Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

QoS-Aware Matchmaking for Adaptative Microservice-based Applications

Polona Štefanič*[1,2] | Petar Kochovski[2] | Omer F. Rana[1] | Vlado Stankovski[2]

[1]Cardiff University, School of Computer Science and Informatics, Queens Building, 5 The Parade, Cardiff CF24 3AA, UK
[2]University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, 1000 Ljubljana, Slovenia

Correspondence
*Vlado Stankovski, Email: vlado.stankovski@fri.uni-lj.si

Present Address
Večna pot 113, 1000 Ljubljana, Slovenia

Abstract

Applications that make use of Internet of Things (IoT) can capture an enormous amount of raw data from sensors and actuators, which is frequently transmitted to cloud data centres for processing and analysis. However, due to varying and unpredictable data generation rates and network latency, this can lead to a performance bottleneck for data processing. With the emergence of Fog and Edge computing hosted microservices, data processing could be moved towards the network edge. We propose a new method for continuous deployment and adaptation of multi-tier applications along edge, fog and cloud tiers by considering resource properties and non-functional requirements (e.g. operational cost, response time and latency etc.). The proposed approach supports matchmaking of application and Cloud-To-Things infrastructure based on a subgraph pattern matching (*P-Match*) technique. Results show that the proposed approach improves resource utilisation and overall application Quality of Service. The approach can also be integrated into software engineering workbenches for the creation and deployment of cloud-native applications, enabling partitioning of an application across the multiple infrastructure tiers outlined above.

KEYWORDS:
Fog Computing, Cloud Computing, Matchmaking, Microservices, Adaptation, Provisioning, Deployment

1 | INTRODUCTION

With a rapid growth in Internet of Things (IoT) and devices which continuously change their physical location, there is a need for data processing infrastructure to exist in-proximity to these devices. Current approaches rely on transmitting large amounts of generated data from such devices to cloud data centres for processing. However, this can lead to increased latency and computational overhead, as data needs to be transferred across multiple network hops from the source to the analysis platform (and back for actuation/results).

Fog computing provides hierarchically organised computational and network resources, that are closer to the user/data source. Conversely, cloud computing resources can be heterogeneous, virtually pooled, and geographically located away from IoT devices. Fog computing nodes can be more diverse than cloud nodes, less powerful and limited in capacity and capability.

It is therefore necessary to utilise various decision making mechanisms for deploying application components across computational resources located from Cloud-To-Things continuum. The adaptation of an application across these resources is the key challenge considered in this work. Such Quality of Service (QoS)-aware adaptation determines on which available resource

[0]**Abbreviations:** QoS, Quality of Service; IoT, Internet of Things; AI, Artificial Intelligence

instances an application components should be placed, and represents a major problem in many cloud applications. The method presented in this paper supports an automated QoS-aware continuous adaptation of application-components connected in a linear pipeline.

The abstract perspective above is realised by allocation of applications with microservice architecture across Cloud-To-Things continuum, whilst respecting specific QoS constraints. Applications utilising a microservice architecture can be represented as an acyclic graph, whereas the (interconnected) network nodes (i.e potential deployment options) represent a (complete) complex graph.

The research hypothesis of this work states that based on pattern matching, it is possible to map application components across a Cloud-To-Things continuum by utilising a subgraph isomorphism approach. For this purpose, we propose an orchestration algorithm based on Ullmann's algorithm [1] for subgraph isomorphism pattern matching. Since Ullmann's algorithm is only suitable for unlabelled graphs, we have designed a new algorithm for subgraph isomorphism matching that considers multi-labelled graphs. We consider QoS requirements as multi-labels on nodes and edges of a containerised microservice-based application. These requirements are matched with multi-labels, such as compute resources and network-level metrics of deployment options across a Cloud-To-Things continuum graph. As the present algorithm provides resource pattern matchmaking, we refer to it as *P-Match*. Our results show that *P-Match* outperforms other similar approaches, such as Markov Decision Processes (MDP) of Kochovski et al. [2] by: (i) enabling resource matchmaking of multiple application components on fog and cloud infrastructure; (ii) retrieving the results in less execution time; (iii) requiring fewer iterations to converge, and (iv) choice of optimal deployment options based on QoS constraints.

The rest of the paper is structured as follows: Section 2 introduces current approaches and mechanisms for efficient and optimised continuous adaptation of microservices on Cloud-To-Things continuum based on QoS parameters. Section 3 proposes our decision making approach for application component continuous adaptation based on a graph pattern matching algorithm, undertaken as part of the provisioning stage. The system architecture and interaction between modules in the architecture is described in Section 4. Section 5 provides an IoT-based motivating application scenario to demonstrate the use of the proposed approach. Section 6 contains a description of the experimental setup, our preliminary results and evaluation. In section 7 we describe how the proposed approach can be used to enhance the capabilities of Science Gateways and finally, section 8 concludes the paper with a discussion of our future work in this area.

2 | RELATED WORK

Big data [3] and video analytics workflow pipelines [4] have become popular and widely used in business and scientific applications [5]. A workflow pipeline in this context presents a paradigm for managing computational steps or stages in the application, where each stage represents a computational function/service. These components are usually executed in a successive manner, where the output of one function provides an input to the next in the pipeline. Hence, to generalise, an application is split into smaller number of stages or sub-components that can be considered as microservices. For returning more accurate, fast and real-time results such applications need high performance computing resources with low latency, and should be therefore submitted and executed on multi-cloud infrastructures and make effective use of geo-distributed data processing along multiple cloud data centres and fog nodes. Therefore understanding how such microservices should be distributed across fog and cloud environments remains a challenge.

Various research efforts have recently focused on deployment of microservices on fog infrastructure. The Osmotic Computing paradigm [6,7] proposes an abstraction for the execution of lightweight microservices at the edge of the network coupled with more complex microservices running within cloud data centres. Osmotic computing proposes mechanisms for migrating services between the edge, fog and cloud systems based on performance and security triggers, enabling an application to adapt its behaviour over time. The benefits of this approach lie in distributing load among edge and Cloud resources and dynamic microservices management in order to satisfy QoS requirements, however, the authors do not provide any optimisation or decision making method that would support their approach.

Moreover, processing of real-time video streams on clouds systems and at the network edge has also gained importance and popularity recently. Wu et al. [8] propose a dynamic cloud resource provisioning algorithm which uses network usage statistics to support video on-demand streaming. Despite the fact that the algorithm performs with low utilisation costs at clouds, the authors only provide evaluation on a central cloud and do not consider multi-cloud systems or edge resources at different geographical locations in order to assure lower latency. On the other hand, Zamani et al. [9] succeeded to overcome the limitations of a traditional

cloud-based approach by proposing a model for leveraging the use of computational resources across the edge, fog and cloud data centres for analysing real-time data streams from video cameras, to support object detection and recognition. However, the authors show that using edge resources results in low latency in comparison to clouds, yet they do not provide any optimisation or decision making methods for service placement or video processing based on the trade-off among more (conflicting) QoS parameters.

Particular research interest is focused towards optimal placement of microservices on fog and cloud infrastructure for enhanced QoS. For this purpose, several approaches have proposed the management and optimisation of non-functional requirements and QoS attributes. Garg et al.[10] propose deterministic approaches based on an Analytic Hierarchy Process (AHP), which considers QoS requirements in order to rank cloud-based infrastructure deployment options. Although the approach returns promising results, AHP is a computationally intensive process, particularly if considering a large amount of parameters, as it will produce a computational overhead due to an extensive amount of decision variables.

