, defines the time points when the application's execution context is updated, and the metric vector moves from time point to time point one element at the time. With *event driven sampling* it is not possible to know *a priori* when the next event time will be, and it is not possible to say how many events that will occur in an interval, Δt.

For every time point one must assert if the current configuration is still valid. A *feasible* configuration satisfies the problem constraints, Equations (3) and (4), and these constraints must be evaluated at every time point when a new measurement arrives. If the current configuration is infeasible, a new and better configuration must be found.

The first problem comes from the fact that mathematical programming solvers assume that the objective function, here the *utility function* of Equation (2) returns the same value if it is evaluated twice for the same proposed configuration candidate, $\boldsymbol{c}$. Since the utility function (2) is *conditioned* on the application's execution context, $\boldsymbol{\theta}(t_k)$, its value will change when the metric vector changes, *i.e.*, at every monitoring event time point. A *stateless* solver, like the ones described in Section 4 above, therefore requires that the measurements are *frozen* at $\boldsymbol{\theta}(t_k)$ with t_k being the time point when the solver is started to search for a better configuration than then current infeasible configuration. If the first time point after the solver has converged on a solution is t_{k+s} it means that there are s monitoring events not considered by the solver, and the solution provided by the solver will be optimal only for the time point when it was started, hence the notation $\boldsymbol{c}^*(t_k)$. Using predicted metric values is one way to overcome this problem: Instead of freezing the current metric values at $\boldsymbol{\theta}(t_k)$, one will solve the problem for the metric values at $\boldsymbol{\theta}(t_{k+s})$, and therefore obtain a valid optimized configuration $\boldsymbol{c}^*(t_{k+s})$.

Alternatively, one can use a *stateful* solver. This solver will run continuously in the background receiving the metric vectors as they are observed and use this information to prepare a new solution for the 'current time', which can again be a predicted future taking into consideration the time it takes to reconfigure the application. This solver must then be capable of *learning* the better solutions as new measurements arrive based on the event driven changes of the application's execution context. This deliverable describes the initial approaches and implementation ideas, but the implementation and validation will be for the last part of the project.

5.2 The Learning Automata (LA) approach

The observation is that solving an optimization problem with real valued variables has polynomial complexity in the number of variables. It is the combinatorial part of the problem, *i.e.* the discrete variables that causes the exponential complexity of the optimization algorithms. The idea is therefore to use the utility value as a *reward strength* for a set of *reinforcement learning* algorithms [30], one for each discrete variable. Each learner has the domain of its variable as possible 'actions', and its goal is to identify with some probability what is the best action for the current situation. Most reinforcement learning algorithms focus on learning the probability vector for the full action set, which is not needed here. There is a sub-class of reinforcement algorithms called *Learning Automata* (LA) that aims at identifying only the best possible action in the most efficient way [31], which has been proposed for the optimization of Cloud applications [11].

5.2.1 Discrete variable: Selecting from the variable's domain

The LA work by maintaining a vector of probabilities, $\boldsymbol{p_i}(t_k)$, and when the automaton receives a positive feedback it has an algorithm $P^+(\boldsymbol{p_i}, U)$ that will increase the probability of the currently selected 'action', which is the value for the variable c_i in its discrete domain $\mathbb{D}_i$. Should the LA receive negative feedback, it will reduce the probability assigned to the currently selected 'action' according to an algorithm $P^-(\boldsymbol{p_i}, U)$ that distribute the reduced probability mass for c_i over the other possible values of the variable domain increasing the possibility of selecting a different value for c_i from

the domain on the next iteration. The probability values of the vector $\boldsymbol{p_i}(t_k)$ will always sum to unity. An initial solver based on these techniques was included in the MELODIC platform. The main idea is shown in Algorithm 1.

Algorithm 1 Finding values of integral variable c_i using Learning Automata

	Input: $\boldsymbol{\theta}(\boldsymbol{t_{k+\mathrm{h}}})$	*New metric vector*
	$U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi})$	*Current utility value*
	Output: c_i	*New value for configuration variable i*
1	**if** $U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_{k+\mathrm{h}}), \boldsymbol{\phi}) \geq U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_k), \boldsymbol{\phi})$ **then**	*New utility better or equal to current*
2	$\boldsymbol{p_i}(t_{k+\mathrm{h}}) = P^{+}\left(\boldsymbol{p}(t_k), U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_{k+\mathrm{h}}), \boldsymbol{\phi})\right)$	*Increase the probability of the current choice*
3	**Else**	*New utility worse than current*
4	$\boldsymbol{p_i}(t_{k+\mathrm{h}}) = P^{-}\left(\boldsymbol{p}(t_k), U(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_{k+\mathrm{h}}), \boldsymbol{\phi})\right)$	*Decrease the probability of the current choice*
5	**end**	*Probability vector update*
6	**if** ($\boldsymbol{g}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\phi}) > \mathbf{0}$ **or** $\boldsymbol{h}(\boldsymbol{c}(t_k) \mid \boldsymbol{\theta}(t_{k+h}), \boldsymbol{\phi}) \neq \mathbf{0}$) **then**	*Current configuration infeasible*
7	$\mathrm{c_i} = \{ \mathrm{c} \in \mathbb{D}_i \mid \mathrm{c} \sim \boldsymbol{p}_i\,(\mathrm{t_{k+h}}) \}$	*Select randomly a new value from the domain*
8	**End**	
9	**return** c_i	*Value of configuration variable i*

If there are several discrete variables, the new utility value for the current context can be evaluated once, and all the probability updates can possibly be done in parallel. It is also possible to select the new configuration variables in parallel if the current configuration is infeasible.

There are, however, two issues with this approach:

1. It is important to balance exploration of the variable domain and exploitation of knowledge of the best values. The learning, *i.e.* the change in the probabilities, should therefore be made in small steps, and the initial probabilities should preferably be initialized with historical data from past application executions. Without prior knowledge, each value in the domain shall be given the same initial probability, $1/|\mathbb{D}_i|$.
2. After the LA have assigned values to the discrete configuration variables, one will still not be able to assess if their values are *feasible* because the feasibility is decided for the whole configuration vector and this most likely also contains real variables that must be assigned values. Thus, one cannot speed up the convergence by discarding infeasible values and reselecting values only for the discrete variables. Furthermore, it is in general not even possible to decide if there are real variable values that will make the configuration feasible given the chosen discrete variables.

The implementation of this strategy must make the polymorphism among the different types of discrete variables transparent to the solvers irrespective of whether the discrete variable is a set; a range over integral variables, *i.e.* a set that is not completely enumerated; or a set of ranges over integral variables; or a set of strings. Figure 9 shows the class diagram for representing transparently all variable types in the LA solver.

5.2.2 Real variables: Optimization

With the discrete variables assigned and fixed one may solve the mathematical programme for the real values keeping the application's execution context constant. In principle any solver with polynomial complexity may be used, for instance the one mentioned in Section 4. It should be possible to obtain the solution quickly because the small number of real valued variables to solve. The time budget is basically until next measurement event because that will render obsolete the fixed application execution context used by the solver to find the best real variable values, see Deliverable *D2.3 Proactive utility - Framework and approach*. If no solution can be obtained within this time budget, one has two choices: stop the solver and start again hoping for more time before the next measurement event; or one may just start another parallel solver, and another, and so forth. As each solver terminates, one may simply check the feasibility of the solution for the current application execution context at that time, and if the configuration is feasible and the utility of this configuration is better than the currently running configuration, then it is a better configuration that should be deployed. This heuristic should work in most situations, but there is a risk that all the parallel solvers produce infeasible configurations for the application's execution context at the time the solvers terminate, and hence no acceptable solutions can be found to replace the currently infeasible application configuration.

The alternative is to deploy an *anytime* optimization algorithm [32]. Anytime algorithms can be stopped at any time and returns the best solution available when interrupted. The problem is of course that only a limited part of the search space can be explored, and one will run the risk to find only a locally valid solution. *Simulated Annealing* (SA) is a class of algorithms that can find the global optimum in non-linear systems, and Wah and Chen have proposed a constrained anytime version [33]. Alternatively, one may explore the Pareto front of the multi-objective optimisation problem, *i.e.* when the various dimensions of the utility problem are considered as individual objectives. The Pareto front is the set of objective vectors where no single utility dimension can be improved without deteriorating the utility in other dimensions. Domínguez-Ríos *et al.* recently proposed an exact anytime algorithm for multi-objective combinatorial optimization [34]. Their TBA-algorithm will find all points on the Pareto front if allowed enough time. However, even if terminated early, Domínguez-Ríos *et al.* claim that the found points on the Pareto front will be dispersed, and it should therefore be possible to approximate the Pareto front by interpolation over the Pareto points returned by the TBA-algorithm.

5.2.3 Real variables: Regression

Given the issue of timely solving the optimization problem to have a valid solution for the real configuration variables for the current application's execution context before this context is changed by the next measurement event, it would be desirable to compute or approximate the real configuration variables as a function of the integral variables and the application's execution context. The feasible region $\mathbb{F}(\boldsymbol{\theta}(t_k))$ can be partitioned into a region for the discrete configuration variables $\mathbb{F}_D(\boldsymbol{\theta}(t_k))$ and a region for the real configuration variables $\mathbb{F}_R(\boldsymbol{\theta}(t_k))$ such that $\mathbb{F}(\boldsymbol{\theta}(t_k)) = \mathbb{F}_D(\boldsymbol{\theta}(t_k)) \cup \mathbb{F}_R(\boldsymbol{\theta}(t_k))$ and $\mathbb{F}_D(\boldsymbol{\theta}(t_k)) \cap \mathbb{F}_R(\boldsymbol{\theta}(t_k)) = \emptyset$. The vector of the discrete configuration variables, $\boldsymbol{c_D}(t_k) \in \mathbb{F}_D(\boldsymbol{\theta}(t_k))$, is then decided by the learning automata as described above in Section 5.2.1, and the idea is to learn a mapping for each real value $c_r(t_k) \in \mathbb{F}_R(\boldsymbol{\theta}(t_k))$ from the discrete configuration variables and the metric vector, $c_r(t_k) = f_r(\boldsymbol{c}_D(t_k), \boldsymbol{\theta}(t_k)) \in \mathbb{F}_R(\boldsymbol{\theta}(t_k))$.

The optimisation approaches of Section 4 must be used to populate the data set that can be used to train one regressor function $f_r(\boldsymbol{c}_D(t_k), \boldsymbol{\theta}(t_k))$ for each real configuration variable. It can be expected that the regression error between the trained regressor and the result of the optimisation decreases as the number of monitoring events increases. At one point in time, the regression error will be less than a small bound, ϵ, after which the regressor functions can just be evaluated to obtain the corresponding real valued configuration variables as new application configuration context vectors arrive. This initial use of an optimizer can be omitted if one has access to historical data for the training, or there are pre-trained regressor functions available.

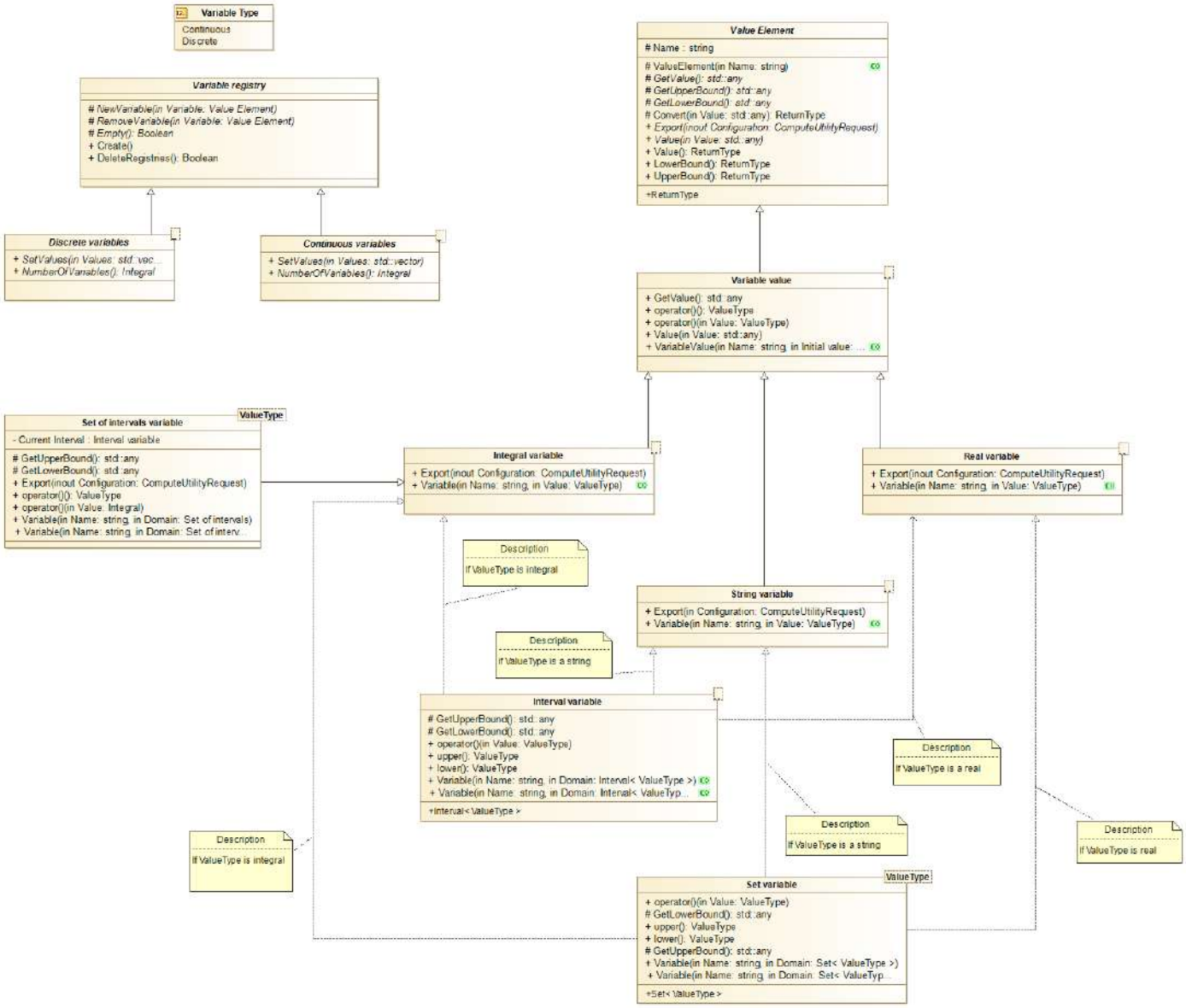

Figure 9 The class diagrams of transparent variable modelling

Once the regressor has been trained, one can evaluate the regressor for each new measurement $\boldsymbol{\theta}(t_{k+h})$ of the application's execution context to have the real value of the corresponding configuration variable. With all the discrete and real variables proposed, one can evaluate the SLO constraints (3) and (4) to assess that the candidate configuration is feasible, and if so, then evaluate the utility of this candidate configuration to see if it is better than the currently running, and propose the candidate configuration as the new configuration if it is better.

The relations the regressors try to identify are in general non-linear. It is therefore important to use regressors capable of modelling non-linear relationships. The *Support Vector Regressor* (SVR) is one candidate supporting non-linearity through the application of a non-linear *kernel* [17]. Many different kernels have been proposed, each having specific properties. However, the drawback is that the kernel must be selected *a priori* based on the anticipated relationships. The *Self-Normalizing Neural Network* (SNN) is a deep learning based alternative developed for regression [18]. It avoids the need for a hypothesis for the non-linearity but requires the number of layers to be fixed as a parameter. Furthermore, experiments by the proposers of the SNN show that very deep networks may be needed to get the best accuracy, and this makes the training and re-training of the network more costly.

Three other regression approaches will also be evaluated for this purpose: The idea of fitting polynomials locally applying different polynomials to different sections of the data [35], using fractional polynomials [15], or piecewise interpolating splines [36].

5.3 Model predictive control

The above discussion assumes a two-step heuristic where the integral variables are selected first, and then the real valued variables are decided given the values assigned to the discrete variables. Alternatively, the whole configuration vector is produced either in response to every change in the application's execution context, or whenever the current configuration is no longer feasible for the current application's execution context, and no explicit optimisation of the utility function is necessary.

If the initial configuration, $\boldsymbol{c}(t_0)$, has been calculated carefully to produce the global optimum of the utility function, one may model the system evolution as a dynamic time variation around this optimal system state. In the language of *automatic control* this means that a control signal is applied to stabilize the system at the optimal utility point given the "disturbances" of the changes in the application execution context. Thus, the challenge is to model the changes in the optimal system state, *i.e.* the configuration vector, as a function of the changes in the monitoring measurements. Then a control signal can be applied to eliminate the effect of the changes aiming to keep the resulting configuration vector optimal for the new execution context. Hence, the postulate is that the changes in the configuration vector can be modelled as a first order dynamical system,

$$\frac{d\boldsymbol{c}(t)}{dt} = \boldsymbol{f}(\boldsymbol{c}(t) \mid \boldsymbol{\theta}(t), \boldsymbol{u}(t), \ \boldsymbol{\psi}) \tag{12}$$

where $\boldsymbol{c}_0 = \boldsymbol{c}(t_0)$ and $\boldsymbol{u}(t)$ is the vector of changes in $\boldsymbol{c}(t)$ necessary to keep the system stable, *i.e.* to make the derivative zero in Equation (12), and $\boldsymbol{\psi}$ is a vector of model parameters that must be estimated from historical data [37]. The model must be created from first principles regarding the anticipated influence on the elements of the configuration vector. For the case of the big data farming application of Section 2.3, the number of instances will change positively with the estimated training time completion as more worker virtual machine instances will be needed as the training time grows larger. At the same time, the number of cores per worker virtual machine and the number of worker virtual machines will negatively impact the cost, and one would in this case probably like to introduce the cost as an additional system state that should be kept constant.

The benefit of modelling the system as a dynamical system is that it will then admit *model predictive control* [38], *i.e.* using the dynamical model one may pre-compute the control signal that will keep the system stable based on predicted metric values and thereby ensure that the right reconfiguration steps are taken early enough for the system to be reconfigured when the new configuration is needed. The idea is that one uses the model to compute the optimal control signal for a finite time horizon, and then apply the first control signal to the system waiting for the next measurement from the system to start over the optimization. The issue and complexity are to find a representative parametrised system model, as this may be as difficult for the DevOps engineer as modelling the utility function by first principles.

