IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 12, 2021, accepted January 25, 2021, date of publication February 5, 2021, date of current version February 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3057582

Systematic Approach for Generation of Feasible Deployment Alternatives for Microservices

ISIL KARABEY AKSAKALLI[1], TURGAY CELIK[2], AHMET BURAK CAN[3], (Member, IEEE), AND BEDIR TEKINERDOGAN[4]

[1]Department of Computer Engineering, Erzurum Technical University, 25050 Erzurum, Turkey
[2]MilSOFT Software Technologies Corporation, 06800 Ankara, Turkey
[3]Department of Computer Engineering, Hacettepe University, 06800 Ankara, Turkey
[4]Information Technology Group, Wageningen University and Research, 6706 Wageningen, The Netherlands

Corresponding author: Isil Karabey Aksakalli (isil.karabey@erzurum.edu.tr)

ABSTRACT Microservice architectures rely on the development of modular and independent software units, which typically address a single task and communicate with other microservices via well-defined interfaces. This has several benefits such as easier maintenance and update of services. However, deploying a microservice-based application is often more complicated than a monolithic application. While a monolithic application can be deployed one at a time on a group of similar servers behind a load balancer, a microservice-based application consists of different microservices and each microservice usually has more than one runtime instance that needs to be configured and deployed. For a small number of microservices and applications, the deployment could be done manually. However, a large number of microservices are frequently observed in practice. In such cases, the deployment becomes cumbersome and error-prone and does not scale with the increased number of services. To cope with this problem, we present a systematic approach and the corresponding tool support for enabling the deployment of microservices to resources that have limited capacity. Hereby, we model and define the design space given the deployment parameters and automatically derive the feasible deployment solution. The approach is validated using a taxi-hailing system case study inspired by Uber which has spread all over the world in recent years.

INDEX TERMS Automated deployments for microservices, capacitated task assignment problem (CTAP), deriving feasible microservice deployments, optimization algorithms.

I. INTRODUCTION

Cloud computing enables efficient and flexible use of IT resources, including well-configured computing power, unlimited storage, network capabilities, and managed services such as databases and messaging queues. With the use of cloud computing, infrastructure development and management costs can be substantially reduced by deploying software applications to remote cloud servers and using managed services for different needs such as object storage, messaging queues, and load balancing. In this way, companies can focus more on developing functional capabilities and dealing with architectural issues such as scalability and fault tolerance instead of spending effort on building and operating on-premise infrastructures. Although cloud platforms usually have a lower cost than on-premise systems, the cost of cloud resources still needs to be taken into account. To minimize unnecessary resource usage and reduce the cost, the cloud facilities should be used as efficiently as possible. On the other hand, the application on the cloud server needs to be scaled with the increased demand.

Despite the benefits of cloud computing, migrating monolithic applications to cloud appeared to be problematic in practice. Due to the lack of modularity, applications often need to be scaled entirely, which leads to unnecessary resource usages and increased cloud bills. To overcome the limitations of monolithic applications, microservice architecture has been proposed that decomposes applications into loosely-coupled modular services, which are highly independent, focus on realizing a single functional service, and can be separately maintained. The use of microservice architecture substantially supports scalability

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

in case of increased application workload and performance fluctuations.

Before microservices architectures arose, the tiered monolithic architectures were widely adopted. However, as the system grows and more developers work on a single code base, the architecture becomes complex and the software development life cycle iterations have begun to slow down. In addition to scalability, other different aspects such as keeping development and deployment iterations short, managing large databases, adding new features to the system, and dealing with fluctuating traffic become difficult, risky, and problematic in the monolithic architectures. All these problems forced some large corporate companies such as Netflix, Amazon, Uber, and Etsy to adapt to cloud-based microservice architectures. Today, Netflix's microservice architecture-based solution handles approximately two billion API endpoint requests processed by more than 500 microservices [33].