Focusing on Virtual Machine (VM) placement and service allocation, Wang et al.[11] use a stochastic bin packing algorithm, which addresses CPU and bandwidth requirements. The authors manage to reduce the server usage by 30%, however the approach is focused on CPU resources and Virtual Machines (VMs), and does not consider other QoS constraints (such as memory utilisation or storage) and other virtualisation options (e.g. containers). Similarly, Srikantaiah et al.[12] consider the VM consolidation problem as a multi-objective bin packing problem and evaluate trade-offs for minimising energy consumption and resource utilisation and maximising workload performance. According to their evaluation, the approach is able to allocate multiple application tiers across physical servers in such a manner that each tier operates with optimal performance and with minimum energy usage. Nevertheless, the approach does not consider nodes at the network edge and other QoS parameters except CPU and disk resources. Zhang et al.[13] propose the Virtual Cloud Resources Allocation model based on a Constraint Programming (VCRA-CP) approach which recommends the reduction of resource usage (and therefore the associated cost) during provisioning in order to achieve the desired QoS requirements. Given this model, it reduces QoS violations and proposes lower resource usage costs. Those approaches consider QoS requirements in the optimisation process equally, which is a time-consuming and computationally intensive process.

Mi et al.[14] propose a Genetic Algorithm Based Approach (GABA) for VM reallocation in cloud data centres. GABA is based on multi-objective optimisation, and returns a set of reduced options for potential placement locations. GABA is suitable for deployment according to time-varying requirements and network conditions. The benefit of such approach is self reconfiguration of VMs, however it only works with VMs and they have focused on cloud data centres and not on geographically distributed edge nodes as well. Another recent work[15], based on an integer linear program formulation, proposes an algorithm for initial application deployment and aims to find a near-optimal and time-efficient solution. This work achieves a near optimal solution and scales well also with increased number of services. However, the algorithm proposes application deployment across a single administrative domain and lacks deployment to multi-domains. Another similar work is focused on application task redeployment, and is based on Markov Decision Process theory[2]. Upon initial evaluation of the deployment options against hard constraints, the method builds a stochastic automaton in which each state represents one possible deployment option. The reward function takes into account QoS constraints. The pros and cons of MDP method are described in the evaluation section, in addition to a comparison of our approach with MDP.

Aral et al.[16] introduce a Distributed Replica Placement (D-ReP) algorithm for distributed, dynamic creation and replacement of data replicas across edge nodes. Additionally, they have implemented a messaging methodology for notification of edge nodes based on nearby replicas. Given this information, data requests can be forwarded to these replicas instead of the cloud data centre. Although an interesting approach, the approach leads to trade-offs between network latency and cost. In comparison to the present *P-Match* method, authors generate the undirected topology graph with Barabasi-Albert model. On the contrary, the present method was evaluated by utilising VM instances on the Amazon Elastic Compute Cloud whereas the infrastructure topology forms a complete graph. However, the drawbacks of D-ReP represent notification messaging for replica discovery which can cause network overhead. On the other hand, P-Match is focused on placing services which are based on QoS constraints that are predefined and cause minimal network overhead.

Although some of the described algorithms offer QoS-aware deployment of microservices[2,14,15], they only consider the deployment of a single microservice and do not consider multiple microservices, or the dependencies between them. For this purpose, we have designed an algorithm for microservice redeployment that is based on a matchmaking of application constraints and compute resources and QoS attributes for fog and cloud infrastructure. We define a Cloud-To-Fog infrastructure as

a labelled graph and a pipeline-based application as a directed acyclic graph. Mapping an Application graph to the Infrastructure graph (see Figure 3) can be defined as a graph embedding. Therefore, our approach utilises subgraph isomorphism pattern matching and is described in Section 3.

3 | MULTI-DOMAIN DEPLOYMENT OF MICROSERVICES

3.1 | High-level description of a problem

We consider a workflow pipeline as a connected set of microservices that can be deployed, one after another, in a pipeline manner. These microservices collectively form the application to be executed. We consider both coupled and interdependent (micro)services that exchange data. The deployment plan for the entire application is generated for all microservices once, at the beginning of the deployment phase. Given a heterogeneous Cloud-To-Fog infrastructure, the aim of the proposed method is a subsequent continuous adaptation of such applications to meet the end-to-end QoS requirements.

After an initial deployment of microservices on the infrastructure, the application QoS is monitored. In case performance assurance cannot be guaranteed (due to observed threshold violations of QoS metrics) our approach triggers recalculation of possible deployment locations for microservices. However, continuous adaptation must take into consideration dependencies and data flow requirements between microservices. Our approach *P-Match* proposes a QoS-aware adaptation approach for each microservice/component, identifying a feasible set of solutions that all meet QoS requirements. As a result, *P-Match* returns a ranked list of deployment option sets: each set represents a new continuous deployment plan for the available application components.

The proposed *P-Match* method can be used to support decision-making for pipeline-based application components and can be integrated into software engineering workbenches, such as Software Workbench for Interactive, Time Critical and Highly self-adaptive cloud applications (SWITCH)[17], particularly in the provisioning phase to support service provisioning and placement. The steps involved in the software engineering life-cycle of cloud-native applications are depicted in Figure 1 .

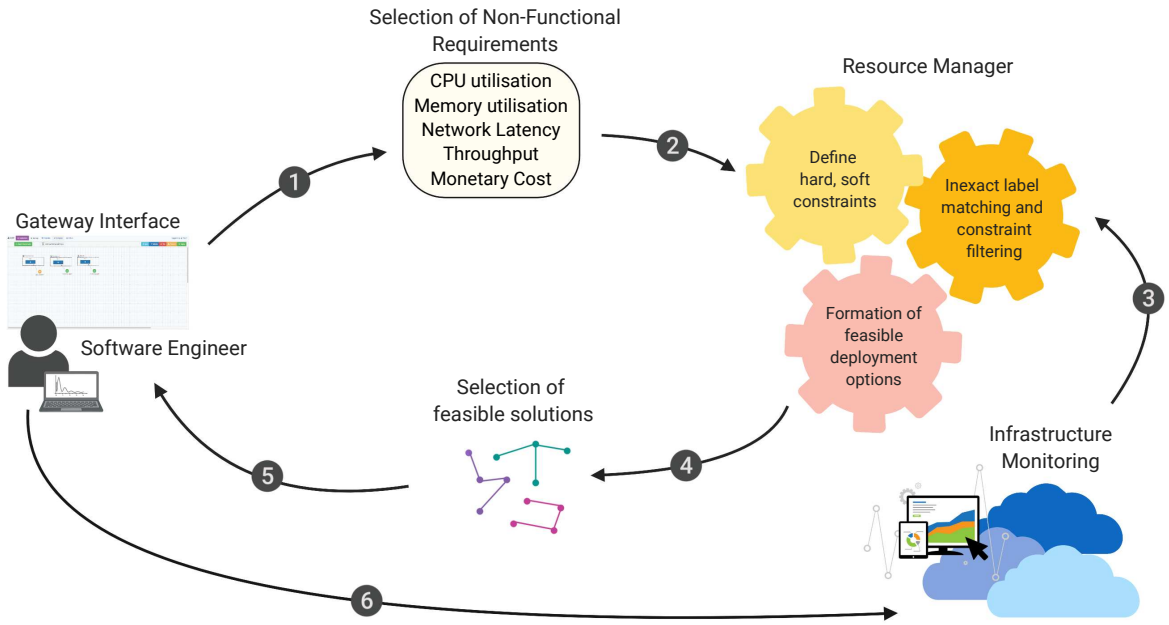


FIGURE 1 Decision-making process for continuous adaptation of AI-based microservices as part of a software engineering life-cycle, for cloud-native applications.