5.4 Adaptive critic

A similar approach linked to control theory is *adaptive critic control* where the idea is that the learning system can only learn from a sequence of actions, like in two-player games where the winner is only known when the game finishes after a sequence of moves of each player [39]. For deciding on the configuration vector, one would assign one learning agent per configuration variable with reward and penalty update functions. Whether to reward or penalise the learners will be decided by the "critic" when the game iteration has been played to conclusion. The optimizer will play the same configuration vector for each game iteration, and it is intuitive that the end of the game is defined as the time point when this play is no longer feasible. Each play corresponds to a new measurement event, and a utility value in the unit interval will be scored for each play. Hence the largest total utility value possible in one game iteration is the number of times the configuration vector was played in the iteration, and the least total utility value is zero. The "strength" of the configuration is therefore the total utility value divided by the maximum utility values possible, but since this is just the length of the iteration, this is the average utility scored during the game iteration. The critic may decide to reward the

learners for the configuration if the average utility is larger than a certain threshold, which may be ½ but it can be any value in the unit interval. At the end of the game iteration, the current configuration is infeasible, and a new configuration is proposed based on the random choices of each of learning agents. One must of course ensure that the proposed configuration is feasible, and there is a developed theory on how to design the learning in order for the system to remain stable [40]. The benefit of this approach is that the configuration vectors will be evaluated over a sequence of application contexts and must therefore be robust to different execution contexts to be reinforced by the critic. The downside is that the learning takes place only once per game iteration when the current vector is infeasible, and as learning always requires a substantial number of feedbacks, it can take significant time to find an optimized configuration as it is optimized over all execution contexts, not only the current context. This is accentuated by the fact that the elements of the configuration vector are chosen individually by individual learners, and so even if each choice is optimal individually, the resulting configuration vector may be far from optimal.

It is noted in passing that adaptive critic control is often used in modern literature synonymous to adaptive dynamic programming, which will be discussed in Section 5.5 below.

5.5 Supervised Learning approach

Description of the approachThe proactive solver is trained to find the optimized configurations based on the current $\boldsymbol{\theta}(t_k)$ and the predicted execution context $\widehat{\boldsymbol{\theta}}(t_{k+h})$ for time t_{k+h}. At any time point we do have the *state* $\boldsymbol{s}(t_k)$ defined as the measured application execution context vector and the corresponding optimal configuration found by solving the optimization problem.

$$\boldsymbol{s}(t_k) = \left[\boldsymbol{\theta}^T(t_k), \boldsymbol{c}^{*T}(t_k)\right]^T$$

The supervised learning method is used for training the solver, as shown in Figure 10.

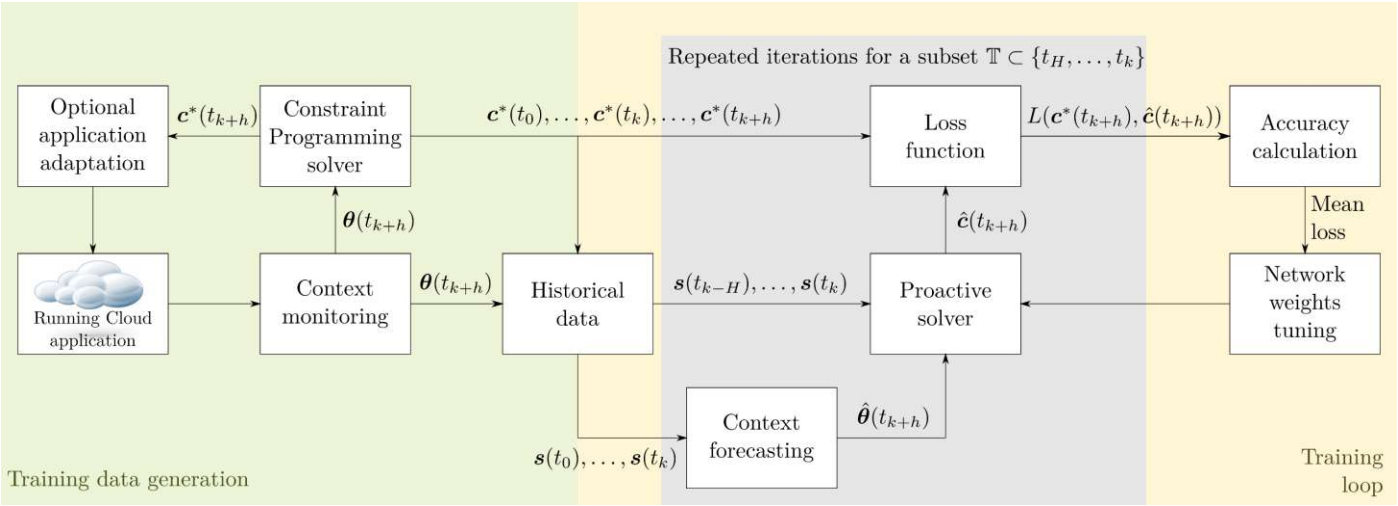

Figure 10 Supervised Learning schema for Cloud application optimization

The input for the proactive solver at time point t_k is defined as: $\boldsymbol{s}(t_{k-H}), \boldsymbol{s}(t_{k-H+1}), \ldots, \boldsymbol{s}(t_k), \widehat{\boldsymbol{\theta}}(t_{k+h})$ where H is the backward historical window to consider and h is the prediction horizon. The output of the proactive solver is then

$$\hat{\boldsymbol{c}}(t_{k+h}) \approx argmax_{\boldsymbol{c} \in V}\, U\left(\boldsymbol{c} \,\middle|\, \widehat{\boldsymbol{\theta}}(t_{k+h})\right)$$

The learning process starts with passing the input data to the Proactive solver learning module, that contains one or several neural network modules. The proactive solver produces the optimal configuration for the predicted application context. This configuration is then compared with the target configuration produced by a standard constraint programming solver to calculate the error using a loss function $L: \mathbb{V} \times \mathbb{V} \mapsto \mathbb{R}$. The loss function can be defined as any differentiable function. In this paper we experiment with standard loss functions and one new loss function defined as

the difference between the utility value of the target configuration and the utility value of the configuration returned by the proactive solver, UACC, $\Delta U(t_k)$, is defined as

$$\Delta U(t_k) = U\left(\boldsymbol{c}^*(t_{k+h}), \hat{\theta}(t_{k+h})\right) - U\left(\hat{\boldsymbol{c}}(t_{k+h}), \hat{\boldsymbol{\theta}}(t_{k+h})\right)$$

In fact, it might not be possible to use the UACC as a loss function as it cannot be assumed that every application's utility function is differentiable.

The proactive solver, based on the neural network machine learning (ML) models, needs additional activities related to model training. In contrast to stateless solvers, which are not aware of the state of the application and solve the optimization problem based on the parameters passed to them, this solver requires training on the historical data for $t_0, \ldots, t_{k+h}$ for each particular application. The target output needed for training,

$$\boldsymbol{c}^*(t_{k+h}) = argmax_{\boldsymbol{c} \in V}\, U\big(\boldsymbol{c}(t_{k+h}) \mid \boldsymbol{\theta}(t_{k+h})\big)$$

is calculated by invoking a reactive Constraint Programming Solver to calculate the optimal configuration for $\boldsymbol{\theta}(t_{k+h})$, as it can be seen in Figure 5. Based on the application states gathered until time t_k, predictions for time t_{k+h} are generated by the context forecasting component.

5.5.1 Solver components

The solver architecture and components are illustrated as in Figure 11.

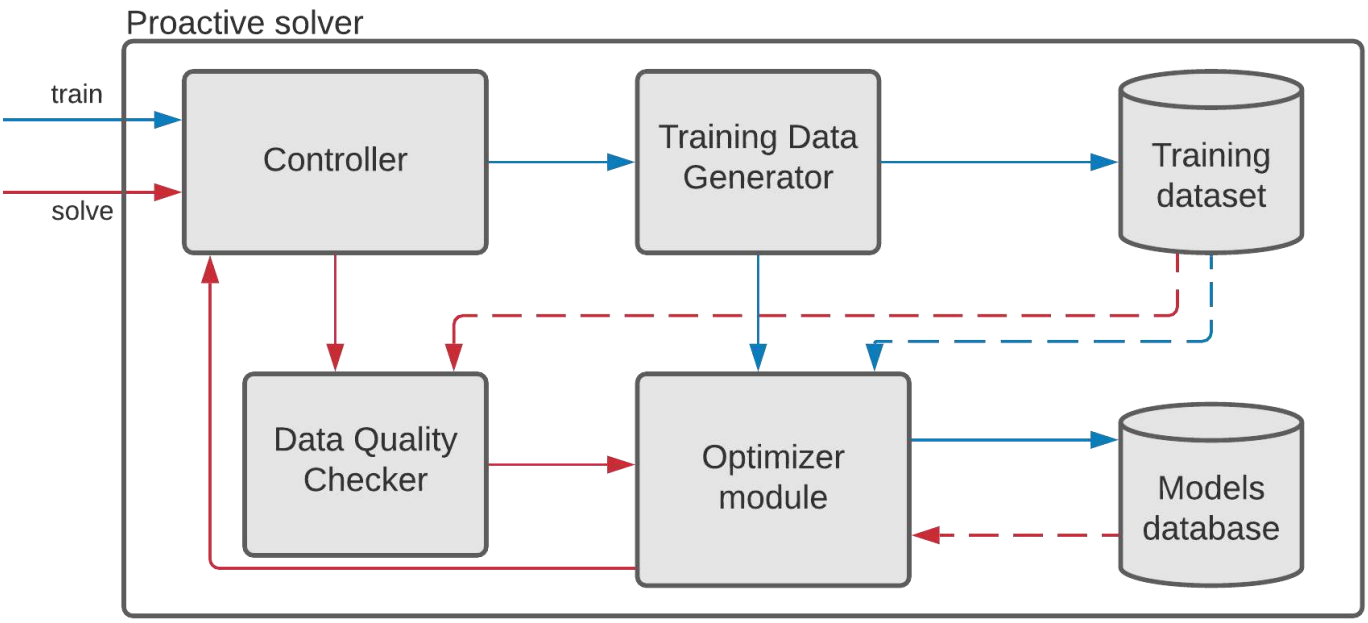

Figure 11 Solver architecture and components

The blue arrows in Figure 11 show the training of the solver control flow while red arrows show the process of optimizing the application configuration for given point in time. The dotted lines indicate reading from databases. The trained solver is integrated with the MORPHEMIC platform (see Section 5.5.5) so that it can be invoked to find the optimized configuration at any time. When a request to return optimized configuration for given predicted application context comes, the solver decides if the currently trained models can be used or if there is a need for retraining. The solver consists of the following key components shown in Figure 11:

- Controller which is the module responsible for receiving requests for starting training of the solver and when the solver models are trained, for receiving request to return the estimated optimized application configuration $\hat{\boldsymbol{c}}(t_{k+h})$.
- Training Data Generator responsible for generating the data needed for training, including interaction with the reactive solver to generate the target outputs.
- Data Quality (DQ) Module responsible for doing automatic data quality assessment, which includes calculating basic statistics (e.g. standard deviation, median) of the dataset and comparing distributions of the input metrics and variables using the Kolmogorov-Smirnoff test [41]. It assumes a null hypothesis that the new data come from the same distribution as the metrics on which the solver has been trained. On this basis DQ module decides whether to train the solver on the new data that was gathered during the application execution since the last training.
- Optimizer module the core part of the solver. The proactive solver has internally three stages: A set of neural networks that can be of different types and each output an estimate for the predicted future configuration vector, $\hat{\boldsymbol{c}}(t_{k+h})$, given the predicted future application execution context, $\hat{\theta}(t_{k+h})$, for the future time point t_{k+h}. The predicted vectors are then *assembled* into a single predicted configuration for each neural network type. The final stage is simply to select the predicted configuration that gives the highest utility value for the future time point. This architecture is illustrated in Figure 12.

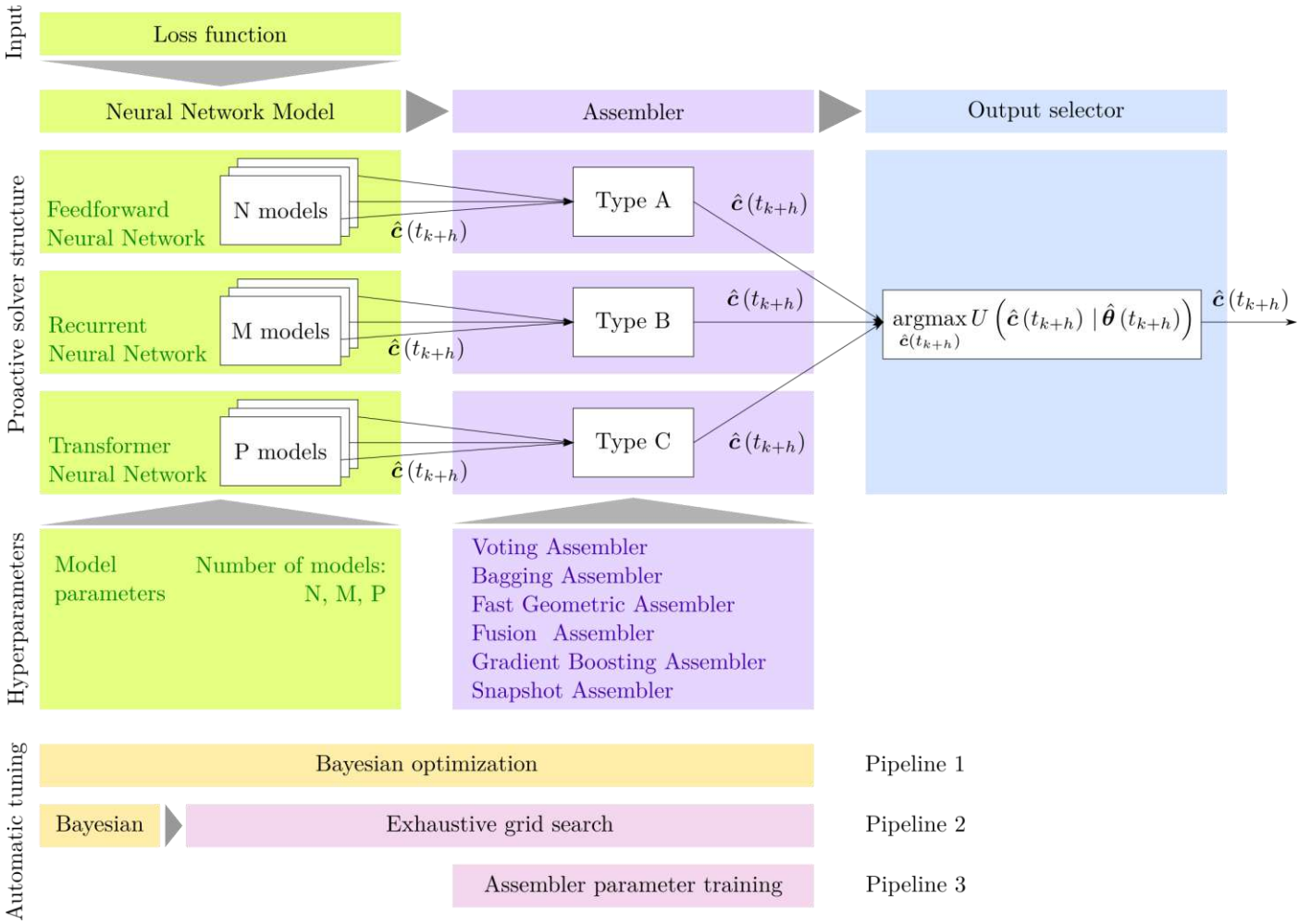

Figure 12 Internal architecture of the proactive solver optimizer module and the automatic hyperparameter tuning pipelines

However, this means that there are quite a few choices to be made, and parameters to be tuned. One must select how many neural network models to train for each neural network type, and then tune the parameters of each neural network type, *e.g.*, the number of layers and the number of nodes per layer, or other neural network type specific parameters. Then one must select the assembler to use for each neural network type and tune the parameters of the selected assembler type. It is impossible to manually tune to optimality all these parameters. Thus, the role of the *Optimizer module* is to train the hyperparameters of the proactive solver, and it offers three automated *pipelines* to undertake the hyperparameter tuning:

- Pipeline 1 selects a set of optimized model parameters, assembler type and ensemble size using the Bayesian optimization [42]. Tools from the Optuna[10] library were used for this purpose.
- Pipeline 2 optimizes first the neural network parameters using the Bayesian optimization [42]. Then an exhaustive grid search is applied to find the most appropriate assembler and number of network models to use corresponding to each of the selected neural network types [43]. The hyperparameter tuning for this approach is approximately 10 times faster than for Pipeline 1.
- Pipeline 3 is responsible for retraining a selected assembler. It requires manually to provide the architecture and parameters of the neural network types, the number of each neural network type to use, and the assembler type to use. Owing to the error-prone manual specifications needed, this pipeline will not be used in the following.

Three different neural network types are available for the proactive solver in this study. It should be noted that the hyperparameter tuning may result in no models to be used for a particular type, and so it is not given that all three types are actually in use for a given application. The neural network types are the following:

- Feed Forward Neural Network (FFN) is the most basic type of neural networks, for which connections between the nodes do not form a cycle but connect the subsequent network layers[44].
- Recurrent Neural Network (RNN) [45] is an neural network type that consists of multiple hidden recurrent layers, e.g. LSTM [46] or Gated Recurrent Unit layers[47].
- Transformer [48] uses the encoder-decoder architecture and *attention* mechanism [48]. The encoder consists of convolution layers [49] which encode an input time series into a d -dimensional feature vector, where d is found in the optimization process, and attention layers. The decoder part is responsible for calculating predictions based on feature vector.

Finally, a larger set of assemblers are available for each neural network type:

- Voting Assembler This is the simplest assembling method and works by fitting several base neural network models on the whole dataset. The final prediction is calculated as an arithmetic mean of the individual predictions.
- Bagging Assembler is similar to the Voting Assembler, but now each neural network model is being trained independently on a *different subsample* of the training set [50]. The final prediction is calculated as an arithmetic mean of the predictions of the estimators.
- Fast Geometric Assembler minimizes the training error along a path that connects two points in the space of the neural network weights [51]. The algorithm starts from training one neural network model, then creates copies and train them using a specific cyclical learning rate schedule. It should prevent the neural network models from reaching the same global minimum but does not decrease the accuracy of the ensemble prediction.
- Fusion Assembler aggregates an average output from all base neural network models first. After that, the training loss is computed based on this average output and the target values [52]. Forward and backward propagation is simultaneous for all base neural network models.