The microservice architecture design has obvious benefits with the increased modularity of services. On the other hand, the interactions of fine-grained services with a single functionality and complex configurations of runtime environments complicate microservices-based systems. Microservice interactions are generally designed to be asynchronous for the sake of scalability and have complicated invocation chains. For instance, Netflix has five billion service invocations per day and 99.7% of these calls result in cascading invocations of other microservices. Similarly, Amazon has hundreds of microservice invocations to render a page [38]. Besides, a microservice type may have thousands of instances running on different resources. Deploying these microservice instances with a feasible deployment configuration at a minimum cost is vital for using cloud resources more efficiently. Therefore, the deployment configuration of a large number of microservices on a group of available resources is a significant obstacle [24]. Deployment configuration of the services heavily affects the communication cost among services and utilization of computing power. The deployment configuration can be carried out using a team of human experts. However, in large systems, the number of microservice instances can dramatically increase to a scale that is not tractable by human experts, which makes deriving the most feasible deployment configuration harder. Furthermore, while a team responsible for creating and running a set of microservices is usually aware of the communication and computation costs of their services, it is difficult to estimate the impact of a change in a service on the other related services in practice [24]. For example, a change in a microservice can create additional communication overhead on other services that are consuming outputs of this service and may cause these services to have scalability problems.

In the literature, the scalability and design challenges of transitioning from a monolithic application to a microservice architecture have been widely addressed. However, to the best of our knowledge, the feasible deployment of microservices to cloud resources has not been explored in detail. In practice, deployment of microservices and configuration operations are manually performed by human experts, even with the use of some popular container technologies like Kubernetes [23], Docker Swarm [14], Apache Mesos [1]. However, as explained above, finding a feasible deployment is not a trivial task. If the deployment cannot be performed according to the given configuration, reconfiguration takes a long time depending on the size of the system. To cope with this problem, more systematic and formal approaches are needed to find feasible deployment alternatives.

This study provides an approach that systematically derives feasible deployment alternatives from application design and available resources by using Capacitated Task Assignment Problem (CTAP) solver algorithms. The approach derives feasible deployments of defined microservices on physical resources that have different memory capacities and computation powers. Initially, the microservice types, the model for data exchange among microservices, and available physical resources to deploy the microservices are designed. This design is then used for defining sample runtime execution configurations that provide dynamic properties of the system such as the number of instances of each microservice type and update rate of specific data for each producer/publisher service. After designing the Microservice Runtime Execution Configuration Model and the Microservice Infrastructure Model containing physical resources, deployment alternatives for the microservices are derived algorithmically. The approach is realized with a tool that supports the automatic generation of feasible deployment alternatives. The proposed approach is validated using an industrial case study simulating a taxi-hailing system inspired by Uber's microservice architecture [35].

The contributions of this proposed approach are described as follows:

- The proposed approach provides a set of deployment alternatives to the development team with different total communication and execution costs at the early design phase before the system is fully developed and deployed.
- Since the cost of change increases exponentially in later phases of the development lifecycle, the approach provides foreseeing the design problems such as excessive service interactions, high memory requirements, etc. at early design phase.
- As the constraints of deployment operation, communication costs between services, execution costs of services on servers, the memory capacity of services, and servers are taken into consideration. After adapting these constraints to the well-known Capacitated Task Assignment Problem (CTAP) [29], the approach enables utilizing different solvers for generating alternative deployment models at the design phase. Finding a deployment alternative using this approach is completed in a much shorter time than expert judgment in a system consisting of thousands of microservices.
- A toolkit that supports the realization of each step of the approach including microservice design,

analysis of the generated deployment model, comparison of alternative deployment approaches is designed and developed. Furthermore, the toolkit supports manual deployment models to enable a performance comparison of a human-designed deployment model with automatically generated deployment models.
- The toolkit allows porting different CTAP solvers to derive feasible deployment alternatives. We ported the genetic algorithm proposed by Mehrabi *et al.* [27] to the tool as a sample CTAP solver.

The rest of the paper is structured as follows. Section II provides background and context. Section III describes the case study to be used in successive sections. Section IV gives the problem statement. Section V explains the approach and detailed algorithmic solutions for evaluating alternative deployment options. Section VI briefly describes the tool support for the approach. Section VII summarizes the studies in the literature that address the deployment approaches for microservices, and finally, Section VIII concludes the paper.

II. BACKGROUND AND CONTEXTS

In this section, the necessary information for understanding and supporting the proposed approach is given. In Part A, several alternative microservice deployment patterns are explained. Part B defines the reference architecture for the deployment pattern that we focused on in this paper.