3.1.1 | Multi-Tier AI Application Model Design

The Artificial Intelligence (AI) pipeline model consists of a group of microservices that collectively represent the overall application. However, the functionalities in the pipeline can be distributed along multiple tiers, such as edge, fog and cloud as illustrated in Figure 2 .

At the edge of the network, fixed and dynamically positioned IoT devices generate data that is transmitted over a public network. The generated raw data passes through multiple fog nodes and continues to the cloud data centres. Initial data pre-processing can be performed on the edge and fog nodes, however more complex AI tasks such as object detection and recognition (with greater demands on computing resources) should consequently be processed in cloud data centres. We propose the distribution of the (AI) data pipeline functionalities, such as (i) data collection, cleaning and data pre-processing – on edge and fog nodes, and (ii) object detection and recognition using deep learning models on fog and cloud infrastructure respectively, as illustrated in Figure 2 .

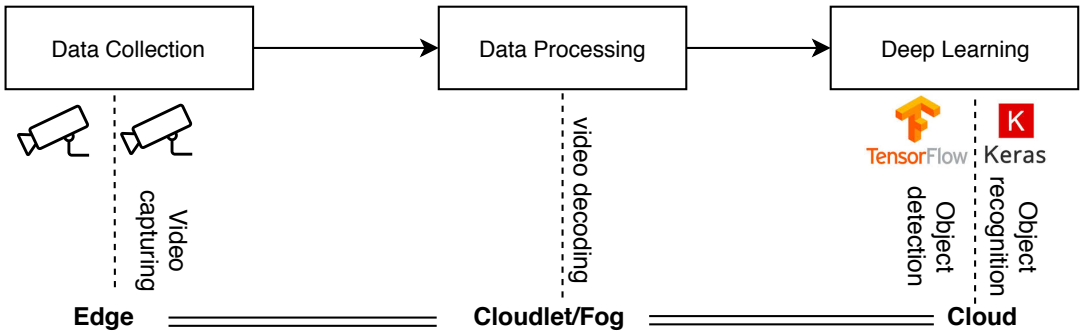


FIGURE 2 Multi-Tier AI Application Model Design and placement along Edge-Fog-Cloud continuum.

3.2 | Selection and description of Non-Functional Requirements

During the creation of cloud-native applications, a software developer selects which Non-Functional Requirements (NFRs) are important, such as network throughput, latency, CPU and memory utilisation, operational cost, etc. However, the type of NFRs to consider can vary between microservices. Furthermore, it is necessary to define which NFRs constitute a hard or soft constraint. Hard constraints are mandatory due to their essential influence on service operation, therefore they must be satisfied during run time uninterruptedly. On the other hand, soft constrains are not mandatory and tend to improve QoS. Additionally, for each microservice and the associated QoS metrics, acceptable threshold values/ranges are specified. NFRs can be understood as parameters with categorical, continuous and ordinal values[18] and as such are suitable for statistical analysis. In the current study, we have considered the following metrics, presented in Table 1 . Any hard and soft constraints are also passed to the decision-making algorithm for processing.

TABLE 1 QoS metrics and constraints and their description.

Metric	Description
Network Latency (L_{Inf})	The total round trip time between requests and response among deployment options.
vCPU utilisation ($vCPU_{Inf}$)	Usage of processing resources or computational capacity of a Virtual Central Processing Unit (vCPU).
Memory utilisation (Mem_{Inf})	The amount of memory used by a software application at a particular time instance.
Operational cost (C_{Inf})	Monetary cost of pooled virtual infrastructure entities, such as virtual machines in a Cloud system (USD/h)
Network Latency Constraint (L_{App})	Constraint on the network latency that specific application component needs to successfully communicate with other components.
vCPU utilisation constraint ($vCPU_{App}$)	Minimal number of processing resources needed for an application component to execute
Memory utilisation constraint (Mem_{App})	Minimal amount of memory needed for application component to process data
Operational cost (C_{App})	User constraint on the monetary price of on-demand resources
m	Number of infrastructure deployment options
n	Number of application components (microservices)

3.3 | Mapping Application's Components to the Infrastructure Deployment Options

Cloud-To-Things infrastructure can be represented as a multi-labelled graph. The Infrastructure graph, as seen in Figure 3 , contains multiple vertices that represent heterogeneous instances of processing nodes and possible hosting locations (options) for services. These locations can differ in their available resources (e.g. CPU and memory). Vertices are linked to one another with edges to form a complete or partially connected graph. The multi-labels on the vertices and edges are values that represent specific amount of compute resources, network-level parameters such as latency, and operational cost of the deployment options in the Infrastructure. On the other hand, a microservice-based application (Application) can be represented as a labelled directed acyclic graph (DAG). The vertices represent a microservice, and labels on the vertices represent hard and soft constraints on compute resources for each microservice. Labels on the edges represent dependencies between services, for example an average minimal latency. Mapping an Application graph to the Infrastructure graph for the purpose of efficient adaptation can be defined as a graph embedding or graph pattern matching problem, and is NP-hard. One way of solving this graph pattern matching problem is to utilise subgraph isomorphism. For this purpose, we have developed an algorithm for subgraph isomorphism based on Ullmann's algorithm [1].

Virtual infrastructure and microservice-based applications impose a graph structure, and application components can be placed on any node. However, under the consideration of QoS constraints and requirements, utilising subgraph isomorphism seems a natural choice due to many deployment combinations that clearly can correspond to subgraphs.

3.3.1 | Inexact label matching and constraint filtering

In our approach, an application (source) graph is defined as a $G_P = (V_p, E_p, Lv_p, Le_p)$ where V_p and E_p are sets of vertices and edges and Lv_p, Le_p denote labels on vertices and edges respectively. An Application or pattern graph contains n vertices and p edges. Similarly, the Infrastructure graph corresponds to the target graph $G_t = (V_t, E_t, Lv_t, Le_t)$ where V_t is a finite set of vertices and E_t, a finite set of edges where each edge connection goes from every vertex to the other and vice versa in such a manner that they form a complete graph. An Infrastructure or target graph has m vertices and r edges. For instance, the labels on vertices of the Infrastructure graph correspond to compute resources (e.g. CPU and memory) that each deployment option possesses and labels on edges for our particular experimental setup specifies network latency.

The algorithm initially searches for all feasible infrastructure deployment options. This is achieved by reducing the number of deployment options, based on filtering of label values on vertices and edges of both Application and Infrastructure graphs (see Figure 3). Adjacency matrices A and B for Application and Infrastructure graphs (see Figure 3) respectively are then specified, whereas each element in the adjacency matrix B of Infrastructure graph corresponds to 1 if the mapping of the vertices and edges fulfils QoS constraints of a particular microservice (or 0 otherwise), such that:

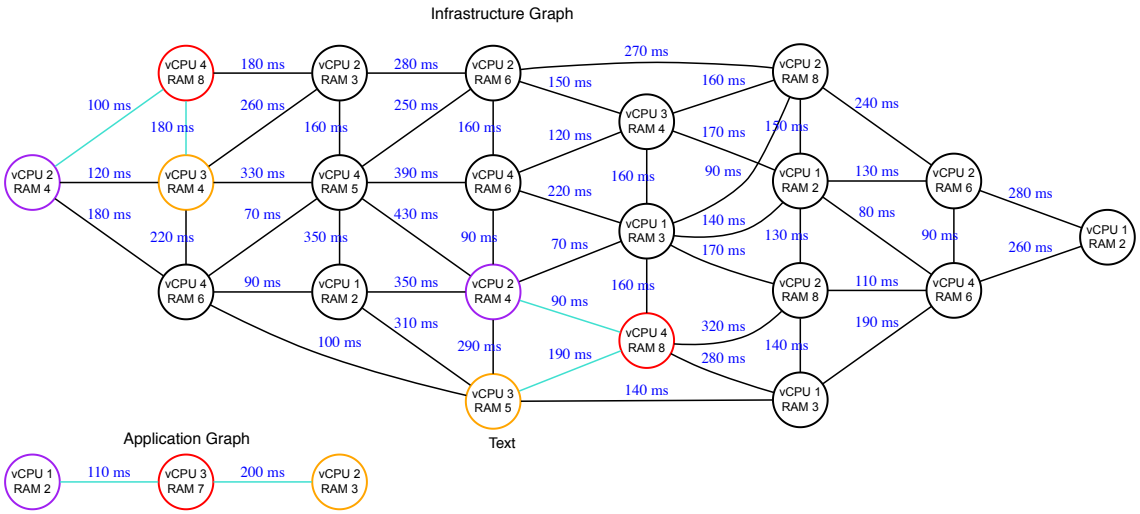


FIGURE 3 Microservice-based Application Mapping onto a Complex Infrastructure Graph.

$$m_{ij} = \begin{cases} 1 & CPU_P(i) <= vCPU_T(j);\ Mem_P(i) <= Mem_T(j);\ L_P(i) \leq L_T(j). \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The result of this step is reduced number of possible deployment options that fulfil QoS constraints of microservices associated with an application.

3.3.2 | Feasible infrastructure deployment options sets formation

In this step, a search is performed on the Infrastructure graph for resource combinations that are suitable for the deployment of the directed acyclic graph (representing the application). Formally, we use matrix M to permutate the adjacency matrix B of the Infrastructure graph to find all matrices C that correspond to the matrix A according to the equation: $C = [c_{ij}] = M(MB)^T$. Subgraph isomorphism exists, if we have found all feasible solutions, e.g. all possible combinations in Infrastructure graph to which Application graph can be mapped: if $C = [c_{ij}] = A = [a_{ij}]$.

Using this step, the search algorithm retrieves all subgraphs that represent a set of feasible deployment combinations based on application QoS constraints. As all solutions are feasible, and we need only one deployment combination, we rank the set of possible infrastructure deployment options to get only one optimal solution set.

3.4 | Ranking optimal Infrastructure deployment options and decision-making

The original algorithm by Ullmann is suitable for unlabelled graphs only, whereas graph pattern matching can also include other properties of vertices and nodes, including the labels on the vertices and edges. However, for our particular problem, we need to consider multi-label values on vertices and edges. Therefore, we have developed a core algorithm (see Algorithm 1) that returns a set of feasible deployment options and ranks them according to their score. From previous step, we have already obtained all possible combinations for placement options of an application microservices. For example, for application, constructed of two such services, the algorithm generates twopartite combinations.

To derive suitable infrastructure deployment options, the algorithm first sums up the cost of deployment options in all feasible solutions. Afterwards, the feasible combinations are ranked, and the algorithm takes the minimal value and calculates the difference between these and all other values of the remaining feasible combinations. The same procedure is used for the average latency metric: the algorithm sums up the average latency of all deployment options, and calculates the minimal value and the difference among minimum average latency and all other feasible combinations. The last step of the algorithm is to get the final score by summing up both differences of all feasible combinations: operational cost and average network latency. Afterwards, the results are ranked and the overall minimal value is considered as the solutions to use.

4 | SYSTEM ARCHITECTURE AND FLOW

This section presents a system architecture which makes use of *P-Match*. Our aim is to enable integration of *P-Match* into existing software engineering tools in order to support QoS-aware continuous adaptation of cloud native applications. The architecture is based on services and components that are part of the SWITCH Architecture[17]. The sequence of interactions between components of the architecture are shown in Figure 4 .

The Gateway interface is a Graphical User Interface that guides the developer through application composition. It can facilitate rapid development, where the application can be created by choosing and combining (pre-existing) components by dragging and dropping these components onto a canvas. Additional command line interface (1) can also be supported for assembling application logic. Once the application logic and flow is created and the developer confirms the application logic, the system generates YAML (YAML Ain't Markup Language) orchestration specification file (2) and passes it through Custom Resources to the Kubernetes Application Programming Interface (API) together with infrastructure requirements (3). In addition to infrastructure requirements, the user must also specify hard and soft constraints that each application component needs in order to function correctly. A user must specify hard constraints on compute resources for each component (4) and communication and dependencies among components as well, for example, maximal and *acceptable* latency between components to ensure inter-component data exchange/interaction. Application constraints can also be included in the YAML file, as well as requirements for the infrastructure. Furthermore, infrastructure requirements and constraint specifications are both defined in Custom Resources (5). Custom Resources are extensions of the Kubernetes API, making Kubernetes more modular and customisable. Custom

Algorithm 1 Redeploying Application Components (Microservices)

Input *Network latency (Latency), vCPU utilisation (vCPU), Memory utilisation (Mem), Operational cost (C), Latency constraint ($Latency_c$), vCPU utilisation constraint ($vCPU_c$), Memory utilisation constraint (Mem_c), Operational Cost constraint (C_c), number of deployment options (instance_num), number of regions (region_num), number of application components (m)*
Output *Optimal deployment configuration*

Initialise: $instance\_num, region\_num, m = 2$
for each r *in* $region\_num$ **do**
  for each i *in* $instance\_num$ **do**
    $instance\_list \leftarrow r, i$
  end for
end for
for each $inst_1$ *in* $instance\_list$ **do**
  for each $inst_2$ *in* $instance\_list$ **do**
    if $inst_1$ *in region* $\neq$ $inst_2$ *in region* AND
      $vCPU[inst_1] >= vCPU_{c1}$ AND $Mem[inst_1] >= Mem_{c1}$ AND
      $vCPU[inst_2] >= vCPU_{c2}$ AND $Mem[inst_2] >= Mem_{c2}$ AND
      $Latency(inst_1, inst_2) <= Latency_c$ **then**
      $feasible\_candidates \leftarrow inst_1, inst_2$
    end if
  end for
end for
$cost\_sum \leftarrow sum(inst_1, inst_2)$
$sorted\_cost\_sum \leftarrow sort[feasible\_candidates, cost\_sum]$
$min\_cost\_sum \leftarrow min[sorted\_cost\_sum]$
$sorted\_latency \leftarrow sort[feasible\_candidates, latency]$
$min\_lat\_sum \leftarrow min[sorted\_latency]$
$redeployment\_conf \leftarrow sum(min\_lat\_sum, min\_cost\_sum)$
return $redeployment\_conf$