[10] https://optuna.org/

- Gradient Boosting Assembler is a sequential ensemble method. At each iteration, the aim of a new base neural network model is to fit the *pseudo target* computed based on the target values and the output from the neural network models fitted before [53].
- Snapshot Assembler uses the non-convexity of neural networks and the ability of optimizers to converge to, and escape from, local minima [54]. The neural network models are trained sequentially such that training of an neural network model is finished in a local minimum. Each time the model converges, its weights are saved, and the corresponding neural network model is added to the ensemble. The learning rate is changing cyclically from a large rate at the beginning, to a small rate near a local minimum.

5.5.2 Generating the testing dataset

The experimental evaluation was performed on a dataset gathered from the real-world application developed by one of the MELODIC project's use case providers. This application, called Genome, is based on the Spark framework[11]. The application combines and compares genome mutations, which requires significant amount of computing power when the processing is in progress. The goal of the application owner is therefore to automatically deploy the Genome application with focus on cost savings but ensuring to finish all computations that contain many Genome mutation simulations in a given time frame. The detailed description was provided in Section 3.3.

Our dataset contains data that was collected in 8 runs of the Genome application on various Cloud configurations. The application ran on Virtual Machines provided by Amazon Web Service. The application was launched two times with the reactive reconfiguration changing the actual worker cardinality, $c_1 \in [1,10]$, and six times on the constant configuration. The number of simulations of Genome mutations was $N = 900$ and the deadline was set to be one hour $T = 3600s$. During each launch, the values of the metrics were collected with $30s$ interval. For each of the time series gathered during one launch, eight predictions were calculated using two forecasting methods, that will be described in details in *D2.2 Implementation of a holistic application monitoring system with QoS prediction capabilities*, Temporal Fusion Transformer (TFT) [55] and NBeats [56]. The prediction horizon was set to $h = 10$ which means that predictions were calculated for t_{k+10} and for each forecasting method four predictions for time points in the future $t_{k+h}, \dots, t_{k+h-4}$ were generated. It gives 64 time series in total.

5.5.3 Analysis of the training set

This section includes the analysis of the dataset generated according to the description provided in Section 6.7.3 for the Genome data farming application described in Section 3.3. The list of features includes 7 metrics, 7 corresponding metric predictions and three other metrics: *Act Worker Cardinality*, *Act Woker Cores*, *Simulation Elapsed Time Raw Metric*.

There are four variables that should be predicted by the solver:

- c_1 Worker Cardinality which represents the number of worker instances,
- provider_Component SparkWorker takes only one value which stems from the fact that so far all experiments were performed on one provider
- c_2 Worker Cores which represents the number of cores available for one Worker instance,
- c_3 Worker Ram variable which represents the amount of RAM available on one Worker instance.

Table 2 presents their basic statistics.

[11] https://spark.apache.org/.

Table 2 Basic statistics of the target variables

	Worker Cardinality	Spark Worker	Worker Cores	Worker Ram [MB]
min	1	0	4.0	16384
median	1	1	4.0	16384
mean	2.93	1.0	5.02	17600.54
max	30	1	16	30720
std	7.01	0.0	3.34	3995.13

However, simple statistics do not provide much information about the dataset and a more detailed analysis was done. First, scatter plots and histograms were analyzed, which give information about the variability of the data from observation to observation, and about the distribution of its values. The example of charts generated for this analysis can be seen in Figure 13 and Figure 14.

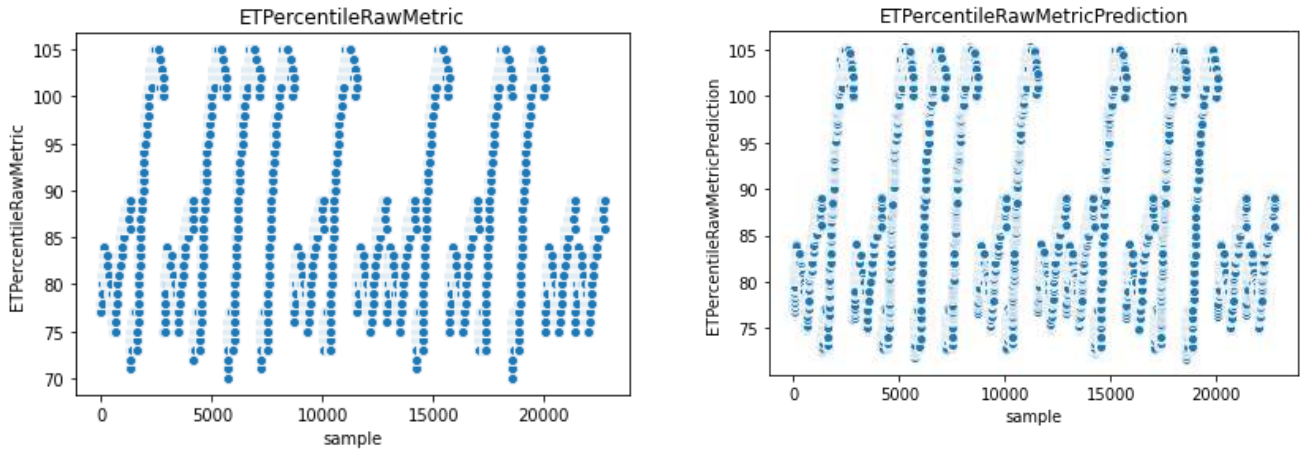

Figure 13 Plot of ETPercentile metric and its prediction for all launches of application

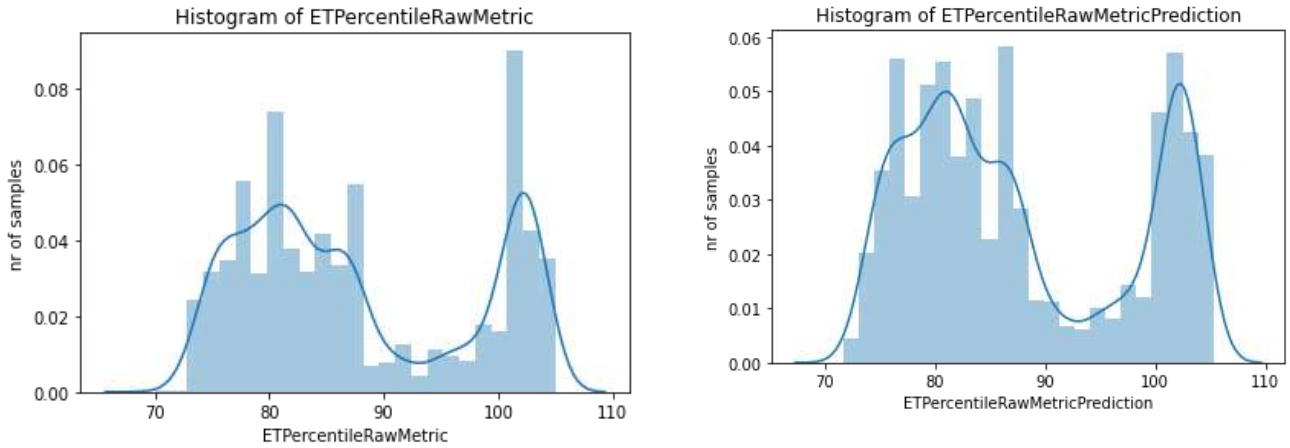

Figure 14 Histogram of ETPercentile metric and its prediction

It was observed that the input variables are periodic, where one period corresponds to one launch of the application. Most of the variables are continuous, except these connected with parameters of Virtual Machines: Minimum Cores, Act Worker Cardinality and Act Worker Cores, which are discrete. Also, we analyzed the target variables. It is worth emphasizing that the target variables are discrete, as opposed to most metrics and metric predictions. Seasonality is observed also for these variables.

Finally, we carefully analyzed data gathered during one launch of the application. The plots below allow us to carefully observe how the data behaves within one period. We omitted variables taking only one value. Both metrics can be easily approximated by a linear function and they seem to be complementary (the larger the first of these metrics is, the smaller is the second one).

In the case of Simulation Left Number Raw Metric in Figure 15 we also observe that two "families" of experiments could be distinguished. The first one corresponds to a linear function with more negative slope (dark line), the second one corresponds to a linear function with less negative slope (light line). Such two families of experiments have been observed for many variables. We believe that they correspond to different configurations of the application.

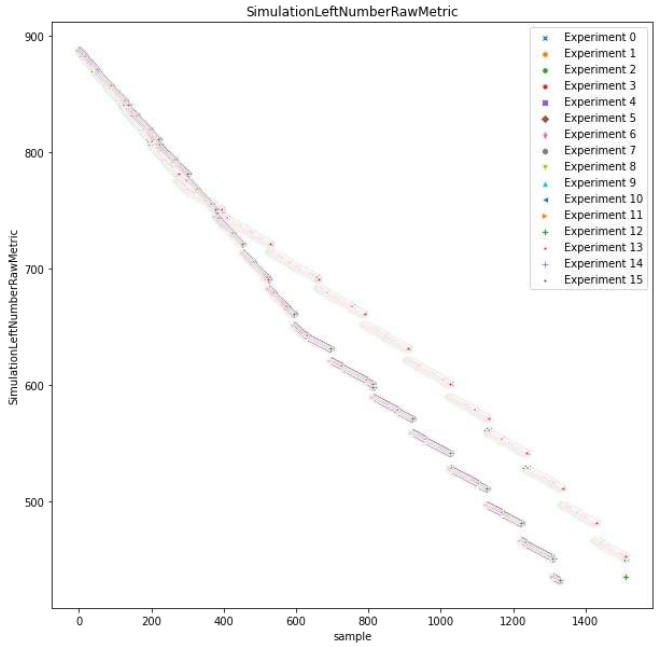

Figure 15 The number of simulations remaining to be computed

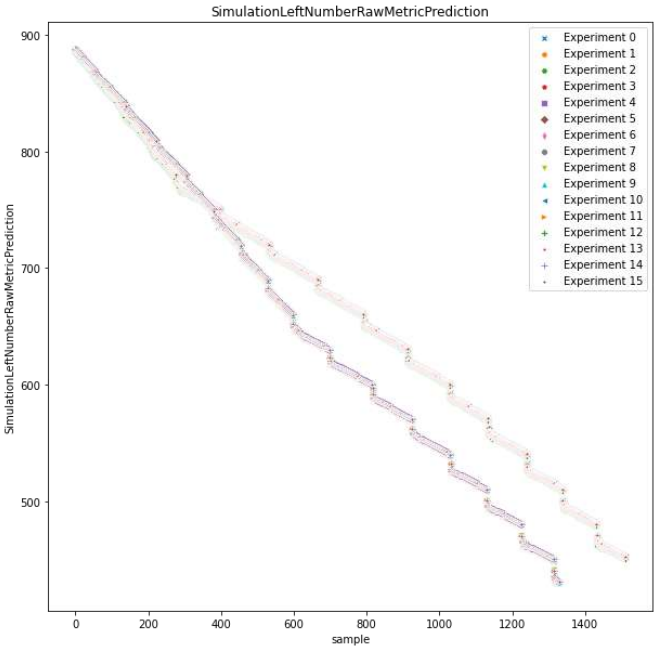

Figure 16 The forecasted number of simulations to be done

The previously seen pattern is reproduced by the trained forecasters as can be seen in Figure 16. Based on the estimated upper bound for a simulation time shown in Figure 18, one can estimate the time remaining as shown in Figure 17.

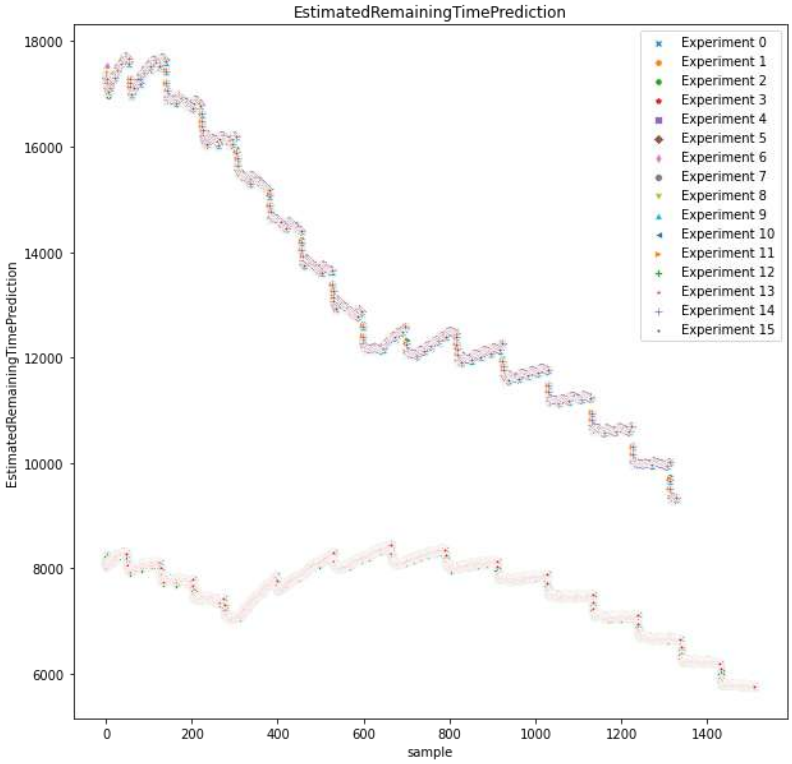

Figure 17 Estimate of the time needed to do the simulations left

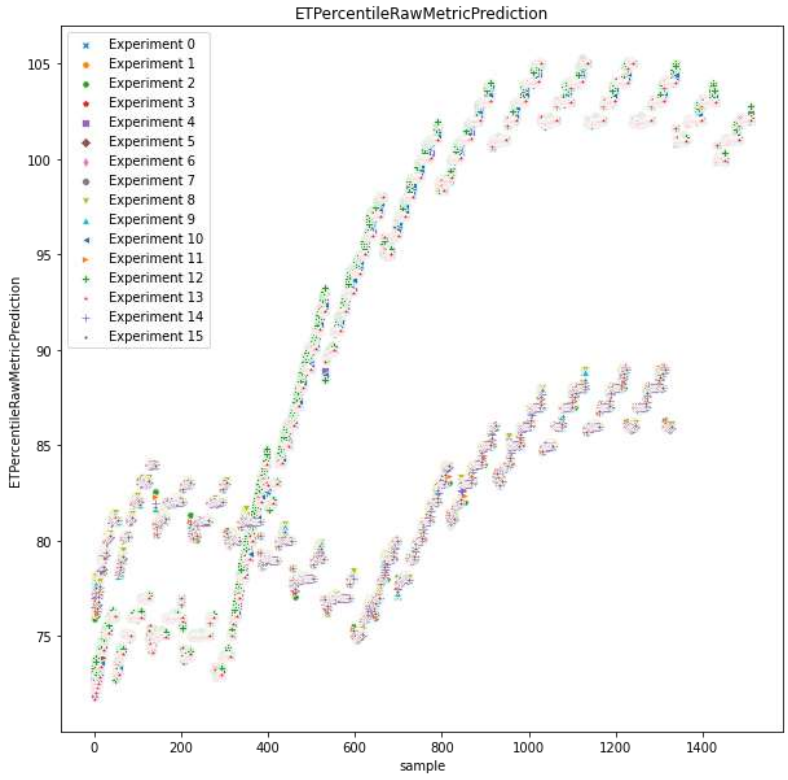

Figure 18 The prediction of the upper bound on a single simulation time

The correlation matrix has been calculated for the variables averaged over the experiments. The elements of the correlation matrix are the Pearson correlation values [57], which we found to be the most appropriate for our case. Discrete variables have been approximated with continuous functions. It can be easily observed that the dataset includes pairs of strongly correlated (both positive and negative correlation) metrics, *e.g.*, Simulation Elapsed Time Raw Metric and Simulation Left Number Raw Metric. This observation might be useful in reducing the size of the input dataset for neural network models.

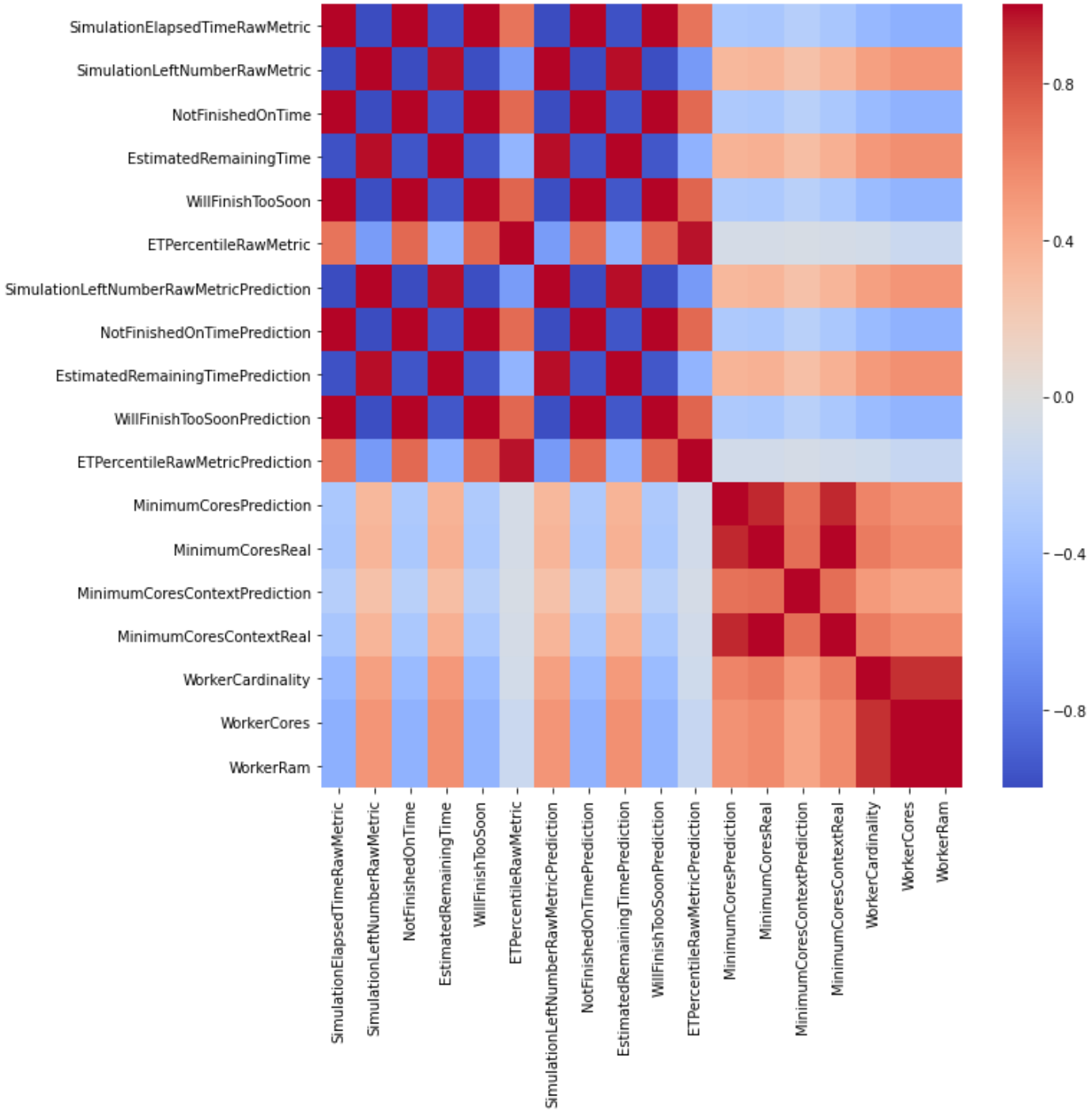

Figure 19 Correlation matrix between input and target variables.