A. MICROSERVICE DEPLOYMENT PATTERNS

There are several methods for the deployment of microservices to nodes that represent computation resources such as a virtual or physical server machine. These methods are (1)''Multiple service instances per node'', (2) ''One service instance per node'' and (3) ''Serverless deployment'' [31].

In multiple service instances per node, more than one service instance is deployed on each physical or virtual server. Instances of each service can run on one or more servers with a known set of reserved I/O ports. This approach reduces overall communication costs since multiple communicating instances share the same environment. The most important disadvantage of this model is that it has little or no isolation of service instances with respect to the shared resources. This might cause security or reliability problems in case of service misbehavior or malfunction.

To avoid the isolation problems, ''one service instance per node'' approach has been proposed. In this model, each service instance runs separately on its node to enable service isolation. This model is specified in two forms: one service instance per Virtual Machine (VM) and one service instance per container. In one service instance per virtual machine model, each service is packaged as a VM image such as an Amazon EC2 AMI [2], and each service instance is initialized using the VM image. Each service instance has isolated memory space and CPU, and it is executed independently. It provides better security and reliability in comparison with the multi-service per node approach since a security exploit or malfunction in a service is isolated in the corresponding node. The major disadvantage of this model is resource usage efficiency since each service instance incurs the overhead of the entire VM with the OS. Moreover, VMs are also typically slow to start/restart due to the same reason. The structure of the ''one service instance for per VM'' is shown in Figure 1a.

Due to the disadvantages of VMs, ''one service per container'' has emerged as a more lightweight virtualization alternative. In this deployment model, each service instance runs within its own container. A container is a virtualization mechanism that runs at the operating system level. Containers have their own port namespace and root filesystems. Memory, CPU, and I/O resources of containers can be limited. The advantages of the containers are similar to VMs. It isolates the service instances from each other and enables easy monitoring of resources consumed by each container. Container technologies generally provide a container management API to manage container instances [31]. Containers also encapsulate the technology that is used to implement services and provides a secure environment like VMs. Nevertheless, unlike VMs, containers are lightweight technologies, and creating and initializing a container image is often very fast. The structure of the ''one service instance per container'' is shown in Figure 1b.

In addition to all these microservice deployment strategies, there is a relatively new approach named the serverless deployment model. This approach prevents the user from having to choose between service delivery on a container or virtual machines. For example, AWS Lambda [4] is one of the serverless platforms that support developing micro applications with different languages like Node.js, Java, and Python. While deploying a microservice, the service is packaged as a ZIP file and uploaded to the AWS Lambda platform. AWS Lambda automatically scales up and down the number of microservice instances according to load. The user is billed according to the number of handled requests. Although it reduces deployment problems for developers, unsuitability for long-running services, forcing stateless and idempotent design of services, vendor lock-in on the available technologies are some disadvantages of the serverless deployment [31].

B. MICROSERVICES REFERENCE ARCHITECTURE

Among the microservice deployment models in the literature, we focus on ''one service instance per container'' model since it is more lightweight and popular than one service/multiple services per VM approach, and the serverless architecture is not suitable for all kinds of services as explained in the previous section. Figure 2 shows the reference architecture for one service instance per container approach. Despite our focus on the ''one service instance per container'' model, the proposed approach is also applicable to ''one service instance per VM'' model.

At the highest level, a microservice-based distributed system consists of several interconnected physical servers/VMs. Each physical server/VM hosts one or more containers. Each container hosts one microservice instance with a well-defined