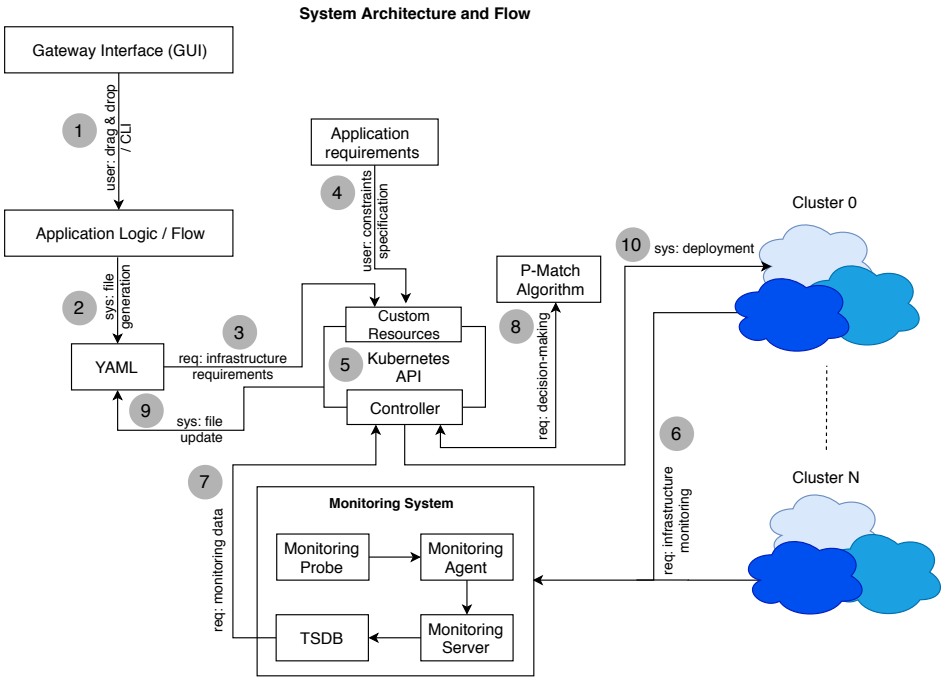


FIGURE 4 System Architecture and Flow.

Resources can be updated on a cluster during runtime, using dynamic registration independently of a cluster[19]. Before *P-Match* can be used, it is also necessary to carry out runtime infrastructure monitoring (6) for the metrics used in P-Match.

For this purpose, the SWITCH project[17] has implemented a monitoring system (7) that is composed of several sub-components. The monitoring probes are lightweight and decentralised components that are placed at Virtual Machines and containers in order to collect metrics from running applications. On the other hand, monitoring agents register monitoring probes

and manage data collection that represents the current state of the application and infrastructure from monitoring probes. Furthermore, the data is passed to the monitoring server and stored into a Time Series Database (TSDB). Based on this collected monitoring data, QoS metric values are passed to *P-Match* as an input data (8) through the Kubernetes Controller. After *P-Match* generates the provisioning plan for QoS-aware matchmaking for application components, the Kubernetes API on behalf of the Controller (5) updates the YAML orchestration file (9). The Controller must allow the deployment on a cluster within the selected geographical region. The updated YAML orchestration file is than sent to the Kubernetes API where it is used for the deployment and execution of the application using resources identified in the YAML file.

In comparison with similar cloud-based frameworks, such as CloudWave and Ubuntu Juju that are suitable for the creation, provisioning and deployment of microservice-based applications, we opted to make use of SWITCH. The latter offers a flexible and programmable[17] model that supports control of application logic. SWITCH also enables interaction with the underlying virtual infrastructure, and provides GUI-based specification of QoS-constraints for each application component that is hosted on such infrastructure. Additionally, SWITCH framework supports TOSCA-based orchestration and is integrated with the Kubernetes API. This makes SWITCH accessible by a significant user community, as both TOSCA and Kubernetes are widely used deployment environments for microservice-based applications.

5 | USE CASE SCENARIO

5.1 | The traffic management scenario

A traffic management scenario is used to illustrate the benefits of using both fog and cloud resources. A traffic management system needs to acquire data from fixed field sensors and autonomous vehicles in real time. A variety of fixed sensors are placed along roads and highways as road side units for detecting traffic conditions, such as traffic flow and congestion monitoring, vehicle density and incident detection, over-speeding etc. Additionally, autonomous vehicles communicate with one another and with road side units and transmit their information on speed and location for forecasting potential congestion and to estimate travel time[20].

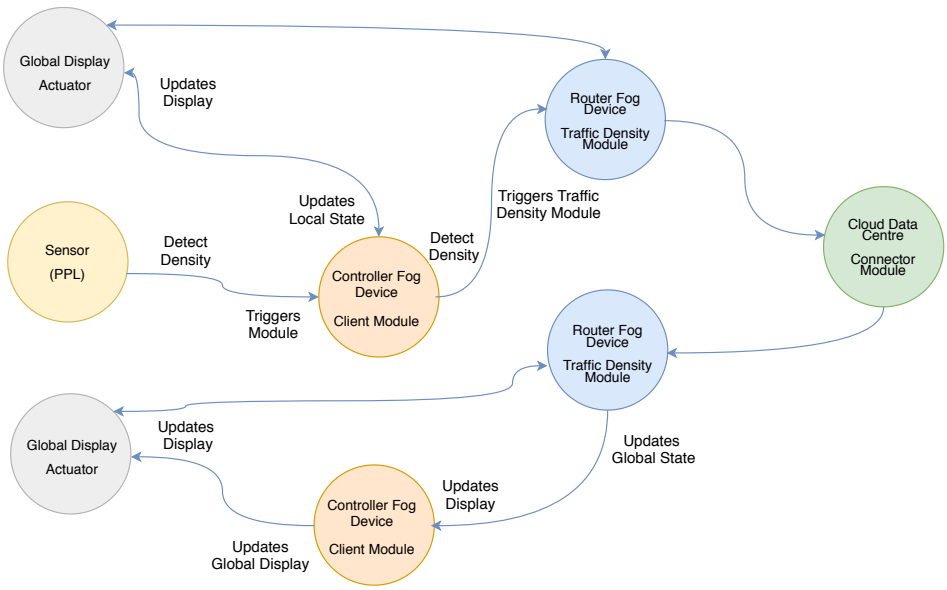


FIGURE 5 Traffic Management scenario: Integrating Fog and Cloud.

In this scenario, data sent from IoT devices to cloud-based systems must be processed and analysed in near real-time. Sending data to a cloud data centre means high latency and consequently longer response time for end users. On the other hand, processing data at the network edge and fog nodes offers low latency and consequently faster response to traffic events and therefore better application QoS.

The traffic management system includes the following components: (i) *Input sensors* are installed at fixed distance from one another on roads and collect and send data to a fog device for analysis; (ii) *Global Display Actuators* receive up-to-date responses from controllers or router fog devices and present parameter values such as lane closure signs or average vehicle speed, vehicle route displays and other similar messages; (iii) *Controller Fog device* is responsible for accessing compute, storage and network resources; it receives input from sensors and sends updates to the actuator; (iv) *Router Fog device* is a network router device linked to the controller fog device and enables multiple communication protocols. For instance, such devices can enable co-existence of several communication channels, using IEEE 802.11p, Bluetooth, Wifi-Direct, or low bandwidth protocols such as LoRAWAN and NB-IoT, which enable switching among those channels according to network availability and quality conditions[20]. Moreover, Router Fog device is also capable of supporting computational analysis; (v) *Cloud data centre* is responsible for maintaining the entire application state. The workflow showing interaction between these different components is provided in Figure 5 .

6 | EXPERIMENTS AND RESULTS

In the following subsections we describe the design and realisation of our experiments on Amazon Elastic Compute Cloud (Amazon EC2). Results and evaluation for *P-Match* method are also presented.

6.1 | Experimental Setup

We have run benchmark tests on 7 different Amazon Elastic Compute Cloud (EC2) regions worldwide[1] and considered two specific modules: (i) Client module and (ii) Connector Module (Cloud data centre) (see section 5). On each region we have utilised several on-demand instances that correspond to different deployment options.