5.5.4 Initial evaluation with the three network architectures

This section includes the results of the experiments with the Supervised Learning solver on the dataset that was described in detail in Section 5.5.3.

The data was split into a training dataset, test dataset and a validation dataset such that distributions of the variables and correlations between them are equal, which is checked by applying the Kolmogov-Smirnoff test [41] and comparing the correlation matrices. The split was made so that the data coming from a single launch of the application is not split, and every dataset includes the data generated for all actual configurations used. Thanks to this input data distributions are very similar for the three datasets.

The main goal of the experiments was to compare evaluation results on the test of the proactive solver with respect to various assembling methods and loss functions. In particular, we wanted to answer the following questions:

1. What neural network type and loss function, with which the model has been trained, do predict the configurations with the highest possible utility?
2. Is it possible to validate the solver using a typically used loss function, e.g. Mean Squared Error (MSE), Symmetric Mean Absolute Error (SMAPE), or Mean Absolute Error (MAE), instead of the utility based loss functions?

The second problem is particularly important as utility is a hard-to-implement, case-dependent function, which might not be differentiable, hence it could not be a loss function, or it might cause a gradient explosion [58] or the gradient is vanishing [46]. Gradient explosion is a situation where the gradient start rapidly to rise to a value outside the range of real numbers during training, and a vanishing gradient means its value start rapidly to decrease to a value very close to zero during training. Furthermore, the utility calculation problem may have a greater computational complexity than the problem of calculating a typical loss function, which again results in longer training.

In this experiment, Pipeline 2 was used. Every training was repeated three times to make sure that the obtained results are reliable. In the first step, we optimized the neural network models and training parameters: the sequence length, $l_{best} = 60$, the scaling method using a Standard Scaler, the model optimizer using the standard Adam optimizer, and the batch size, $b_{best} = 8$. Then we applied grid search with all available assembler types and four ensemble sizes $n = \{1,3,5,7\}$. The forecasting horizon was set to $h = 10$ corresponding to 300s with the sampling rate of 30s. Our benchmark configurations were the configurations computed by the CP Solver.

We trained the neural network models and assemblers using the previously mentioned loss functions, MAE, SMAPE, MSE and UACC defined utility loss. UACC has been implemented as a custom Pytorch metric[12]. UACC is differentiable and it can be used as a proper loss function. However, it is a composition of exponential and fraction functions, which can potentially lead to a gradient explosion. The ratio of finished trainings depends on the loss function. In case of MAE and SMAPE the trainings were not successfully completed due to vanishing gradient problem, for UACC the problems were caused by gradient explosion. For MSE, the percentage of successfully finished trainings was 69.5%, for MAE it was 90.7%, for SMAPE it was 61.1%, and for UACC only 51.9%. We assume that a training was finished in case when the predictions calculated for the test data had no *Not a Number* values.

Optimized Cloud application configurations calculated for the tests set were used for the final evaluation. Figure 8 presents results for three models per training loss function. We selected models whose predicted configuration produced the highest utility on the test set. It should be noted that the UACC function used as the accuracy function was the same as in UACC, but calculated using real data $\theta(t_{k+h})$.

[12] https://torchmetrics.readthedocs.io/en/latest

Table 3 Evaluation results for the best models trained with MSE, MAE, SMAPE, UACC. The presented error values are the mean errors per single configuration.

loss fn	assembler	estimator	n	MAE	SMAPE	UACC	Utility
MSE	Fusion	Transformer	5	0.0778	2e-5	0.0101	0.4899
	Fusion	Transformer	3	0.0712	2e-5	0.0215	0.4785
	Bagging	Transformer	5	0.0665	2e-5	0.0223	0.4769
MAE	Snapshot	Transformer	3	0.0956	2e-5	2.65-08	0.5001
	Snapshot	Transformer	5	0.0956	2e-5	2.65-08	0.5001
	Voting	Transformer	3	0.0956	2e-5	2.65-08	0.5001
SMAPE	Gradient B.	FFN	3	0.8761	0.0001	2.65-08	0.5001
	Fusion	FFN	6	0.0956	0.0001	2.65-08	0.5001
	Gradient B.	Transformer	5	0.0956	0.0001	2.65e-08	0.5001
UACC	Voting	RNN	7	0.2438	7e-5	0.1486	0.3514
	Snapshot	RNN	7	0.8768	0.0001	0.2844	0.2159
	Fusion	RNN	3	0.9621	0.0001	0.2844	0.2159

Results show that MAE and SMAPE are better loss functions than UACC, i.e. they predict configurations with smaller absolute error $\Delta c_i(t_k) = c_i^*(t_k) - \hat{c}_i(t_k)$, $i = 1,2,3$ and relative error $\Delta c_i(t_k)/c_i^*(t_k)$, and with higher accuracy in terms of the utility error. The utility for the estimated optimized configurations is almost the same as the utility calculated for optimized target configurations.

To summarize, the results of the experiments lead us to the conclusion that the proposed method for supervised learning solver gives promising results. The accuracy was measured using the utility function, which is the most important measure for the application owner.

Based on the application states until time t_k and the predictions for time t_{k+h}, the solver is forced to internally consider the prediction error, as its output configurations $\hat{\boldsymbol{c}}(t_{k+h})$ are compared to the optimal ones $\boldsymbol{c}^*(t_{k+h})$ produced by the constraint programming solver. The results clearly showed that the solver can also learn to handle this uncertainty. However, it should be noted that we used the most modern forecasting methods that were optimized for forecasting Cloud applications metrics as the source of the predictions. Therefore, future experiments contain the examination of how big prediction errors the solver can handle.

The solver's internal architecture complexity and the number of models and assemblers used may be surprising. We developed various models to examine which models perform best for Cloud application optimization in general. The goal was to find the best model, assembler, and loss function. Results showed that the utility accuracy can be used as a loss function, but it has some limitations while the standard loss functions achieve similar results in terms of utility. It seems that the solver can be limited to only the Transformer neural network type and the Fusion and the Snapshot assemblers as they get the lowest error on the test set while there is no clear indication about the ensemble size n. However, it should be noted that experiments were performed on a single application with specific metrics and utility function. The solver is further evaluated on more applications, including use case applications, and after this long-term evaluation the final best solver architecture will be confirmed.

The evaluation clearly shows that the error in terms of the difference in the utility is very low so the results can be considered as very good. It is important to notice that these results are the initial evaluation and the final evaluation based on more applications (including use case partners applications) will be reported in *D3.4 Planning and adaptation results*.

5.5.5 Integration with Morphemic

The solver, as based on the neural networks machine learning models, needs additional activities related to model training and periodically retraining. As opposition to stateless solvers, which are stateless and solve optimization

problem based on the parameters passed to them, this solver requires previous training on the historical data for each particular application. It is also different approach then stateful solver, which does not require previous training, but needs to keep and update state continually. The described solver can be used for multiple applications, but for each application separate model needs to be trained and stored.

The abovementioned characteristic of the predictive machine learning solver requires slightly different way of integration with MORPHEMIC platform, especially comparing to stateless solvers. Below is presented the integration of the solver with inner optimization loop, see diagram presented in Figure 20. The integration with outer optimization loop is also possible and will be further investigated.

The training and optimization sequences are presented on Figure 9. The input to the training flow is historical data (metrics and predictions) for a given application. The training flow is as follow:

1. The request to train the model for the predictive machine learning solver is sent from the deployment or reconfiguration process implemented in Camunda via ESB to the Controller.
2. The Controller fetches Node Candidates from Node Candidate cache.
3. The Controller fetches CAMEL model from CDO.
4. The Controller saves initial machine learning model in the Model DB.
5. The Controller invokes Training Data Generator to start preparing training data, as described in section 4.5.2.
6. The Training Data Generator fetches metrics and predictions from Persistence Storage.
7. Based on the historical data, for each time interval the optimal configuration is found by CP Solver.
8. Finally training data are saved to Model DB and proper status is returned to the Controller.
9. The Controller invokes the Supervised Learning Gym to start training of the model
10. The Supervised Learning Gym normalizes and process data and start training of the neural network.
11. The current best trained neural network model is saved in Model DB.
12. After finishing the training, the best neural network model is updated in Model DB and status of training is returned to Controller.
13. The Controller is validating the trained neural network model using test CP model as optimization problem to be solved by invoking the trained neural network model with the parameters of test CP model through Supervised Learning Gym.

The result of the training flow is the trained neural network model which will be used to estimate the best solution for the given parameters in the reconfiguration process and set of the data (metrics, prediction and best configuration for them found by CP solver).

The input to the optimization flow is actual trained neural network model and current values of metrics and predictions. The optimization flow is as follow:

1. To solve the optimization problem for current configuration the Controller is invoked by the Metasolver. The Metasolver passes current values of metrics and predictions to the Controller.
2. The Controller invokes Training Data Generator to prepare input to the model, based on current values of metrics and predictions and historical data.
3. The Training Data Generator prepares data to be provided to actual neural network model to obtain solution.
4. The Controller passed the prepared data to Supervised Learning Gym.
5. The Supervised Learning Gym fetches actual neural network model and invoke it with prepared data.
6. Neural network model returns the solution for the given data.
7. The solution is returned to Controller and then to Metasolver.

The outcome of optimization flow is the estimated best configuration for the current values of metrics and predictions. This configuration should be used by MORPHEMIC platform to initially deploy or reconfigure configuration of deployed application.

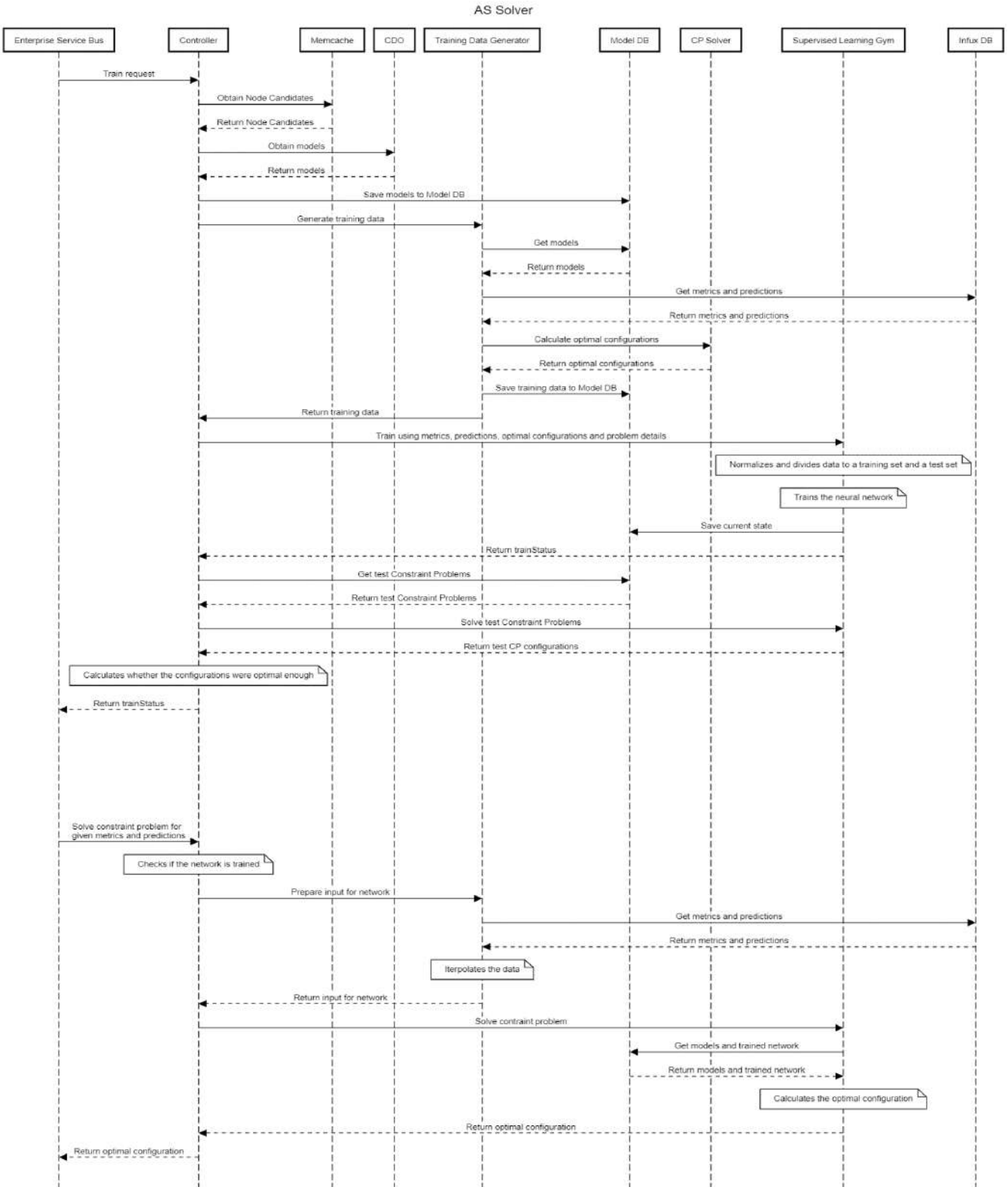

Figure 20 Sequence diagrams for integration with MORPHEMIC

5.6 Adaptive Dynamic Programming

As argued in deliverable *D2.3 Proactive utility - Framework and approach*, the current way of maximizing the utility function for the given context will prevent forecasting as the configuration and the application's execution context are closely bound under the optimization problem. It is therefore necessary to consider the system state $\boldsymbol{s}(t_k) = [\boldsymbol{c^T}(t_k), \boldsymbol{\theta^T}(t_k)]^T$ as one vector moving from state to state driven by either the *monitoring events* when a new value of one of the monitored metrics is updated or as *control events* when any element in the configuration vector changes. Like Equation (12) one may then write the *discrete system update equation* as

$$\boldsymbol{s}(t_{k+1}) = \boldsymbol{f}\big(\boldsymbol{s}(t_k), \boldsymbol{u}(t_k)\big) \tag{13}$$

where $\boldsymbol{u}(t_k)$ is the multivariate control signal applied to the system at time t_k.

An important observation can be made at this point: All the optimization approaches discussed so far aim for maximizing the utility of *only* the next configuration. This is true regardless of a *reactive optimization* finding the best configuration for a newly observed application execution context vector, or a *proactive optimization* approach finding the best configuration for the time point of a predicted future application execution context. As the application execution context changes with every new monitoring measurement arriving from the running application, the reactive solution will be outdated when it is deployed as many monitoring events may have happened while the solver searched for a solution. The proactive optimization will equally ignore all the monitoring events occurring before the time point for which it predicts the application execution context. The two optimization approaches are only looking at one single time point of the event history when finding the best configuration to deploy. Hence, none of the discussed approaches tries to maximize the utility over a time horizon. In contrast, the *Dynamic Programming* approaches discussed in this section will try to maximize the total utility of the application over a time horizon.

Since we want to be able to have the resources available for the application when they are needed some time in the future, we need to apply the control over some time horizon, like the concept of model predictive control described above in Section 5.3: Using the model of Equation (13) we can iterate the state into the future for a given control history. Optimality implies that we want to find the control signal so that the total utility is maximized. However, the far future is more uncertain than the near future even for the best state forecasters, and one may apply a discount factor to the future utilities depending on how certain the state prediction is. If we know the optimal actions to take, $\boldsymbol{u}^*(t_k)$, for all the time points $t_k, t_{k+1}, t_{k+2}, \ldots$ then the optimal value function being the total discounted utility over time

$$\begin{aligned} V^*\big(\boldsymbol{s}(t_k)\big) &= \max_{\boldsymbol{u}} \sum_{i=k}^{\infty} \gamma^{i-k} U\big(\boldsymbol{s}(t_i), \boldsymbol{u}^*(t_i)\big) \\ &= \max_{\boldsymbol{u}} \left[U\big(\boldsymbol{s}(t_k), \boldsymbol{u}^*(t_k)\big) + \sum_{i=k+1}^{\infty} \gamma^{i-k} U\big(\boldsymbol{s}(t_i), \boldsymbol{u}^*(t_i)\big) \right] \\ &= \max_{\boldsymbol{u}} \big[U\big(\boldsymbol{s}(t_k), \boldsymbol{u}^*(t_k)\big) + \gamma V^*\big(\boldsymbol{s}(t_{k+1})\big) \big] \\ &= \max_{\boldsymbol{u}} \left[U\big(\boldsymbol{s}(t_k), \boldsymbol{u}^*(t_k)\big) + \gamma V^*\Big(\boldsymbol{f}\big(\boldsymbol{s}(t_k), \boldsymbol{u}^*(t_k)\big)\Big) \right] \end{aligned} \tag{14}$$

This last equation is called the *Bellman equation* after the inventor of dynamic programming. Unfortunately, solving this equation is exponential in the number of states and possible control actions since it is necessary to evaluate value function for all combinations of states and control actions to find the optimal control actions. There are three unknowns in this equation:

1. The system model $\boldsymbol{f}\big(\boldsymbol{s}(t_k), \boldsymbol{u}(t_k)\big)$
2. The optimal value function $V^*\big(\boldsymbol{s}(t_k)\big)$
3. The optimal control signal sequence $\boldsymbol{u}^*(t_i)$

The learning of these three signals will be discussed in the subsections below.