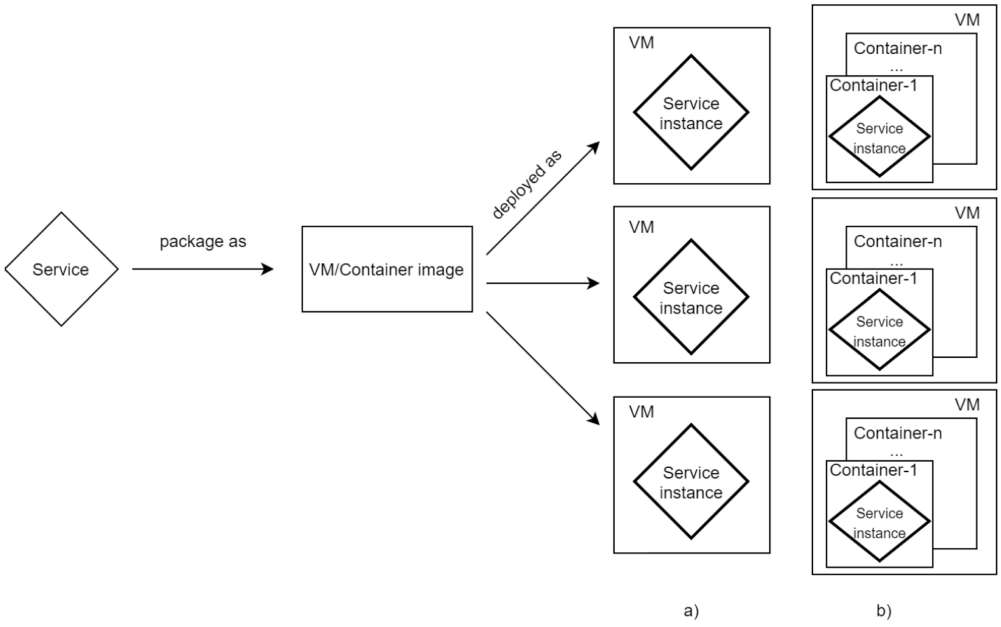


FIGURE 1. a) One service instance per VM b) One service instance per container.

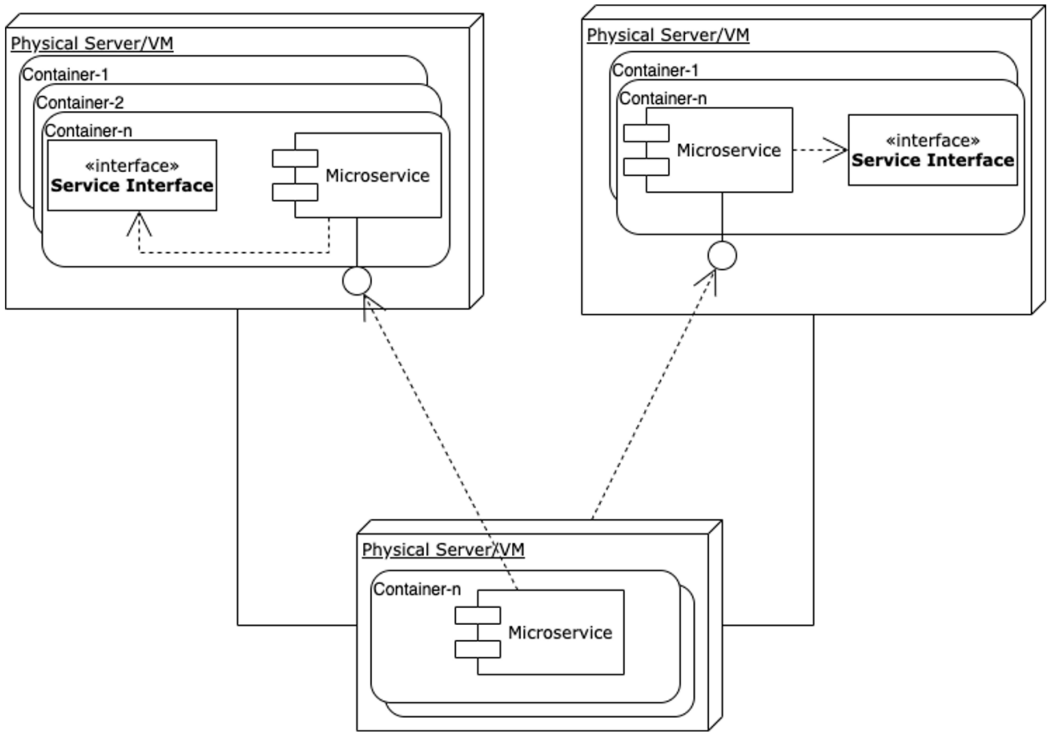


FIGURE 2. Microservices reference architecture.

interface. The services must be packaged as a container image in a ''one service instance per container'' model. A container image is a file that consists of an image of applications and libraries which are necessary to run the service. After the service is packaged as a container image, one or more containers can be initialized. Generally, multiple containers run on each physical or virtual server. Some cluster managers such as Kubernetes [23], Docker Swarm [14], and

Marathon [25] can also be used to manage containers. A cluster administrator treats servers as a resource pool. It decides where each container should be deployed depending on the provided configuration.