The Client module collects data from sensors, such as information on traffic and lane state and stores the data into a database microservice. On each EC2 region, we have configured a test t2.small instance and deployed the microservice. The query requests were send from every test t2.small instance to all test instances on various regions. We measure the associated round trip time (RTT) or latency of the query request and response. This is how we build our Infrastructure graph for deploying both microservices. Monitoring data was collected over a one month period, whereby queries were sent every hour. For simplification of our experiments, we assumed that the RTT from specific instance at some region to the instances at the same region should be the same, since it does not correlate with other compute properties of the instances, such as CPU and memory capacity. For network latency we took the average value.

Additionally, to extend the experiments and consider a variety of other constraints, such as infrastructure-based metrics (CPU and memory) and operational cost of an instance we have configured several EC2 instances with different number of allocated compute resources and on-demand operational cost on all regions. Additional utilised instances are: t2.medium, m5.large, m5.xlarge, m5.2xlarge, m5.4xlarge. The majority of utilised on-demand instances are available in all regions, except t2.small and t2.medium are not available at ap-east-1 (Hong Kong) and me-south-1 (Bahrain) for now.

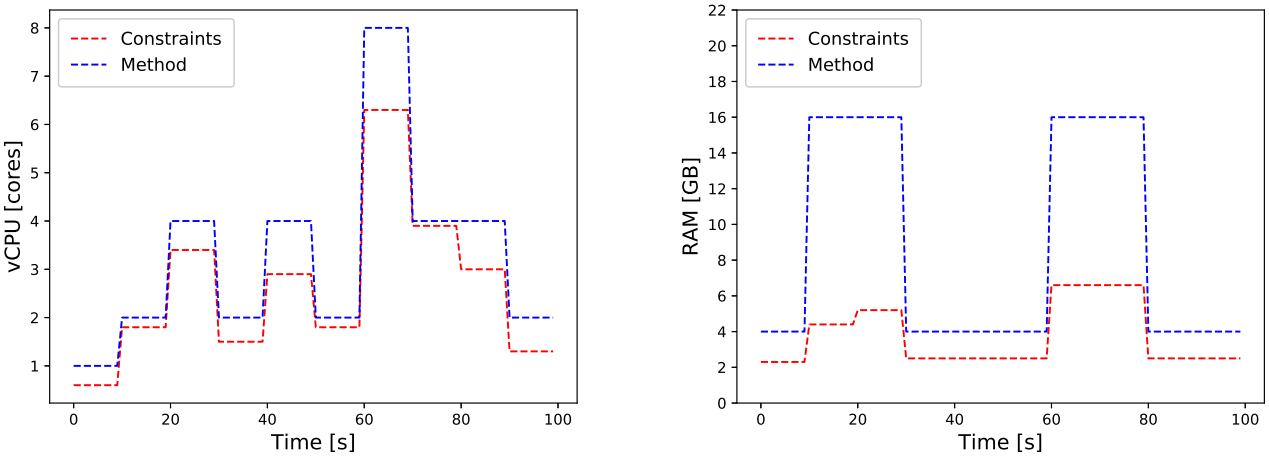
We then created two microservices with initial constraints on CPU and memory utilisation and network latency. Our main goal was to find the optimal deployment option for both microservices based on initial constraints at the same time. The present method was initially tested by using two dependent microservices, connected in a pipeline graph, with specific time-critical requirements – for example, low network latency between services. For both microservices that can be part of a traffic management scenario, we have simulated data exchange (file upload to one another) that should be transmitted as soon as possible in order to enable further data processing and analysis, under different workload. In our experimental setup, we have packed both microservices as Docker containers, making them highly portable and easily deployed across a number of different types of infrastructure.

6.2 | Results and evaluation presentation

We have considered 42 deployment options which differ in pre-allocated amount of compute resource and operational cost. For simplification of our experiments, we have stated that the deployment options located within the same region [2] have the

[1] The following data centres have been utilised: Ohio, N. California, Hong Kong, Canada (central), London, Bahrain and São Paulo.
[2]

same latency. The test software application was composed of two pipeline-based microservices. In order to define constraints on microservices' requirements, we have prepared deployment scenarios, whereas we set workload requirements, such as 800 and 1600 requests every 10 seconds. For our particular problem, we have generated the workload by using open source load generator, such as httperf[3]. We utilised Amazon CloudWatch and monitored the usage of virtual CPU (vCPU) and memory utilisation. Based on that information, we set constraints for both microservices, which are as follows: for microservice 1: CPU and memory utilisation < 40% and > 80% and for microservice 2: network CPU and memory utilisation < 20% and >60% and network latency <= 90 ms as dependency among both microservices respectively. For microservice 1, we applied workload 1 for which we assumed the following constraints: vCPU=2 cores, memory=4GB; and for microservice 2: vCPU=2, memory=3GB, and average max latency=90ms among the microservices.



(a) Adaptation of CPU utilisation for microservice 2.

(b) Adaptation of memory utilisation for microservice 1.

FIGURE 6 The line charts illustrate the change of a workload demand for CPU and memory utilisation for both microservices.

The results of the continuous deployment for *P-Match* are illustrated in Figure 6 , where red lines depict how the demand for more constraints on CPU (Figure 6 a) and memory (Figure (6 b)) utilisation have arisen according to the workload that changed every 10 seconds. On the contrary, blue lines show that at after every 10 seconds, the workload changes. Consequently, *P-Match* proposed infrastructure adaptation is specified, based on changes in CPU and memory utilisation.

We have compared P-Match method with MDP method of Kochovski et al.[2]. For this comparison, we have applied the same input parameters to both methods. Additionally, we applied another hard constraint to both methods, namely the geo-location: forasmuch as the placement was considered along one component per fog or cloud tier, we excluded the placement of more than one microservice within the same region[4] (see "x" in Table 2). In Table 2, the ids are distinct for each instance, yet marking the same order of the instances in every region, which goes in this order: t2.small, t2.medium, m5.large, m5.xlarge, m5.2xlarge, m5.4xlarge, for instance: id=0 is t2.small, id=1 is t2.medium, ... id=5 is m5.4xlarge[5].

6.2.1 | Results for MDP Method

In the second part of our evaluation process, we compare *P-Match* with the MDP method of Kochovski et al.[2]. MDP is a powerful stochastic decision-making method for dynamic environments, such as IoT environments. Its results are based on prior usage knowledge (i.e. deployment decisions) and current monitoring metrics (i.e. infrastructure and network level metrics). As a result, it is capable to predict probable future behaviour of the system and estimate an optimal deployment option that in the long-term

[3]https://github.com/httperf/httperf

[4]The latency within instances in the same region is considerably low and placing all microservices at the same region would be always chosen as best option.

[5]Due to page limitation we have excluded the description of the properties of on-demand EC2 instances. They can be found at: https://aws.amazon.com/ec2/pricing/on-demand/

is going to provide an efficient outcome. In order for the method to deliver a solution, it firstly generates a probabilistic model which is then iteratively refined. For the purposes of this study, the probabilities in the MDP method were derived from the knowledge gained during a usage period of one week, whilst the reward values were estimated using the number of threshold violations at deployment time.

The MDP model is defined as a tuple $M = (S, A, P, R, \gamma)$, where: S is a finite set of states (i.e. each state is a different deployment option), A is a finite state of actions (i.e. re/deploy action that is responsible for retrieving a solution and idle that is responsible for halting the iterations once a solution is found), P represents the probability values for transitioning between states, R is the expected reward values for transitioning between states and γ is the discount factor that enforces the method to deliver precise solution with less iterations. These elements play crucial role in the decision-making process and they are all described in details in the above cited study.