5.6.1 Learning the system model

The brute force approach to obtaining a system model is to generate a set of feasible states and possible control signals in those states based on the system constraints, apply these to the system and for each pair of state and control, and observe the resulting system state. Then one has the arguments to the system model function and the outcome in form of the next resulting state of Equation (13), and then one may use regression methods to infer the system function mapping. The trained regressor can then be used to map new, unknown states and proposed output to new state vectors quickly.

With respect to an implementation of the Cloud deployment it means that one must select randomly legal values for the application's execution context and then solve the optimization problem to obtain the optimal configuration for this context, which gives the full state vector. The goal should be to sample the execution context space as uniformly as possible. Then one can mix and match the state vectors, say $\boldsymbol{s}_{(\mathbf{0})}(t_k) = \left[\boldsymbol{c}_{(0)}^T(t_k), \boldsymbol{\theta}_{(0)}^T(t_k)\right]^T$ and $\boldsymbol{s}_{(\mathbf{1})}(t_k) = \left[\boldsymbol{c}_{(1)}^T(t_k), \boldsymbol{\theta}_{(1)}^T(t_k)\right]^T$ where $\boldsymbol{\theta}_{(0)}(t_k) \neq \boldsymbol{\theta}_{(1)}(t_k)$: One can select the metrics defining the execution context from one vector and a "current application configuration" from another, $\boldsymbol{s}_{(\mathbf{2})}(t_k) = \left[\boldsymbol{c}_{(0)}^T(t_k), \boldsymbol{\theta}_{(1)}^T(t_k)\right]^T$. The selected "current" configuration, $\boldsymbol{c}_{(0)}(t_k)$, is obviously not optimal for the selected context $\boldsymbol{\theta}_{(1)}(t_k)$, but one knows the optimal configuration for the selected execution context from the selected full state vector from which the metric values were taken. The optimal control signal is then some transformed difference between the known optimal configuration for the selected metric values and the configuration selected as the "current", $\boldsymbol{u}_{(0)\to(1)}^*(t_k) = \boldsymbol{c}_{(1)}(t_k) - \boldsymbol{c}_{(0)}(t_k)$. The difference holds only if the control signal is added to the configuration, which may not be the case as the control signal can be used in many ways to decide how to move from one configuration to the next. After finding the required control signal, one has the "current" state vector consisting of the current application configuration and the selected context, the optimal control signal, and the future state vector consisting of the selected context and the configuration already found to be optimal for the selected context. One may then learn the system regression function from these data triples.

This method is computationally expensive as one may need a significant number of different state vectors to obtain a good regression model, and for each state vector one must solve the optimisation problem. However, for each new state vector the training set will grow exponentially because the context part of the state vector can be matched against all the application configurations of the already known state vectors, and the new configuration can be matched one by one against the context parts of all the already known state vectors. Eventually, the size of the data set can become problematic before one is able to learn sufficiently well the system regression function.

The alternative is to learn the system model recursively as the configurations are applied. Assuming that the measurements are frequent enough for the model of Equation (13) to be well approximated by its linearization, one may use recursive least squares estimates of the system state [59]. For non-linear systems one may use extended variants of Kálmán filters [60] or kernel adaptive filtering [61]. The approach would be that one initially solves the system normally using a traditional solver but starts a solver whenever a new measurement of the application's execution context arrives independent of the optimal configuration for the previous execution context has been found. As soon as a solver returns the optimal configuration for a context, then the configuration, the context and the control signal being the delta from the current context to the found optimal context are fed as the state and the control signal to the recursive estimator. Given that the system model is an invariant mapping from the current system state and the control signal to the next system state, one may assume that the recursive model estimator will converge. Thus, at one point one may stop the recursive updates of the model and just use the identified static mapping as the system model.

Even if the system model can be learned by these approaches, it takes the control signal as input, and hence it is crucial to derive a good model for the control signal based on the observed change in the application's execution context. This is a *performance model*, and a first version of this is discussed in Section 8 below.

5.6.2 Learning the value function

The Bellman equation (14) is a one-step move at time t_k taking the system state from the current state to the next optimal state by the optimal reconfiguration action $\boldsymbol{u}^*(t_k)$. However, the next optimal state will be unknown since the

measurements at time t_{k+1} is a part of the next state vector, and since some of these measurements may sample continuous values, there are infinitely many next states. One must therefore approximate the solution of the value function in the Bellman equation with a parameterized function of the known configuration decision variables and possibly other discrete metric values. It is therefore necessary *to learn* the value function based on feedback from the most frequently visited states, and this is the core idea of *Reinforcement Learning* (RL) [30]. A fundamental approach from reinforcement learning to solve the Bellman equation is to consider what is called *value iteration*: One start with an initial guess $V_0(\boldsymbol{s})$ of the value of the value function for all possible states $\boldsymbol{s}$ without considering when the state is visited. From a starting state $\boldsymbol{s}_0$ one will find the control signal that moves the system to the next state $\boldsymbol{s}_1$ with a feasible control signal so that the value of equation (14) is maximized. Then $V_1(\boldsymbol{s_0})$ is updated to this new maximal value. This maximisation process is then repeated with all states as starting states producing updated values $V_1(\boldsymbol{s})$ for all states, and the process can start over. It can be proven that $V^*(\boldsymbol{s}) = \lim_{n\to\infty} V_n(\boldsymbol{s})$ and in the process one has identified the optimal action for each state, *i.e.*, the control signal $\boldsymbol{u}^*(\boldsymbol{s})$.

The problem is evident: one need to visit iteratively all possible states until the iterative value assignment converges to a solution that is close enough to the optimal value function for all states. The sheer size of the state space prohibits this approach. However, this is where the system model (13) comes to our help as it is not necessary to enumerate all states since the next state reached by applying a control signal is given by the system model. The idea is then to start with a set of randomly chosen representative monitoring values and solve the optimisation problem for these to form a smaller set of system states for which the value functions will be found. The value iterations for a given state $\boldsymbol{s}$ is then the alternating sequence of iterative updates of the control signal and the value function

$$V_i(\boldsymbol{s}) = U\big(\boldsymbol{s}, \mathbf{u}_{i-1}(\boldsymbol{s})\big) + V_{i-1}\Big(f\big(\boldsymbol{s}, \mathbf{u}_{i-1}(\boldsymbol{s})\big)\Big) \tag{15a}$$

where the control signal is found for each iteration by solving the related optimisation problem

$$\boldsymbol{u}_i(s) = \arg\max_{\boldsymbol{\nu}}\big\{U(\boldsymbol{s}, \boldsymbol{\nu}) + V_i\big(f(\boldsymbol{s}, \boldsymbol{\nu})\big)\big\} \tag{15b}$$

These two equations constitute the *inner loop* for updating the value function and the control signal and it is performed inside an *outer loop* sweeping over all selected states $\boldsymbol{s}$. Although conceptually simple, there is no guarantee that this iteration sequence will eventually converge to the true value function and corresponding control signal. Liu *et al.* prove that the bound on the approximation error between the true value function and the optimal value function is proportional to the number of iterations done in the above equations ([62] Lemma 3.3.1), and they propose several algorithms to keep the approximation error small and bounded [62]. Their best algorithm seems to be the General Value Iteration ADP algorithm that should be further evaluated for use in MORPHEMIC. The main idea behind this algorithm to add a functional representation of the approximation error to each of the iterations (15a) and (15b) as penalty functions and compensate for these.

Given that the value functions can only be found iteratively for a limited number of states, there is a need to generalize the values function to give values to unseen states. Liu *et al.* proposes a Back Propagating Neural Network (BPNN) with two layers for the value iteration (15a), and they train the weights of this network recursively with the iterations of the inner loop, *i.e.*, the value iterations, using the hyperbolic tangent as the activation function [62]. They call this network the 'critic network' where this label probably emphasises the connection to the adaptive critic discussed in Section 5.4 above where the reward, *i.e.*, the value, is only known after multiple plays in the iterative game.

5.6.3 Learning the control sequence

The optimal control signals of the selected subset of states are found by solving the optimisation problem (15b). To approximate the control signal of an unknown state, Liu *et al.* proposes again a BPNN with the same structure as the BPNN for the value function and trained recursively. Owing to the close connection to the reinforcement learning, this network is labelled the 'action network'.

Alternatively, one may try to learn the control sequence directly. This is known as *policy learning*, and Wei *et al.* proposes a Q-learning method for the case when the far-future reward discount factor $\gamma = 1$ [63]. They also propose a model-free multi-objective Q-learning method under a quadratic utility function.

6 Workflow analyser

6.1 The problem

In Service Oriented Architecture (SOA), a single application should be composed by many independently deployable smaller components. Each component may have different services that can communicate with others using different ways, such as REST API for example.

Each service in an SOA embodies the code and data integrations required to execute a complete, discrete function. The service interfaces provide loose coupling, in other words, they are designed as self-contained components that can be called with little or no knowledge of how the integration is implemented underneath. Henceforth, SOA defines a way to make the components reusable more effectively, thus reducing development time and the associated cost.

ProActive Workflow implements the concept of the SOA and divides a single application (also called a job) into multiple smaller tasks. Each task may be completely independent from the rest and may require or provide a certain service(s) from or to another task(s). Moreover, ProActive controls the workflow using a scheduler in charge of enacting and following up on the progress of the determined management actions. The main goal of ProActive is to maximize performance by binding computations and data location.

6.2 Workflow scheduling

The workflow scheduling is responsible for the control and the dispatching of the execution of a certain process over a finite set of computing resources: this distribution should align with high-level objectives, such as the minimization of the execution duration of the process. Nevertheless, the set of available computing resources is process specific as some processes require a specific type of resource, e.g., machine learning process that requires a node with a GPU. In this section we introduce the concept of a workflow model and its execution, then we present the computing resource management and monitoring and finally we present how the integration of the Scheduler with Morphemic is achieved.

The first step consists of defining the application model using CAMEL language where the user can define the application architecture, the deployment requirements and the resource requirements. Once the application is defined and modelled, the next step consists of transforming it into a workflow model by extracting the needed information from the CAMEL model. This transformation is two-folded: first the CAMEL model is sent in a Json format to the Scheduling Abstraction Layer (SAL), then the latter will generate the WF and submit it to the ProActive Scheduler (Process is detailed in the next paragraphs).

In ProActive, a workload requiring scheduling is defined as a workflow model. It is a directed graph composed of tasks (apexes) representing atomic processing and execution control flows (edges). Figure 21 shows an example.

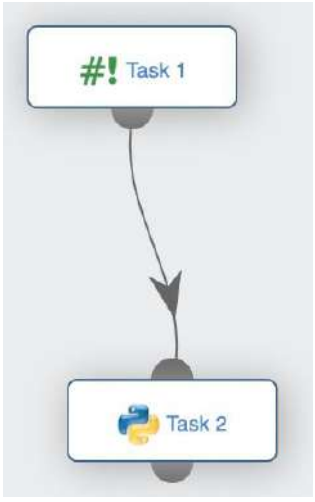

Figure 21 Simple workflow example

In this example, Task 2, which is a Python task, depends on Task 1, which is shell script task. Thus, Task 2 will be executed after Task 1 and will be able to access Task 1's results as output data. Each task of the workflow is specified following a set of properties that can be found in the deliverable *D4.1: Architecture of pre-processor and proactive reconfiguration*.

For a CAMEL component model, for which a virtual machine (VM) is acquired, and on which an application will be installed and started, the SAL creates 4 sequential dependent tasks in which we are the following:

The first task is responsible for acquiring the VM. After this task is finished, the next one is responsible of preparing the needed infrastructure and defining the environment variables that are used when the application will start. The third task handles the installation of the components by executing the needed pre-scripts, and finally, once this task is finished, the last one starts the execution of the application.

The dependencies between the application's components are extracted from the communication definition part in the CAMEL model. The following use case shows an example.

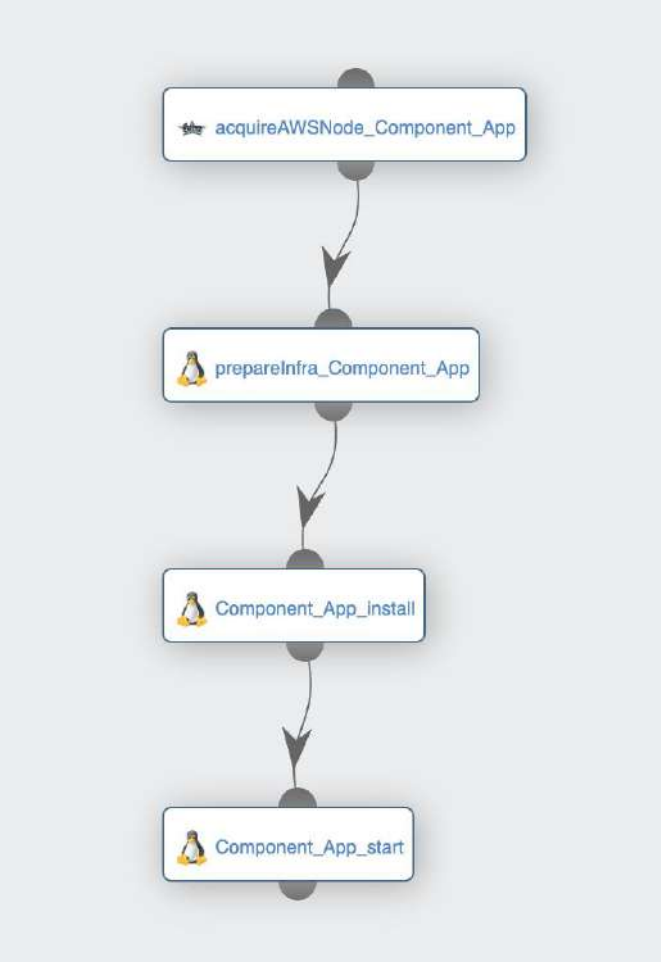

Figure 22 A CAMEL component deployment sub-workflow for deployment in a Virtual Machine

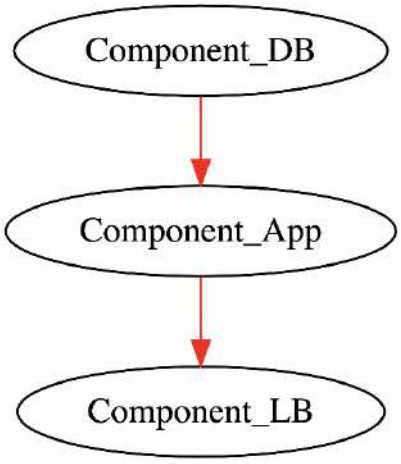

Figure 23 Example application's dot graph

Figure 23 shows the dot graph of an application consisting of 3 different components. The graph is obtained from the SAL endpoint that transforms a submitted WF into its corresponding dot graph format. Based on this use case, "Component_App" must be executed after "Component_DB" and depends on the latter's results. The same dependency exists between "Component_App" and "Component_LB". In order to determine the dependencies, prior to the application submission to the scheduler, the system uses the "link" definition in the CAMEL model. For example, regarding this use case, Figure 24 shows what is defined inside the CAMEL model to create the dependencies between the 3 components:

```
link DB_App from App.AppReqPort to DB.DBProvPort req ReqModel.DB_App

link App_LB from LB.LBReqPort to App.AppProvPort req ReqModel.App_LB
```

Figure 24 The communication definition using CAMEL for the components from Figure 23

Based on the previous link definition, "Component_App" is requiring the port that is provided by "Component_DB" and "Component_LB" is requiring the port that is provided by "Component_App". This explains the dependencies showing on the dot graph (represented by the red arrows).

6.3 Optimized resources for workflow execution

The preceding discussion indicates that it should be possible to use the connectivity definitions in the CAMEL application model together with the SAL to decide on the order of execution of the components, and their shared dependencies on data components. This opens for two optimizations that can be done in the MORPHEMIC outer loop before the application is submitted for deployment:

8. Application computing components using the same application data storage component should be bound to be executed on the same Cloud provider. If the data storage component is bound to data existing prior to deploying the application, this implies that the computing components will be moved to where the data recedes. This will reduce the cost of accessing the data needed by the computing components, and it will hopefully also reduce the latency in reding and writing data.
9. Application computing components with clear output-input data dependencies must execute in sequence. One can therefore bundle such sub-sequences of the application's workflow onto the same virtual machines. In practice, the MORPHEMIC pre-processor will create a *super component* in the CAMEL model and compute the resource requirements for each resource of this component as the maximum requirement of the requirements from any of the grouped application components.

The idea behind the second optimization is that multiple application components may share the same virtual machine (VM), and thereby save cost by avoiding starting individual VMs for each component in the workflow sub-sequence.

These VMs will for parts of the workflow sub-sequence execution be idle waiting for upstream components to get the input data ready or be idle waiting for the workflow completion after having delivered the data to the downstream component.

Based on this observation, one can implement the grouping approach even if there are no explicit application component workflow indicated in the CAMEL model by inserting a clock sensor for each application component that register when the component start execution and when it stops. One can start with the default deployment with one VM for each application component and based on the information about the start time and end time of each component one may learn statistically the execution sequence, and hence optimize the CAMEL model to adapt the deployment by grouping components according to their execution sub-sequences.

The big benefit of sub-sequence grouping as super components is in the run-time adaptation since there will be less components to consider in the application configuration. Given that the optimization problem has exponential complexity, reducing the number of decision variables in the application's configuration will have a significant impact on the time needed to find the configuration that is optimal for the application's current execution context.

This grouping is only valid if there are Cloud resources, *i.e.*, virtual machines, capable of hosting the group. It is expected that a node candidate will be available for a group since the requirements of the super component is the maximum of the requirements of the individual components in the group, and therefore if the individual components can be hosted by available node candidates, so can the group. However, it could be that the maximum in different dimensions cannot jointly be met by a single node candidate, even though resources offered by VMs normally grow jointly in all dimensions. Thus, the workflow set will be filtered to verify that there are node candidates also for the 'super-components'.

If the result of this filtering is an empty set, the grouping must be re-done with additional constraints imposed on the joint resource requirements resulting in two or more smaller super components corresponding to subsets of the components in the workflow sub-sequence. This task can be seen as a *bin-packing problem* where one tries to minimize the number of VMs used for hosting the given set of components, *i.e.*, minimize the bins to pack. There are many algorithms that can be used to approximate the optimal packing [64]. Bansal *et al.* have developed the currently best algorithm for homogeneous VMs where the aim is to pack the components on as few VMs as possible [65]. To our knowledge, the best algorithm for *heterogeneous* VMs that also allows associating a cost to the various bin types available, *i.e.*, VMs, with the goal of minimising the total cost of the bins is the approximation scheme due to Patt-Shamir and Rawitz [66]. Further evaluations are needed to evaluate the efficiency of this approach against the size of the node candidate set.

One may alternatively consider the optimization as a knapsack problem [67], or more specifically as a multidimensional knapsack problem given that the components have resource requirements in multiple dimensions. There are many heuristics proposed for this kind of problems [68]. However, the problem at hand is known as a Multiple Multidimensional Knapsack Problem (MMKP) since it is necessary to pack all the VMs at the same time. This variant of the knapsack problem has received little attention, but Yi and Cai have proposed a polynomial time approximation [69], and Song *et al.* have, to our knowledge, developed the only exact algorithm [70]. The use of MMKP for datacentre management has been demonstrated by Camati *et al.* who evaluated several MMKP heuristics for allocating VM to homogeneous servers [71]. The problem can be relaxed by considering each VM as an individual knapsack that is assigned component instances, and only one of each type. This corresponds to the Multiple-Choice Knapsack Problem (MCMKP), and the best polynomial time approximation scheme is due to He *et al.* [72]. The issue with using MCMKP in this way is that there is no global optimization of the goodness of the assignment and trying to maximise all the packings at the same time leads to a Multi-Objective Multidimensional Knapsack Problem (MOMKP) [73]. However, only problems with at most three objectives have been considered with heuristic approaches exist, and finding a solution is closely related to the Pareto front of the utility optimization problem [9].

The main difference between heterogenous bin-packing and knapsack approaches is that the latter tries to optimize the packing of a given set of virtual machines. Ideally, to find the best node candidates to use, one must solve the knapsack problem for all subsets of the power set of the node candidate set. This may render the knapsack approaches infeasible for realistically sized problems. Fortunately, there is an alternative to bin packing as Voß and Lalla-Ruiz have shown

that the MCMKP can be reformulated as a set partitioning problem [74]. This latter problem is well studied in the field of game theory where is it known as Coalition Structure Generation (CSG).

There are many algorithms for the optimal CSG and the survey by Rahwan *et al.* describes various approaches in this area [75]. Currently, the fastest optimal algorithm is called Optimal Dynamic Programming – Integer-Partition (ODP-IP) [76]. The complexity of this algorithm is $O(3^n)$, and it is considered as a state-of-the-art algorithm. For a larger number of agents, it may be not possible to run an optimal algorithm because of its exponential complexity. This was the motivation for anytime algorithms, such as the Integer-Partition (IP) [77]. They are searching the space in a specific order to provide a solution with expected quality for a given deadline. Michalak *et al.* proposed the distributed version of this algorithm [78]. An issue with the CSG algorithms is that it does not take constraints on the coalitions into consideration, and further research is needed to see if it is possible to extend the algorithms including constraints.

7 Application model and resource optimization

A polymorphic application is an application that can have different forms due to the alternative configuration types that its components can have. Thus, the main goal of the Morphemic platform would be to select the right application form based on the current application requirements and context. Application requirements usually involve overall or component-specific SLO objectives as well as various kinds of constraints for the application components like resource and scaling constraints, per each configuration type of a component. However, users might not supply precise constraints as they might not have the right knowledge to do so. For instance, they might neglect providing upper bounds on resources or they might supply too strict bounds. In the first case, as there is no upper bound, there is a risk that the application cost can get too high by allocating expensive resources. Furthermore, the performance benefit might not be as high as expected from utilising such resources. This is due to the fact that in many cases, the addition of further resource quantities from a specific bound and on might not have a significant positive effect on performance. In the second case, the performance improvement benefits might be lost as it is not possible to go beyond the limits given by the user. In this respect, a first added-value feature of the Morphemic platform lies on updating the user bounds on resources in order to become more precise. This is addressed in section 5.1, where a specific approach to tune the requirements of the application components is proposed, while, architecturally, such an approach perfectly matches the main, envisioned capability of the Vertical Parameter Tuning component.

An application might not be defined as polymorphic from the very beginning. As the development effort for application components is raised due to the need to introduce additional configuration types for them. Thus, it is expected that Cloud applications might be either non-polymorphic at all or they might have a low degree of polymorphism. Further, it is expected that most of the Cloud applications might not be able to exploit the performance benefits from the use of accelerated resources as they might not have the expertise to produce the right application code mapping to accelerated configuration types. Finally, it is probable, where the probability depends on the capabilities of the application owner, that the architecture of the Cloud application is rather incomplete based on the domain and functionality that it delivers.

Due to the aforementioned inefficiencies, section 5.2 introduces an innovative approach towards extending the CAMEL model of Cloud application through:

a. finding missing functional pieces from the current architecture of Cloud applications and suggesting their inclusion to the application DevOps engineers;
b. discovering application parts which can certainly benefit from being deployed in hardware accelerated resources.

Both offerings rely on the incorporation of the Application Profiler module in the MORPHEMIC platform architecture which enables to crawl and classify open-source components from well-known forges and meta-forges. Through such classification, it is then possible to identify potential “template” application architectures in application domains to support the functionality of the first offering as well as to discover new configuration types, especially hardware-accelerated ones, for certain application components based on functionally matching open-source components to support the functionality of the second offerings. Through the incorporation of both offerings, it is then possible to deliver a new, added-value feature of the MORPHEMIC platform which is both innovative as well as highly-required and enables to truly optimise both an application’s architectural model and its allocated resource configuration.

7.1 Tuning the requirements of the application components

There are requirements set in CAMEL for all components, like the range of a number of cores or the range of memory needed for the component to run as best possible. Typically, the developer or application owner have no clue about what the really good values for the upper and lower bounds for these ranges are. The bounds will, in many cases, also depend on what data the application is processing, the data structures used by the respective component, etc.

Thus, the idea is that the application utility and possibly the component performance can be improved by increasing or decreasing the amount of resources required by the component, *i.e.*, by changing these bounds. We opt mainly on the increase in the bounds of the various kinds of resources required by the application with the rationale that the increase of such resources can positively influence the performance of the application. We acknowledge the fact that such an increase can influence the deployment cost, but this is something that can be controlled through the use of the utility function and potentially through the use of a requirement over such a cost. We should also stress that it might be dangerous to modify the lower bounds on the resources with the following rationale:

a. The DevOps engineers are more confident with respect to set lower bounds instead of upper bounds;
b. lower bounds can be checked easily to assess their impact on the performance and proper functioning of application components;
c. changing lower bounds on resources increases the risk that a component might not have the right amount of resources to function properly – so this can jeopardise the execution of the whole application.

The way this idea can be implemented is twofold. First, we can increase the upper bounds by one or two points, especially in case of the initial deployment of the application. For instance, if we have an upper bound on the number of cores, we can increase it by two, *e.g.*, from 3 cores to 5. On the other hand, if we have an upper bound on the main memory size, this can be increased by a factor of two, *e.g.*, from 0.5 GB to 2GB with a factor of two – a usual factor for the memory size as it can be seen in provider offerings. An alternative setting is to just leave the upper bounds as open in this case. This can drive the selection of the resources by considering the lower bounds as strict constraints that influence the proper application functioning, plus budget constraints and the utility function. As such, we can have the greatest possible flexibility in selecting the right amount of resources. Thus, this alternative setting seems more suitable to be followed compared to the first one.

Second, once performance models have been derived by the Performance Module, we can then check changing the bounds in both directions so as to assess the potential of improving the utility function and the respective SLOs. The rationale is that if we see that the effect on following one of the directions, *i.e.*, increasing or decreasing, is not positive, then we can identify the respective stop point or limit. This could be conducted easily through simulation where we need to vary:

a. the lower and upper bounds of the resources;
b. the weights in the components of the utility function to simulate different cases where performance or cost need to prevail while take into account the performance models that have been derived by the Performance Module.

Owing to the fact that the performance models can change over time to become more precise, this tuning process has to be repeated multiple times. In addition, it has to be conducted for each architecture application variant that is currently on effect.

Interestingly, the simulation can actually unveil knowledge which can be utilised to direct the reasoning process followed by the Solvers. Such a knowledge could take various forms like:

I. particular resource limits for different situations with respect to the actual content of the utility function in terms of weights;
II. potentially some conditions about combinations of resource limits – this can be possible in the sense that it can be found that going beyond a specific limit for one kind of resource could be allowable only conditionally with respect to the limits of the other resources. Such insights can lead to updating the CP model to include new constraints that guide the way the resources can be selected.

They can also influence the way the reasoning process is actually conducted. In particular, currently, the fetching of node candidates is conducted just once during the initial deployment process. However, in the context of the continuously changing utility function, the set of matching node candidates might require updating as new node candidates might require to be introduced and existing ones might need to be removed. Thus, an update mechanism on the node candidate selection needs to be incorporated and executed each time in the deployment reasoning process.

7.2 Architecture changes by model templates

Different types of applications may have different architectural patterns and deployment patterns. Collectively these patterns can be seen as model templates. The idea is that after the application profiling, then the right patterns can be selected. It is then a matter of partial matching (homomorphism) of the graph of the template model with the application component graph structure. Mismatches between the two can serve as points of improving the application deployment model by changing the application architecture. Consider for example a 'web application' with one 'web server component' the user model specifies that this server can be multiplied depending on the number of users, but the template model indicate that a load balancer may be needed if server cardinality is larger than one. Thus, the model templates for this application type suggest including a load balancer as an architectural change.

The idea is that we should explore the fields of inexact graph matching between the actual architecture model in CAMEL given by the user and the template model pattern. Based on that matching, architectural changes can be proposed to the DevOps engineers in order to improve the application. Such changes might require adding components to specific parts of the application architecture as well as removing existing components. They could also indicate deployment changes in the sense that the deployment configuration for existing application components might need to be changed. As such, it is evident that the suggestions to be produced need to work over different kinds of templates that might require a different handling in order to be generated as well as checked over the existing application deployment architecture. However, the overall denominator should be the same: the search space would need to be reduced over a specific domain and/or the general domain incorporating patterns which are generally seen across different domains.

7.2.1 Architectural patterns

Architectural patterns need to be derived in the context of a specific domain by operating over the profiling knowledge that has been produced in that domain. The focus should be on real applications that comprise multiple components with communication dependencies and not just single components. While the analysis should be driven by the functional classification of those components. The analysis would require identifying clusters of components that seem to co-exist across multiple applications. These could be the actual patterns to search for which could be selected by imposing a specific threshold over the percentage of applications that exhibit them, *e.g.*, 10% of all applications or 40% over the applications that include at least one component of the pattern – maybe second bound is more relevant but even a combination of such bound could make sense.

The analysis of the profiling knowledge space for deriving the architectural patterns could follow an iterative approach starting with 2 components and increasing the number of components by one for the next iteration. A stopping condition would be that no new patterns are derived for a specific iteration. In each iteration, the task to perform is easy: by considering a unique functional list of components, we check whether *N* of those components are included in the crawled applications. A unique set of component combinations can be derived which can then be filtered based on the aforementioned threshold. Interestingly, it is possible that a combination in the current round can “absorb” a combination in a previous round. In that case, we add all combinations that have been found in the current round in a global set and remove from that set the combinations absorbed from previous rounds. The final set eventually includes all the architectural patterns that have been discovered in that domain.

The matching can then be easily performed by considering all the discovered patterns with the application architecture. The goal would be to find those patterns which maximally match with different parts of the application architecture. Maximally matching would map to finding those patterns for which most components match with those of the application based on the following fair metric: number of matches / number of components in the pattern. From the produced list, we can filter out those patterns which are already fully covered by the application architecture. The

remaining patterns can then be sorted based on the number of changes need to be performed over the application architecture. This sorted list of matching patterns can then be returned to the DevOps engineer.

It must be indicated that it is possible that the sorting could also take into account the number of application components which have been already included in fully matching patterns. The rationale is that application components that participate in fully matching patterns might not be relevant to be included, especially in combination, in other partially matching patterns. Thus, this can ensure that we opt out those parts of the application for which proper architectural patterns are already followed. As such, the ordered list can include first those patterns which map to unmatched application components with respect to fully matching patterns, and then those which increasingly include matched application components with respect again to fully matching patterns.

7.2.2 Deployment patterns

The profiling knowledge that is derived also includes facts about the actual configuration class of the crawled components, *e.g.*, GPU, FPGA, container, serverless. Such a knowledge can thus enable to propose configuration changes to the application. The most crucial issue here is how to identify what is a deployment pattern and whether this might relate also to the application requirements. Further, it can be argued that deployment patterns are related to architectural ones in the sense that we should not see what happens in the individual level of a single component, but we need to focus on the pattern level as it is more meaningful to check the right configuration for the combinations of components that belong to a certain pattern as this can be globally optimal, or at least optimal in that level.

Based on the above problematic, we can make the following assumptions:

a. we rely on matched architectural patterns;
b. we check the configuration of all components in those patterns;
c. we do not need to assess whether one deployment pattern is better than the other, but we need to discover all possible alternatives as the selection of those can depend on both the application requirements and the current Cloud offering landscape. Further, application requirements can change while the utility function is also planned to be modifiable during the application execution.

As such, we can deduce the following definition of what is a deployment pattern: *it is an architectural pattern where the configuration of all pattern components is fixed/stable*. This means that for one architectural pattern, we can have multiple deployment patterns related to it.

Based on this definition, we can actually now sketch the overall idea for deployment pattern matching/identification. For each architectural pattern, we seek the different combinations of configuration classes that have been found for the components of these patterns. Each from these combinations can then form a specific deployment pattern which is directly associated with the architectural pattern.

Once all deployment patterns have been identified, we immediately select those mapping to architectural patterns that fully match the application architecture. For the partially matching patterns that have been proposed to the user, we need to collect user input: *i.e.*, the user should select which patterns will be followed. Once this is done, then we can also include the deployment patterns for those partially matching architectural patterns that have been selected by the user.

The unification of all the identified deployment patterns is then relevant for the application architecture. For each application component included in a matching architectural pattern, we then check the current configuration classes available for it. In case that a configuration class is missing, it needs to be added.

In case that we have an application component not belonging to any of the architectural patterns that have been matched/selected, then we just go over all possible configuration classes that have been derived from the crawling and analysis phase, and we add those currently missing from the current set of configuration classes of that component.

In result, we have added all missing configuration classes for all application components in a systematic way such that there is no configuration class that does not make sense to be additionally incorporated. Further, through the association of deployment to architectural patterns, it is possible to derive some knowledge that can facilitate the selection of the right architecture variant during that first reasoning step in the application's deployment process. In particular, we can

formulate the respective CP-model such that it includes additional constraints that indicate which configuration class combinations are legitimate for particular parts of the application.

The whole idea has relied on the assumption that it is meaningful to consider only configuration class combinations that have been already found in the profiling knowledge and map to certain architectural patterns. However, it could be argued that the respective application space from which the profiling knowledge has been derived is not representative. In that case, a more simplified approach can be followed where we just seek to add missing configuration classes for all individual components of the application based on the profiling knowledge derived. This would then reduce the benefit of further restricting the solution space in terms of application deployment architecture selection through the respective constraints that could be instead deduced through the more sophisticated, aforementioned approach.

7.3 Pre-solving the model for the available resources

Once the requirements have been tuned and the architectural patterns have been identified, these can be used to initiate a search for available Cloud resources that can match the requirements and patters and store these in the node candidate database. Since the tuning and profiling is a continuous process, it is also necessary to prune the set of available node candidates and delete those node candidates that no longer is able to satisfy the requirements of the components.

This picture is complicated by the workflow grouping described in Section 6. The grouping process will create 'super-components' hosting sub-sequences of the workflow where the sub-sequence components are executed in a strict ordering. The requirements of the super components are the maximum of each requirement taken over all the sub-sequence components. The implication is that the workflow based grouping must be done prior to searching for available node candidates.

Finally, once all components and super-components have been defined, the new application model must be solved to ensure that there is at least one feasible application configuration for the application's current or predicted execution context. This implies employing the standard optimization process to find the optimal configuration that maximizes the utility, and provided that the utility of the best configuration under this revised model is better than the utility of the running configuration, the revised model is be released for deployment leading to an adaptation of the running application.

8 Performance modelling

Different Cloud solutions are being developed offering the capability of managing Cloud applications that can handle a dynamic workload while producing result complying with users' expectations, there is a need to determine the relation between the workload handled by the running application and its performance. The workload depends on users' activity therefore is subject to change over time.

8.1 The correlation problem

MORPHEMIC's proactive adaptation feature consists of providing the suitable resource configuration to the running application for improving the utility value before the service layer objective (SLO) defined by the application owner is violated. The accomplishment of this task requires MORPHEMIC to foresee or predict the application load and know the application performance model. The application performance model is a relationship between the application inputs, application configuration, and the application output. The application inputs can be seen as a load handled by the application, this could be the number of requests, the number of users, etc. The application configuration defines the resource under which the application is running, we can enumerate the memory, the number of CPU cores provided to the application, the number of instances of the given, the execution form like Virtual Machines, containers, big data task, etc. The third element, the application output are the Key Performance Indicators (KPI), for instance the response time, the latency, the throughput, the cost, etc.

As shown in the Figure 25, the performance model mimics the application behaviour. Therefore, model produces the expected application output given the input and the configuration under which the application is running. This capability allows MORPHEMIC to forecast chronically the application load, ingest the predicted load and the current application

configuration into the performance model to predict the application outcomes or output. These values will then be compared to the utility objective defined by the application owner. We can understand the performance model as a virtualized application representation that can be seen as a module where inputs are the application input and configuration, and the output are the application utility objective according to the following figure.