The MDP method generates a separate probabilistic model for the two deployments. Each model is composed of 42 states (i.e. deployment options) and 1764 transitions connecting all states to each other. The MDP algorithm first retrieves an optimal deployment option in the fog-tier and then, based on the first solution, it retrieves an optimal deployment option in the cloud-tier. The current version of MDP derives its optimal decision by iteratively updating the utility value (i.e. ranking score) for each state-action pair using the Bellman equation until the values converge. Hence, the MDP algorithm must compute optimal solution for 2 tiers and it generated two probabilistic models; it also has to perform the computation for each tier separately. As the best deployment option, for microservice 1, the MDP method returned the instance with Id=11 and for microservice 2 the instance with Id=23. As seen from Table 2, where the latency values from a single region to all other regions are presented, the MDP method has chosen instance Id=11 in region us-west-1 and instance Id=23 in region eu-west-2 with correspondent latency values of 77.437 ms and 77.417 ms as request and response values respectively. Due to the large number of potential deployment options, the MDP method delivered a solution (i.e. a pair of deployment options) in 310 ms on average.

6.2.2 | Results for P-Match Method

On the other hand, the *P-Match* for 42 deployment options checks 861 pair combinations if every node in cloud and fog infrastructure is linked to every node. The *P-Match* algorithm narrows down the search space by pruning infeasible nodes based on computer resources of both microservices first. After that, *P-Match*, similarly as MDP, first returns an optimal deployment option in the Fog-tier and based upon that decision retrieves an optimal deployment option in the Cloud. For the particular constraints, as the optimal deployment options the *P-Match* returns the instances with Id=3 and Id=21, which are located in region us-east-2 and ca-central-1 and the correspondent latency is 25.197 ms or 25.234 ms (request and response values respectively). The execution time for the *P-Match* was 180 ms on average.

6.2.3 | Discussion

Overall, a comparison of both methods reveals that for the particular set of constraints both methods proposed continuous deployment adaptation, that satisfies QoS requirements. Though both methods gave solutions that satisfy user requirements, *P-Match* gave better results for latency between the regions (see red- and green-coloured squares in Table 2).

Similarly, MDP has chosen instances with much more pre-allocated compute resources than actual demand (id=11 and id=23 are both m5.4xlarge) whereas *P-Match* has made more sustainable decisions (id=3 and id=21 are both m5.xlarge). The P-Match algorithm (see algorithm 1) has a time complexity of $O(n * m * d)$, where n and m represent the number of vertices of an Application and Infrastructure graph respectively, and d is the depth of a search tree[21] which equals the number of vertices V_p of an Application graph. On the other hand, MDP method is the finite horizon MDP that uses value iterations to converge to a solution. The time complexity is $O((S^2) * A)$, where S is the parameter that describes the number of states and A is the number of actions in the automaton[22,23].

Essentially the two methods have different background, one is stochastic (MDP) and the other deterministic (*P-Match*). Different functionalities may be derived from the stochastic and deterministic nature of the methods. Due to such facts, the MDP took more iterations than *P-Match*. However, these two methods can complement each other and provide more efficient decision on deployment and predict future behaviour of the system for increased reliability.

Based on this comparison, we can say that matchmaking is a powerful characteristic to include within a system architecture, as *P-Match* has reached identical or similar decisions to MDP. *P-Match* can also be extended to include additional QoS constraints if needed.

TABLE 2 Comparison of the MDP and *P-Match* methods based on the choice of the instances in specific region. The ids are distinct for each instance, yet marking the same order of the instances in every region, which is: *t2.small, t2.medium, m5.large, m5.xlarge, m5.2xlarge, m5.4xlarge*. Two similar values on latency stand for response and request query.

Instance IDs	AWS Regions	us-east-2 0–5	us-west-1 6–11	ap-east-1 12–17	ca-central-1 18–23	eu-west-2 24–29	me-south-1 30–35	sa-east-1 36–41
0–5	us-east-2	x	52.697	204.094	25.197	86.255	231.463	127.917
6–11	us-west-1	52.708	x	154.426	77.437	137.203	262.665	193.881
12–17	ap-east-1	204.077	154.397	x	202.405	199.994	123.410	314.669
18–23	ca-central-1	25.234	77.417	202.460	x	86.911	234.768	123.228
24–29	eu-west-2	86.354	137.150	203.956	86.916	x	150.467	193.505
30–35	me-south-1	231.481	266.829	127.567	234.706	150.425	x	340.051
36–41	sa-east-1	128.048	193.846	314.677	123.271	193.565	340.013	x

7 | IMPLICATION TO SCIENCE GATEWAYS AND FUTURE CHALLENGES

The presented approach can be used for the efficient distributed deployment and adaptation of pipeline functionalities along edge, fog and cloud tiers with consideration of non-functional requirements. Supporting job placement and submission to edge resources enables extension to existing Science Gateways, which often make use of cloud-based resources for execution. Supporting close-to-the-user enactment of jobs and data placement enables approximate analysis and what-if investigations to be carried out at lower latency, compared to execution on a remote data centre. A Science Gateway can also make use of the proposed approach to deploy services and support data migration in an automated manner. As a gateway can provide unified access to a number of different types of cyberinfrastructure and application libraries, the presented approach enables edge and fog devices to also be integrated in a seamless manner within such an environment. The optimisation approach outlined in this work enables dynamic: (i) deployment and adaptation of services at a cloud data centre and at fog resources concurrently, based on requirements that have been identified by a user via a Web-based interface; (ii) auto-scaling of services across edge and cloud resources based on constraints that have been identified by a user.

The optimisation approach identified in this work can be used to schedule and execute tasks to a particular timeline and to specific resources. In the centrepiece is a Gateway where a user submits tasks e.g. (data pipeline functionalities) and the system (with the present approach integrated) automatically identifies the number of instances and resources needed to execute the tasks. The user would be able to manage tasks and resources through a user-friendly interactive web-based portal for automated job submission, such as[24] – which primarily makes use of a Kubernetes environment.

Edge resources offer additional options for supporting services in Science Gateways, especially if such resources are deployed in proximity to a user, for instance:

- Inclusion of edge resources enables more effective use of computational infrastructure that is in proximity of a user. Requests being generated from a Science Gateway can be routed through edge resources – reducing potential latency of responses. An application orchestrator is identified in[25] to support this. The orchestrator acts as a proxy to generate a response to a user request (generated from a Gateway) using local resources in the first instance. If this cannot be achieved, the orchestrator dynamically can route the query to a data centre.

- Edge resources can carry out a number of functions to enhance the capability of a cloud system: (i) caching of commonly occurring requests & data; (ii) approximate response to user requests[26], which enables an approximate result to a user query to be returned to a user, without requiring a detailed response from a data centre.

- Depending on the type of edge resources being available, a service may be migrated from a data centre to an edge resources using the osmotic computing concept[27]. This approach enables a microservices to be migrated from a data centre to an edge resource closer to the user. The capacity (e.g. type of processor, storage available etc) indicates the type of microservices that can be hosted on the edge resource. The approach borrows from similar ideas used in data centres where multiple, pre-built VM images can be stored to deploy on different types of systems infrastructure.

8 | CONCLUSION

With the emergence of Internet of Things (IoT) that can generate large amounts of raw data, the need for processing and analysing of this data has become a challenge. Additionally, IoT applications that consist of a workflow pipeline can benefit from network edge devices to reduce latency and protect data privacy during data analysis. On the other hand, large-scale cloud data centres offer a greater number of (more complex) compute resources, support scalability and generally have better availability profiles. Due to the pipeline-based nature of IoT data processing, these applications are suitable for the distribution across edge, fog, cloud tiers – thereby making more effective use of resources at each of these tiers. We utilise workflow pipeline functionalities as linked microservices, with specific interdependencies between these services encoded in a graph. As an example, we consider a traffic management application scenario that describes how such services can be deployed across the three tiers mentioned above.