Figure 25 Overall performance model concept

8.2 Performance model general concept

As described above, the performance model represents the running application in terms of relationship between application's execution context, its configuration, and application performance indicators. Those three elements are the performance model parameters. The construction of this relationship lays on correlation that must be established between the performance model parameters. Mathematically, we can use the following expression.

*Output = **F**(input, configuration)*

The first task of the performance module will be the establishment of the function *F*. The performance module must also provide the capability of storing, evaluating its correctness, and updating the function *F*. Therefore, we can deduct three main tasks for the performance module described in the following subsections.

8.2.1 Model establishment

The creation of the model consists of applying suitable techniques for discovering the correlation between performance model parameters. The MORPHEMIC platform is built for running and orchestrating different types of applications with complex business logic. The relationship between performance model parameters could be linear, nonlinear, or a combination of both. This forces us to address the problem by applying an approach capable of creating very complex relationships.

Knowing that the performance model mimics the application behaviour, the model establishment is a process where the performance module will learn the relationship between performance model parameters. Since an application could have certain behaviour under some configuration, and load a totally different one under another different one, the correctness of the learner depends on the number of configurations available.

Machine learning provides different techniques where a model can be trained and adjusted according to data available in the dataset. The supervised machine learning can be used for addressing this problem where the input of the machine learning will be the application load and the application configuration, and the output of the machine learning model will be the performance indicators. Among many algorithms available under the supervised category, we have to face the algorithm selection problem. According to the diversity in terms of behaviour, constructing the performance module using one predetermined algorithm may not be the best since the KN Regression (K-neighbours) algorithm may give a very good performance for a certain type of applications, and not for other types of applications. The performance module will embed many algorithms, which will be trained on the same dataset, and the selection will be based on the

mean squared error. The Utility Generator, which selects the best configuration for a running application, requires different performance models. Otherwise, the performance module must create a model based on the application name and also take into account the target KPI. Therefore, for the application "A" exposing B, C and D as metrics and running under configuration X, Y and Z are required. The performance model must be able to create the models for the product set of the configurations and the measurements.

8.2.2 Model storage

The model must be stored for being used for prediction. There is a requirement to save the machine learning model as a file incorporating the application name and the targeted KPI. Example: A_B.sav, A_D.sav

8.2.3 Model evaluation and update

After having created the application performance model, there is a requirement to evaluate the correctness of the performance model. This latter consists of comparing the predicted value with the real measurements arriving from the MELODIC monitoring engine (EMS). The evaluation of the correctness of the application performance model is a periodical operation and requires a streaming connection to the monitoring exposed by the concerned application. The unit decides the necessity of retraining in charge of the evaluation and according to the error margin tolerated.

8.3 Architecture

The performance model module is connected to the Utility Generator from which the train process and prediction are initiated and to the persistent storage for comparing real value produced by the running application under a certain configuration with the prediction made for that configuration. Figure 26 shows the main sub-components constituting the performance model providing all functionalities required.

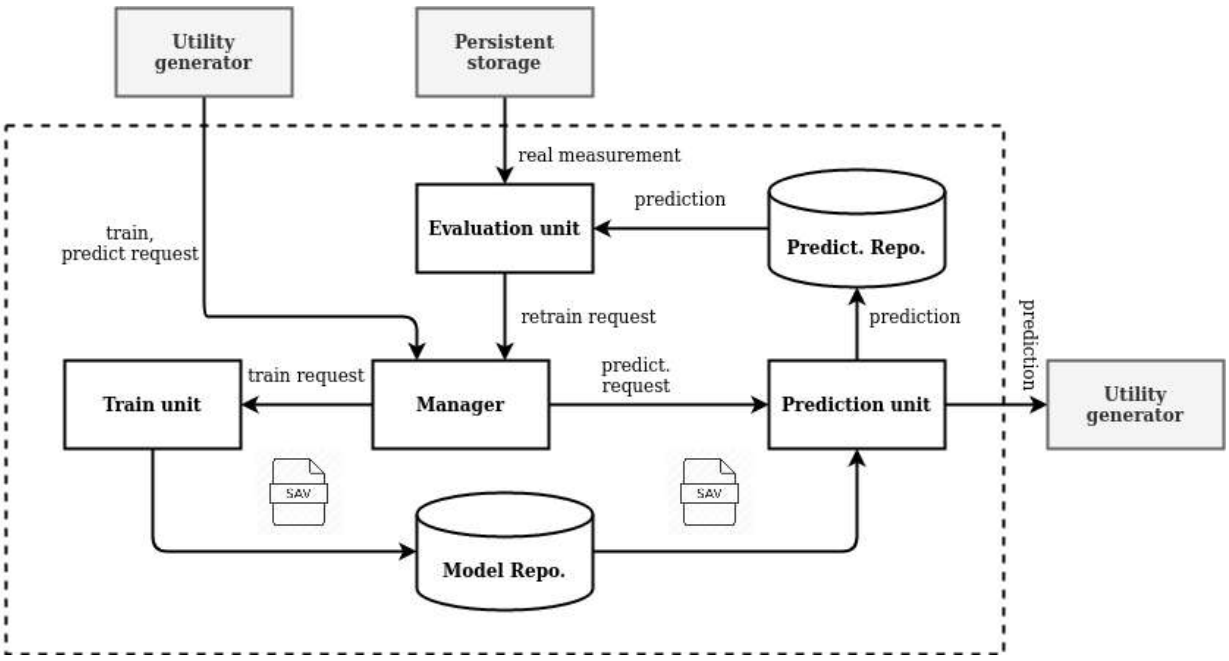

Figure 26 Performance model internal architecture

8.3.1 Manager

The manager has the component interfaces which allows the exchanges of information between the performance model and the Utility Generator. By information, we mean the training data and the configuration for predicting the performance. According to the requirement of the MORPHEMIC platform, two types of interfaces are exposed through

the manager. The REST-API is mostly for receiving training data and the Remote Procedure Call (RPC) for prediction functionality.

The REST-API has the following specifications with the same functionality also available as RPC calls:

/api/v1/train : for initiating a training process.

/api/v1/model: This endpoint returns the information related to the training process.

/api/v1/predict: Using this endpoint, the caller can predict a specific metric (target) given certain configuration.

The train method and model information method are not called frequently, so the REST-API is the most suitable to fulfil that purpose. The RPC interface is used for prediction since the Utility Generator needs several configurations tries for the selection of the configuration that improves the utility value.

Knowing the computing resource consumption of algorithms using machine learning, to avoid the congestion of the manager and the training process is performed in a detached process.

8.3.2 Train unit

The training functionalities are triggered by the manager. The application's name, the URL of the dataset, the features to used, the target field (performance) and the application's variant are passed as parameters. The application component variants accepted are: docker, serverless, GPU, FPGA, edge. The training process starts by calling several regression-oriented algorithms and then select the algorithm providing the best mean squared error. The training data and the machine model are then saved on files for later usage. The key information defining a model in the performance module are the application's name, the targeted metric, and the application's variant. Thus, for an application more than one trained model can be stored.

8.3.3 Prediction unit

The unit to predict the functionalities receives its parameters from the manager. These parameters are the application's name, the targeted metric, the application's variant, and a dictionary containing configurations' name and their respective values. The predict class first verify if the corresponding machine learning module exists. The machine learning is then loaded, and the configuration is passed for producing a prediction. The prediction and the training data are returned.

For evaluating the accuracy of the predictions made, the request received by the predict class combined with the prediction produced are stored on a local database for being compared to the real value corresponding to that configuration. A retrain process can be triggered if the accuracy falls below a certain threshold.

8.3.4 Evaluation unit

According to the dynamic behaviour of Cloud applications driven by the change of workload based on users' activity, the correlation between application's metrics and its configuration must be renewed and kept updated. When the correlation is not valid anymore, the performance model needs to be informed for triggering a retrain process where new observations are considered. The information is possible by evaluation otherwise by comparing the predicted value and the real value of a specific configuration. This requires a subscription to the EMS for receiving these data in a streaming manner.

Considering the predicted value P on the metric A for the configuration values $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ corresponding to the metric time series X, Y, Z. A subscription request must be sent for receiving the metrics X, Y, Z. When values $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ are deployed then the real value of the metric A will be compared to the value P. If the difference exceeded some threshold in percent, then a retrain request will be generated.

9 Application adaptation

There are few benefits of deploying a static application to a Cloud infrastructure. If it is known how much resources and how many servers an application needs to run, one will be better off providing and maintaining that infrastructure

privately rather than renting more expensive Cloud resources. However, an application may have intrinsic variability caused by the data it processes or by the application's users. Furthermore, this will be accentuated if the application is designed to be always available and thus stability, robustness, and resilience become important aspects of the deployment decisions. It will be costly and error prone to have a human DevOps team constantly monitoring the application and exploiting the elasticity of the Cloud computing paradigm to provide or to remove application resources as the demand fluctuates. Autonomic approaches based on the MAPE-K loop paradigm are needed, and this has been used to support optimized Cloud application deployment. The starting point is application-level sensors to detect changes in the execution context. The Cloud platform level sensors will normally only be able to monitor metric values of the deployed resources, like memory use or CPU utilization, and correlating such quantities to infer for instance the number of application users can be extremely difficult. However, the number of users can easily be monitored by the application itself. That is the reason that the monitoring module is still needed to detect changes in the application's execution context from the application level raw sensor measurements.

The best possible deployment configuration must be found once an execution context change has been detected. So the goals of the adaptation are threefold:

- The application resources must be assessed and optimized for the current situation; and then
- The deployment of the optimized application must be decided; and
- A plan must be made for changing the deployment from the application configuration currently deployed to the new configuration.

9.1 MELODIC application reconfiguration

Optimizing the application involves finding the right location and cardinality of the various application components. This typically involve solving a combinatorial optimization problem whose time complexity grows exponentially with the number of factors to consider for the next deployment. The reasoning process is the core part of the MELODIC platform. It is responsible for finding the best deployment solution for the given application utility within the given constraints and current application context. The key components involved in the reasoning part are the following:

1. The Metasolver monitors the problem constraints, the SLOs, and if a constraint is violated, it chooses the right solver for a given problem:
2. The CP Solver, which is a solver dedicated to solving nonlinear constraint optimization problems
3. The Genetic algorithm solver.
4. The Parallel Tempering (PT) based solver, which uses the Parallel Tempering method to solve optimization problems.
5. The Monte Carlo Tree Search (MCTS) based solver, which uses advanced implementation of the MCTS algorithm to solve optimization problems.
6. The Utility Generator which is the module responsible for the calculation of the utility value per feasible application configuration candidate proposed by a solver.

The adaptation part deals with the adaptation and orchestration of the deployment model of the application to be deployed across the chosen Cloud providers. It contains one key component, the Adapter, which translates the optimal CP model solution to a CAMEL deployment model, then prepares, and finally orchestrates the plan of the deployment.

The adaptation process takes the application topology graph resulting from the optimized configuration graph of the previous step, and maps this to the graphs of the deployment pattern. Then the adaptation process creates the execution topology graph. This graph is created as the difference between the currently deployed and the new solutions.

Each of the application topology graphs contains the following entities:

- Application Task - task grouping all actions needed to deploy application.
- Application Instance Task - task for deploying particular instance of the application.
- VM Type Task - task grouping all actions needed to provision given VM type.
- VM Instance Task - task to provision given VM instance.
- Component Type Task - task grouping all actions needed for the given component type.

- Component Instance Task - task to deploy given instance of component.
- Communication Type Task - task grouping all actions for setting given communication between components.
- Communication Instance Task - task to set given communication per one component.
- Hosting Type Task - all actions needed to map component to the VM.
- Hosting Instance Task - deploying given component on particular VM.

They are strictly dependent on each other and should be put in the graph in the right order per component. Also, the location of the tasks per each component depends on the connections between the components. It determines the order of deployment execution steps. The graph is created using the following algorithm:

1. The initial application topology graph is fetched.
2. The target application topology graph is created based on the current solution.
3. The target application topology graph is traversed recursively and differences between these two graphs are identified.
4. Each difference is translated into the relevant action to the execution application topology graph, based on the following rules of model comparison:
    - if a target object is not found in the current model - a CREATE task is generated;
    - if a current object is not found in the target model - a DELETE task is generated;
    - if an object is found in both the current and target models, but some of the attributes are not equal - an UPDATE task is generated;
    - if an object is found in both the current and target models and all attributes are equal - no action is generated;

Algorithm 2 Adapter graph creation and update algorithm

1	initialTopologyGraph = fetchInitialApplicationTopologyGraph()	*Initialize current topology graph.*
2	targetTopologyGraph = create(solution)	*Initialize target topology graph.*
3	**while** (targetTopologyGraph.nextNode() != null)	*Iterate target topology graph*
4	**if** (initialTopologyGraph.checkNode(targetTopologyGraph.getNode() == NON_Exists) createObjectInCloud()	*If the object doesn't exist in current topology but exist in target topology then create object.*
5	**if** (initialTopologyGraph.checkNode (targetTopologyGraph.getNode() == Exists) updateObjectInCloud()	*If the object exists in both topologies then update object.*
6	**while** (initialTopologyGraph.nextNode() != null)	*Iterate initial topology graph*
7	**if**(targetTopologyGraph.checkNode (initialTopologyGraph.getNode() == NON_Exists) deleteObjectInCloud()	*If the object not exists in target topology then delete object*

Based on the presented above algorithm the new execution graph is created. Then the execution graph is executed. The orchestration of execution is done by the Adapter, which traverses through the execution graph and executes each action using the Executionware. Executionware executes the relevant action using the API of selected cloud providers.

9.2 MORPHEMIC component variant activation

One of the key features of the MORPHEMIC is ability to perform architecture optimization. Key assumption for the architecture optimization is to select the most optimal variant of the component implementation, as addition to the resource optimization. The workflow for the architecture optimization based on component variant activation will be as follows:

1. The Architecture Optimizer based on the CAMEL model definition prepares a better CAMEL model with proper models of available architecture variants for each component.
2. The prepared CAMEL model is provided to the MELODIC Reasoning module and the best solution is determined. The best solution contains both configuration of the Cloud resources and types of the architecture per given component. Each component can be deployed in one or more architecture variants simultaneously, based on the utility function calculation. The number, resource configuration and architecture variants will be determined by cardinality of each component variants and selected node candidates to deploy given variant.
3. The deployment will be done through Exectuionware to selected resources and using selected architecture variants.
4. The running application will be monitored, and in case of SLO violation the actions presented in steps 2 to 4 will be executed again.

The process presented above adds additional, unique, and novel dimension to the optimization and adaptation process and will allow for dynamic and continues architecture optimization.

9.3 MORPHEMIC Lazy Adapter

The Lazy Adapter component will be part of the MORPEHMIC Pre-processor. This component will be used in the architecture optimization process. The goal of the Lazy Adapter component is to handle adaptation related actions during the architecture optimization process and to orchestrate deployment of the final configuration through the MELODIC platform. The key features of the Lazy Adapter are as follow:

- Calculating penalty for reconfiguration: in the architecture optimization process the Lazy Adapter will calculate the reconfiguration penalty for each configuration. The reconfiguration penalty will be used by the Utility Generator to calculate value of utility function for each configuration. The Lazy Adapter will be invoked by the Utility Generator.
- Evaluating adaptation constraints: in the architecture optimization process the Lazy Adapter will evaluate adaptation constraints for each configuration. Each solver, during the solution space searching process, will invoke the Lazy Adapter to evaluate adaptation constraints. The Lazy Adapter will be invoked by each solver.
- Preparing final configuration for deployment and request execution using the MELODIC platform: the Lazy Adapter will prepare final configuration based on the solution found by solver. The final configuration will be passed to MELODIC platform to be deployed.

MELODIC platform, based on the final configuration received from the Lazy Adapter will execute the following steps:

1. Replace the current CAMEL model, with all sub-models, in the MELODIC platform from the specific validity date.
2. Execute the reconfiguration of the application based on the received final configuration at the specified validity date.

The newly reconfigured application will be monitored and adapted at run-time, if needed, by the MELODIC Platform inner optimization loop until the outer optimization loop of MORPHEMIC proposes a better configuration.

10 Conclusions

This deliverable has analysed the different steps in the optimization process foreseen for MORPHEMIC and how it interacts with the run-time application management performed by MELODIC. An architecture for the polymorphic adaptation is proposed, and the research challenges for the various components are analysed. The deliverable thereby forms a research programme for the later releases of the MORPHEMIC project.

Abbreviations

API	Application Programming Interface
BPNN	Back Propagating Neural Network
CPU	Central Processing Unit
CAMEL	Cloud Application Modeling and Execution language
CNN	Convolutional Neural Network
CP	Constraint Programming
CSG	Coalition Structure Generation
DAG	Directly Acyclic Graph
EMS	Event Management System
FFN	Feed Forward Neural Network
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
IP	Integer-Partition
JAR	Java ARchive
KNN	K-Nearest Neighbour
KPI	Key Performance Indicators
LSTM	Long short-term memory RNN
MAPE-K	Monitor, Analyse, Plan, and Execute using the obtained Knowledge
MCDM	Multi-Criteria Decision Making
ML	Machine Learning
MRI	Magnetic Resonance Imaging
MOMKP	Multi-Objective Multidimensional Knapsack Problem
MCKP	Multiple-Choice Knapsack Problem
MMKP	Multiple Multidimensional Knapsack Problem
ODP-IP	Optimal Dynamic Programming – Integer-Partition

PT	Parallel Tempering
RAM	Random Access Memory
RNN	Recurrent Neural Network
RPC	Remote Procedure Call
SLO	Service Level Objective
SNN	Self-Normalizing Neural Network
SVR	Support Vector Regressor
TFT	Temporal Fusion Transformer
TPU	Tensor Processing Units
URL	Universal Resource Locator
VM	Virtual Machine

References

[1] 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, May 2018, pp. 360–367. doi: 10.1109/WAINA.2018.00112.

[2] Jeffrey O. Kephart and David M. Chess, 'The vision of autonomic computing', *Computer*, vol. 36, no. 1, pp. 41–50, 2003, doi: 10.1109/MC.2003.1160055.

[3] IBM, 'An architectural blueprint for autonomic computing', IBM, 17 Skyline Drive, Hawthorne, NY 10532, U.S.A., White Paper Third Edition, Jun. 2005. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[4] Peter C. Fishburn, *Utility theory for decision making*, vol. 18. New York: Wiley, 1970. Accessed: Mar. 30, 2014. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0708563

[5] Jeffrey O. Kephart and Rajarshi Das, 'Achieving Self-Management via Utility Functions', *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, Jan. 2007, doi: 10.1109/MIC.2007.2.

[6] Jacqueline Floch *et al.*, 'Playing MUSIC --- building context-aware and self-adaptive mobile applications', *Softw. Pract. Exper.*, vol. 43, no. 3, pp. 359–388, Mar. 2013, doi: 10.1002/spe.2116.

[7] Salvatore Greco, Matthias Ehrgott, and José Rui Figueira, Eds., *Multiple Criteria Decision Analysis: State of the Art Surveys*, 2nd ed. New York: Springer-Verlag, 2016. doi: 10.1007/978-1-4939-3094-4.

[8] L. Zadeh, 'Optimality and non-scalar-valued performance criteria', *IEEE Transactions on Automatic Control*, vol. 8, no. 1, pp. 59–60, Jan. 1963, doi: 10.1109/TAC.1963.1105511.

[9] Geir Horn and Marta Rózańska, '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, Jun. 2019, pp. 147–156. doi: 10.1109/ICAC.2019.00026.

[10] David G. Luenberger and Yinyu Ye, *Linear and Nonlinear Programming*, 3rd ed. Springer, 2008.

[11] Geir Horn, 'A vision for a stochastic reasoner for autonomic cloud deployment', in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud 2013)*, Conference Location: Oslo, Norway, Sep. 2013, pp. 46–53. doi: 10.1145/2513534.2513543.

[12] Gene M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities', in *Proceedings of the AFIPS spring joint computer conference*, Conference location: Atlantic City, New Jersey, USA, Apr. 1967, pp. 483–485. doi: 10.1145/1465482.1465560.

[13] Jon Wakefield, *Bayesian and Frequentist Regression Methods*. New York: Springer-Verlag, 2013. doi: 10.1007/978-1-4419-0925-1.

[14] René Vidal, Yi Ma, and S. S. Sastry, *Generalized Principal Component Analysis*. New York: Springer-Verlag, 2016. doi: 10.1007/978-0-387-87811-9.

[15] Patrick Royston and Willi Sauerbrei, *Multivariable Model-Building: A pragmatic approach to regression analysis based on fractional polynomials for modelling continuous variables*. John Wiley & Sons, Ltd, 2008. doi: 10.1002/9780470770771.

[16] Bradley P. Carlin and Thomas A. Louis, *Bayesian Methods for Data Analysis*, 3rd ed. Chapman and Hall/CRC, 2008.

[17] Alex J. Smola and Bernhard Schölkopf, 'A tutorial on support vector regression', *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, Aug. 2004, doi: 10.1023/B:STCO.0000035301.49549.88.

[18] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter, 'Self-Normalizing Neural Networks', in *Advances in Neural Information Processing Systems 30. Proceedings from the conference on Neural Information Processing Systems 2017 (NIPS 2017)*, Conference location: Long Beach, CA, USA, Dec. 2017, pp. 971–980. Accessed: Aug. 07, 2020. [Online]. Available: http://papers.nips.cc/paper/6698-self-normalizing-neural-networks

[19] Ali E. Abbas, *Foundations of Multiattribute Utility*. Cambridge, United Kingdom: Cambridge University Press, 2018. Accessed: Apr. 20, 2021. [Online]. Available: https://www.cambridge.org/no/academic/subjects/statistics-probability/optimization-or-and-risk/foundations-multiattribute-utility, https://www.cambridge.org/no/academic/subjects/statistics-probability/optimization-or-and-risk

[20] Roger B. Nelsen, *An Introduction to Copulas*, 2nd ed. New York: Springer-Verlag, 2006. doi: 10.1007/0-387-28678-0.

[21] Ali E. Abbas and Zhengwei Sun, 'Archimedean Utility Copulas with Polynomial Generating Functions', *Decision Analysis*, vol. 16, no. 3, pp. 218–237, Aug. 2019, doi: 10.1287/deca.2018.0386.

[22] Mohammad Tabatabaei, Jussi Hakanen, Markus Hartikainen, Kaisa Miettinen, and Karthik Sindhya, 'A survey on handling computationally expensive multiobjective optimization problems using surrogates: non-nature inspired methods', *Struct Multidisc Optim*, vol. 52, no. 1, pp. 1–25, Jul. 2015, doi: 10.1007/s00158-015-1226-z.

[23] Barry O'Sullivan, 'Automated Modelling and Solving in Constraint Programming', presented at the Twenty-Fourth AAAI Conference on Artificial Intelligence, Conference Location: Atlanta, Georgia, USA, Jul. 2010. Accessed: Dec. 05, 2021. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1899

[24] N. Jussien, G. Rochart, and X. Lorca, 'Choco: an Open Source Java Constraint Programming Library', in *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, France, 2008, pp. 1–10. Accessed: Dec. 06, 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00483090

[25] S. Chib and E. Greenberg, 'Understanding the Metropolis-Hastings Algorithm', *The American Statistician*, vol. 49, no. 4, pp. 327–335, Nov. 1995, doi: 10.1080/00031305.1995.10476177.

[26] R. H. Swendsen and J.-S. Wang, 'Replica Monte Carlo Simulation of Spin-Glasses', *Phys. Rev. Lett.*, vol. 57, no. 21, pp. 2607–2609, Nov. 1986, doi: 10.1103/PhysRevLett.57.2607.

[27] D. J. Earl and M. W. Deem, 'Parallel tempering: Theory, applications, and new perspectives', *Phys. Chem. Chem. Phys.*, vol. 7, no. 23, pp. 3910–3916, Nov. 2005, doi: 10.1039/B509983H.

[28] T. Bäck and H.-P. Schwefel, 'An Overview of Evolutionary Algorithms for Parameter Optimization', *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, Mar. 1993, doi: 10.1162/evco.1993.1.1.1.

[29] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan, 'A fast and elitist multiobjective genetic algorithm: NSGA-II', *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.

[30] Richard S. Sutton and Andrew G. Barto, *Reinforcement learning*, vol. 9. Boston, MA, USA: MIT Press, 1998.

[31] Mandayam A. L. Thathachar and P. S. Sastry, *Networks of Learning Automata: Techniques for Online Stochastic Optimization*, 1st ed. Boston, MA, USA: Kluwer Academic, 2004.

[32] Thomas Dean and Mark Boddy, 'An analysis of time-dependent planning', in *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence (AAAI'88)*, Conference Location: Saint Paul, Minnesota, USA, Aug. 1988, pp. 49–54.

[33] Benjamin W. Wah and Yi Xin Chen, 'Optimal Anytime Constrained Simulated Annealing for Constrained Global Optimization', in *Proceedings of the 6th International Conference on Principles and Practice of Constraint*

Programming (CP 2000), Conference Location: Singapore, Sep. 2000, vol. 1894, pp. 425–440. doi: 10.1007/3-540-45349-0_31.

[34] Miguel Ángel Domínguez-Ríos, Francisco Chicano, and Enrique Alba, 'Effective anytime algorithm for multiobjective combinatorial optimization problems', *Information Sciences*, vol. 565, pp. 210–228, Jul. 2021, doi: 10.1016/j.ins.2021.02.074.

[35] William S. Cleveland, Susan J. Devlin, and Eric Grosse, 'Regression by local fitting: Methods, properties, and computational algorithms', *Journal of Econometrics*, vol. 37, no. 1, pp. 87–114, 1988, doi: 10.1016/0304-4076(88)90077-2.

[36] Jerome H. Friedman, 'Multivariate Adaptive Regression Splines', *Ann. Statist.*, vol. 19, no. 1, pp. 1–67, Mar. 1991, doi: 10.1214/aos/1176347963.

[37] James Ramsay and Giles Hooker, *Dynamic Data Analysis: Modeling Data with Differential Equations*. New York, NY: Springer New York, 2017. [Online]. Available: RamsayHooker17

[38] Lars Grüne and Jürgen Pannek, *Nonlinear Model Predictive Control: Theory and Algorithms*, 2nd ed. Springer International Publishing, 2017. Accessed: Oct. 25, 2019. [Online]. Available: http://www.springer.com/gp/book/9783319460239

[39] Bernard Widrow, Narendra K. Gupta, and Sidhartha Maitra, 'Punish/Reward: Learning with a Critic in Adaptive Threshold Systems', *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 5, pp. 455–465, Sep. 1973, doi: 10.1109/TSMC.1973.4309272.

[40] Ding Wang and Chaoxu Mu, *Adaptive Critic Control with Robust Stabilization for Uncertain Nonlinear Systems*. Springer Singapore, 2019. doi: 10.1007/978-981-13-1253-3.

[41] F. J. Massey, 'The Kolmogorov-Smirnov Test for Goodness of Fit', *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, Mar. 1951, doi: 10.1080/01621459.1951.10500769.

[42] M. Toussaint, 'The Bayesian Search Game', in *Theory and Principled Methods for the Design of Metaheuristics*, Y. Borenstein and A. Moraglio, Eds. Berlin, Heidelberg: Springer, 2014, pp. 129–144. doi: 10.1007/978-3-642-33206-7_7.

[43] P. Liashchynskyi and P. Liashchynskyi, 'Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS', *arXiv:1912.06059 [cs, stat]*, Dec. 2019, Accessed: Nov. 29, 2021. [Online]. Available: http://arxiv.org/abs/1912.06059

[44] G. Bebis and M. Georgiopoulos, 'Feed-forward neural networks', *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, Oct. 1994, doi: 10.1109/45.329294.

[45] W. Luo, W. Liu, and S. Gao, 'A Revisit of Sparse Coding Based Anomaly Detection in Stacked RNN Framework', 2017, pp. 341–349. Accessed: Nov. 29, 2021. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2017/html/Luo_A_Revisit_of_ICCV_2017_paper.html

[46] S. Hochreiter and J. Schmidhuber, 'Long Short-term Memory', *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997, doi: 10.1162/neco.1997.9.8.1735.

[47] R. Dey and F. M. Salem, 'Gate-variants of Gated Recurrent Unit (GRU) neural networks', in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2017, pp. 1597–1600. doi: 10.1109/MWSCAS.2017.8053243.

[48] A. Vaswani *et al.*, 'Attention is All you Need', in *Advances in Neural Information Processing Systems*, 2017, vol. 30. Accessed: Nov. 29, 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[49] S. Albawi, T. A. Mohammed, and S. Al-Zawi, 'Understanding of a convolutional neural network', in *2017 International Conference on Engineering and Technology (ICET)*, Aug. 2017, pp. 1–6. doi: 10.1109/ICEngTechnol.2017.8308186.

[50] L. Breiman, 'Bagging predictors', *Mach Learn*, vol. 24, no. 2, pp. 123–140, Aug. 1996, doi: 10.1007/BF00058655.

[51] T. Garipov, P. Izmailov, D. Podoprikhin, D. P. Vetrov, and A. G. Wilson, 'Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs', in *Advances in Neural Information Processing Systems*, 2018, vol. 31. Accessed: Nov. 29, 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/be3087e74e9100d4bc4c6268cdbe8456-Abstract.html

[52] I. Maqsood, M. Khan, and A. Abraham, 'An ensemble of neural networks for weather forecasting', *Neural Computing and Applications*, vol. 13, pp. 112–122, Jun. 2004, doi: 10.1007/s00521-004-0413-4.

[53] J. Friedman, 'Greedy Function Approximation: A Gradient Boosting Machine', *The Annals of Statistics*, vol. 29, Nov. 2000, doi: 10.1214/aos/1013203451.

[54] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. Hopcroft, and K. Weinberger, 'Snapshot Ensembles: Train 1, get M for free', Mar. 2017.

[55] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, 'Temporal Fusion Transformers for interpretable multi-horizon time series forecasting', *International Journal of Forecasting*, Jun. 2021, doi: 10.1016/j.ijforecast.2021.03.012.

[56] B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio, 'N-BEATS: Neural basis expansion analysis for interpretable time series forecasting', *arXiv:1905.10437 [cs, stat]*, Feb. 2020, Accessed: Sep. 14, 2021. [Online]. Available: http://arxiv.org/abs/1905.10437

[57] J. Benesty, J. Chen, Y. Huang, and I. Cohen, 'Pearson Correlation Coefficient', in *Noise Reduction in Speech Processing*, I. Cohen, Y. Huang, J. Chen, and J. Benesty, Eds. Berlin, Heidelberg: Springer, 2009, pp. 1–4. doi: 10.1007/978-3-642-00296-0_5.

[58] R. Pascanu, T. Mikolov, and Y. Bengio, 'On the difficulty of training recurrent neural networks', in *Proceedings of the 30th International Conference on Machine Learning*, May 2013, pp. 1310–1318. Accessed: Nov. 29, 2021. [Online]. Available: https://proceedings.mlr.press/v28/pascanu13.html

[59] Karl J. Åström and Björn Wittenmark, *Adaptive Control: Second Edition*, Second. Dover Publications, 2013.

[60] 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.

[61] Weifeng Liu, José C. Príncipe, and Simon Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*. John Wiley & Sons, 2010. doi: 10.1002/9780470608593.

[62] Derong Liu, Qinglai Wei, Ding Wang, Xiong Yang, and Hongliang Li, *Adaptive Dynamic Programming with Applications in Optimal Control*. Springer International Publishing, 2017. doi: 10.1007/978-3-319-50815-3.

[63] Qinglai Wei, Ruizhuo Song, Benkai Li, and Xiaofeng Lin, *Self-Learning Optimal Control of Nonlinear Systems: Adaptive Dynamic Programming Approach*, vol. 103. Springer Singapore, 2018. doi: 10.1007/978-981-10-4080-1.

[64] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali, 'Approximation and online algorithms for multidimensional bin packing: A survey', *Computer Science Review*, vol. 24, pp. 63–79, May 2017, doi: 10.1016/j.cosrev.2016.12.001.

[65] Nikhil Bansal, Marek Eliás, and Arindam Khan, 'Improved Approximation for Vector Bin Packing', in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'15)*, Conference Location: Arlington, Virginia, USA, Jan. 2016, pp. 1561–1579. doi: 10.1137/1.9781611974331.ch106.

[66] Boaz Patt-Shamir and Dror Rawitz, 'Vector bin packing with multiple-choice', *Discrete Applied Mathematics*, vol. 160, no. 10, pp. 1591–1600, Jul. 2012, doi: 10.1016/j.dam.2012.02.020.

[67] Hans Kellerer, Ulrich Pferschy, and David Pisinger, *Knapsack Problems*. Springer, 2004.

[68] Soukaina Laabadi, Mohamed Naimi, Hassan El Amri, and Boujemâa Achchab, 'The 0/1 Multidimensional Knapsack Problem and Its Variants: A Survey of Practical Models and Heuristic Approaches', *American Journal of Operations Research*, vol. 8, no. 5, pp. 365–439, Sep. 2018, doi: 10.4236/ajor.2018.85023.

[69] Changyan Yi and Jun Cai, 'Combinatorial spectrum auction with multiple heterogeneous sellers in cognitive radio networks', in *Proceedings of the IEEE International Conference on Communications (ICC)*, Conference Location: Sydney, NSW, Australia, Jun. 2014, pp. 1626–1631. doi: 10.1109/ICC.2014.6883555.

[70] Yang Song, Chi Zhang, and Yuguang Fang, 'Multiple multidimensional knapsack problem and its applications in cognitive radio networks', in *Proceedings of the 2008 IEEE Military Communications Conference (MILCOM)*, Conference Location: San Diego, CA, USA, Nov. 2008, pp. 1–7. doi: 10.1109/MILCOM.2008.4753629.

[71] Ricardo Stegh Camati, Alcides Calsavara, and Luiz Lima Jr., 'Solving the Virtual Machine Placement Problem as a Multiple Multidimensional Knapsack Problem', in *Proceedings of the The Thirteenth International Conference on Networks (ICN'14)*, Conference Location: Nice, France, Feb. 2014, pp. 253–260. [Online]. Available: https://www.iaria.org/conferences2014/ICN14.html

[72] Cheng He, Joseph Y.-T. Leung, Kangbok Lee, and Michael L. Pinedo, 'An improved binary search algorithm for the Multiple-Choice Knapsack Problem', *RAIRO-Oper. Res.*, vol. 50, no. 4–5, pp. 995–1001, Oct. 2016, doi: 10.1051/ro/2015061.

[73] Thibaut Lust and Jacques Teghem, 'The multiobjective multidimensional knapsack problem: a survey and a new approach', *International Transactions in Operational Research*, vol. 19, no. 4, pp. 495–520, 2012, doi: 10.1111/j.1475-3995.2011.00840.x.

[74] Stefan Voß and Eduardo Lalla-Ruiz, 'A set partitioning reformulation for the multiple-choice multidimensional knapsack problem', *Engineering Optimization*, vol. 48, no. 5, pp. 831–850, May 2016, doi: 10.1080/0305215X.2015.1062094.

[75] Talal Rahwan, Tomasz P. Michalak, Michael Wooldridge, and Nicholas R. Jennings, 'Coalition structure generation: A survey', *Artificial Intelligence*, vol. 229, pp. 139–174, Dec. 2015, doi: 10.1016/j.artint.2015.08.004.

[76] Tomasz Michalak, Talal Rahwan, Edith Elkind, Michael Wooldridge, and Nicholas R. Jennings, 'A hybrid exact algorithm for complete set partitioning', *Artificial Intelligence*, vol. 230, pp. 14–50, Jan. 2016, doi: 10.1016/j.artint.2015.09.006.

[77] Talal Rahwan, Sarvapali D. Ramchurn, Nicholas R. Jennings, and Andrea Giovannucci, 'An Anytime Algorithm for Optimal Coalition Structure Generation', *Journal of Artificial Intelligence Research*, vol. 34, pp. 521–567, Apr. 2009, doi: 10.1613/jair.2695.

[78] Tomasz Michalak, Jacek Sroka, Talal Rahwan, Michael Wooldridge, Peter McBurney, and Nicholas R. Jennings, 'A Distributed Algorithm for Anytime Coalition Structure Generation', in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, Conference Location: Toronto, ON, Canada, May 2010, vol. 1, pp. 1007–1014. Accessed: Nov. 08, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=1838206.1838342