We presented a new algorithm for resource matchmaking that is based on Ullmann's algorithm for subgraph isomorphism, and which can be used for graph pattern matching. We represent cloud and fog infrastructure as a multi-labelled graph whose heterogeneous deployment options (e.g. instances) are actually labelled multiple vertices that differ in allocated compute resources. The multi-label edges represent mostly network-level parameters, such as network latency among all instances and operational cost of the deployment options. On the other hand, we represent microservice-based application as a labelled directed acyclic graph. The vertices represent an application component – in this instance, a microservice. The labels on the vertices represent hard and soft constraints, whereas the labels on the edges represent dependencies and requirements between components.

Evaluation of the proposed *P-Match* algorithm is carried out on Amazon Elastic Compute Cloud (EC2) platform, particularly on 7 cloud data centre regions worldwide. The evaluation of the method has shown that *P-Match* outperforms other methods in terms of how many QoS parameters are satisfied when adapting deployment options (locations) for application components. The results show that the present approach indeed returns optimal solutions and consequently proposes an efficient deployment plan. We compare *P-Match* with other approaches, such as MDP, and show that our method has a lower execution time and utilizes fewer iterations to converge. The use of the proposed approach leads to a sustainable consumption of on-demand compute resources improves application Quality of Service. Additionally, the present approach can be integrated into software engineering workbenches for the creation and deployment of cloud-native applications [17].

Our future research directions will be focused on orchestration, such as how to map the proposed deployment plan as part of the overall application workflow to the OASIS Topology and Orchestration Specification for Cloud (TOSCA) and dynamically update TOSCA-based workflow specifications. We will also explore how data learning models can be executed in parallel and consequently horizontally and vertically distributed across edge, fog and cloud infrastructure.

ACKNOWLEDGEMENTS

Funding: This work has received funding by the European Union's Horizon 2020 research and innovation program [grant agreement No 643963 (SWITCH project) and grant agreement No 815141 (DECENTER project)].

References

1. Ullmann JR. An Algorithm for Subgraph Isomorphism. *J. ACM* 1976; 23(1): 31–42. doi: 10.1145/321921.321925

2. Kochovski P, Drobintsev P, Stankovski V. Formal Quality of Service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method. *Inf. Softw. Technol.* 2019; 109: 14-25.

3. Simmhan Y, Aman S, Kumbhare A, et al. Cloud-Based Software Platform for Big Data Analytics in Smart Grids. *Computing in Science Engineering* 2013; 15(4): 38-47. doi: 10.1109/MCSE.2013.39

4. Legg PA, Chung DHS, Parry ML, et al. Transformation of an Uncertain Video Search Pipeline to a Sketch-Based Visual Analytics Loop. *IEEE Transactions on Visualization and Computer Graphics* 2013; 19(12): 2109-2118. doi: 10.1109/TVCG.2013.207

5. Pu Q, Ananthanarayanan G, Bodik P, et al. Low Latency Geo-distributed Data Analytics. In: SIGCOMM '15. ACM; 2015; New York, NY, USA: 421–434

6. Villari M, Fazio M, Dustdar S, Rana O, Ranjan R. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. *IEEE Cloud Computing* 2016; 3(6): 76-83. doi: 10.1109/MCC.2016.124

7. Osmotic computing as a distributed multi-agent system: The Body Area Network scenario. *Internet of Things* 2019; 5: 130 - 139. doi: https://doi.org/10.1016/j.iot.2019.01.001

8. Wu Y, Wu C, Li B, Qiu X, Lau FCM. CloudMedia: When Cloud on Demand Meets Video on Demand. In: ; 2011: 268-277

9. Zamani AR, Zou M, Diaz-Montes J, et al. Deadline Constrained Video Analysis via In-Transit Computational Environments. *IEEE Transactions on Services Computing* 2017; PP: 1-1. doi: 10.1109/TSC.2017.2653116

10. Garg SK, Versteeg S, Buyya R. A framework for ranking of cloud computing services. *Future Generation Comp. Syst.* 2013; 29: 1012-1023.

11. Wang M, Meng X, Zhang L. Consolidating virtual machines with dynamic bandwidth demand in data centers. *2011 Proceedings IEEE INFOCOM* 2011: 71-75.

12. Srikantaiah S, Kansal A, Zhao F. Energy aware consolidation for cloud computing. In: ; 2008.

13. Zhang L, Zhuang Y, Zhu W. Constraint Programming based Virtual Cloud Resources Allocation Model. In: ; 2013.

14. Mi H, Wang H, Yin G, Zhou Y, Shi D, Yuan L. Online Self-Reconfiguration with Performance Guarantee for Energy-Efficient Large-Scale Cloud Computing Data Centers. In: ; 2010: 514-521

15. Faticanti F, De Pellegrini F, Siracusa D, Santoro D, Cretti S. Cutting Throughput with the Edge: App-Aware Placement in Fog Computing. In: ; 2019: 196-203.

16. Aral A, Ovatman T. A Decentralized Replica Placement Algorithm for Edge Computing. *IEEE Transactions on Network and Service Management* 2018; 15(2): 516-529.

17. Štefanič P, Cigale M, Jones AC, et al. SWITCH workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications. *Future Generation Computer Systems* 2019; 99: 197 - 212. doi: https://doi.org/10.1016/j.future.2019.04.008

18. Kassab M, Daneva M, Ormandjieva O. Scope Management of Non-Functional Requirements. *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)* 2007: 409-417.

19. Kubernetes . Custom Resources. Online; accessed 5 April 2020.

20. Rana O, Shaikh M, Ali M, Anjum A, Bittencourt LF. Vertical Workflows: Service Orchestration across Cloud and Edge Resources. In: ; 2018: 355-362

21. Remec P. Integracija problema izomorfnega podgrafa v sistem ALGator. Master's thesis. University of Ljubljana, Faculty of Computer and Information Science. 2015.

22. Littman ML, Dean TL, Kaelbling LP. On the Complexity of Solving Markov Decision Problems. 2013.

23. Papadimitriou CH, Tsitsiklis JN. The Complexity of Markov Decision Processes. *Mathematics of Operations Research* 1987; 12(3): 441–450.

24. GitHub . Argoproj. Online; accessed 6 May 2019.

25. Petri I, Rana OF, Bignell J, Nepal S, Auluck N. Incentivising Resource Sharing in Edge Computing Applications. In: Pham C, Altmann J, Bañares JÁ., eds. *Economics of Grids, Clouds, Systems, and Services - 14th International Conference, GECON 2017, Biarritz, France, September 19-21, 2017, Proceedings*. 10537 of *Lecture Notes in Computer Science*. Springer; 2017: 204–215

26. Zamani AR, Petri I, Montes JD, Rana OF, Parashar M. Edge-Supported Approximate Analysis for Long Running Computations. In: Younas M, Aleksy M, Bentahar J., eds. *5th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2017, Prague, Czech Republic, August 21-23, 2017*IEEE Computer Society; 2017: 321–328

27. Villari M, Fazio M, Dustdar S, Rana O, Jha DN, Ranjan R. Osmosis: The Osmotic Computing Platform for Microelements in the Cloud, Edge, and Internet of Things. *IEEE Computer* 2019; 52(8): 14–26. doi: 10.1109/MC.2018.2888767

How to cite this article: Štefanič P., Kochovski P., Rana O. F., and Stankovski V. (2020),