Microservice instances communicate and exchange data with each other via their interfaces which can be based on different protocols such as RESTful services [39], gRPC [19], publish/subscribe communication channels such as OMG DDS [40], Amazon SNS [41] and message queues protocols such as Apache Kafka [42], Amazon SQS [43], RabbitMQ [44], ActiveMQ [45], Java Messaging Service [46], etc. Besides, both synchronous and asynchronous communication patterns can be used between services according to needs [47]. In fact, this hybrid communication architecture is one of the biggest advantages of the microservice approach compared to the plain Service-Oriented Architectures. By using synchronous communication methods such as HTTP, gRPC, and REST, a service sends a request to another service and waits until the corresponding service responds. When the data is needed immediately, the request/response communication patterns are suitable. However, there may be situations where the requested service is not available or can't respond immediately. In this case, to prevent delays that may cause scalability issues, non-blocking asynchronous communication can be preferred among services by using a message queue or publish/subscribe pattern. Hereby, both synchronous and asynchronous communication patterns can be utilized by using different communication protocols within the same system at microservice architecture.

III. CASE STUDY- TAXI HAILING SYSTEM

In this section, we present a case study describing a taxi-hailing system inspired by Uber [35] to illustrate the problem statement and evaluate the proposed approach. Just like most of the organizations, Uber [35] initially developed this system using a monolithic architecture [30]. Since it was used for one city at the time, a monolith built from a single code base could solve Uber's fundamental business problems. The monolithic taxi-hailing system has a REST API that provides passengers and drivers to be connected. When booking a taxi, three different adapters used with this API enable billing, payments, and email/message services [30]. The system stores all data in a single database. Thus, all features such as passenger management, driver management, notification features, payments, trip management, and driver management are provided by a single application.

As Uber's taxi-hailing system begins to expand worldwide, a single monolithic codebase caused scalability and CI (Continuous Integration) problems [30]. For instance, to update a single feature in a monolithic architecture, all features must be rebuilt, tested, and deployed repeatedly. Additionally, finding the source and fixing of bugs in a single repository becomes difficult for a developer. Furthermore, concurrent updates of existing features or adding new features by different developers requires serious coordination. Besides the development problems, operation of monolithic applications is also hard. For example, scaling a feature on demand requires scaling up all features together which increase the scalability costs in the monolithic architectures. Because of these problems, Uber [35] decided to change its architectural style and adopted microservice architecture by following other popular companies like Netflix [26], Amazon [5], Twitter [28], etc. Figure 3 shows Uber's taxi-hailing system [35] designed with microservice architecture. This architecture connects eight different microservices including billing, payments, passenger management, driver management, notifications, trip management, passenger Web UI, and driver Web UI through an API Gateway. Since every service is isolated as much as possible, a change in one service requires deploying only that service, and also each service can be scaled individually.

The number of service instances varies according to the instant requests to these services. For example, since the number of taxi seekers is higher than the number of taxi bookers and payers, the number of microservice instances in different services also varies according to the demand. This situation requires the number of passenger management instances to be more than the number of payment instances. Besides, the frequency of communication among the services varies according to the number of instant requests and different communication types can be used between services. When a user requests a trip, the following steps will be triggered: Firstly, the Trip Management service is notified via the API gateway. The Trip Management service requests passenger information from the Passenger Management service with the request/response synchronous communication type. After obtaining this information, Trip Management notifies the Dispatcher about trip details. Then, the Dispatcher finds an available driver and notifies both Passenger Management and Driver Management services simultaneously through the pub/sub communication channel. Finally, Passenger Management and Driver Management services request the Notification service to inform the passenger about payment and trip details [48].

We used the system described above to define a case study to validate the approach that we propose. We defined the number of microservice instances as shown in Table 1. Besides, the memory requirements of nodes and microservices, the execution costs of the microservice instances on the nodes, and the frequency of communication with each other are defined by the designer.

IV. PROBLEM STATEMENT

One of the key issues affecting the performance of microservice-based software in a cloud environment is the allocation of microservices to available resources. For small or medium-sized applications with a limited number of services and nodes, service deployment can be efficiently

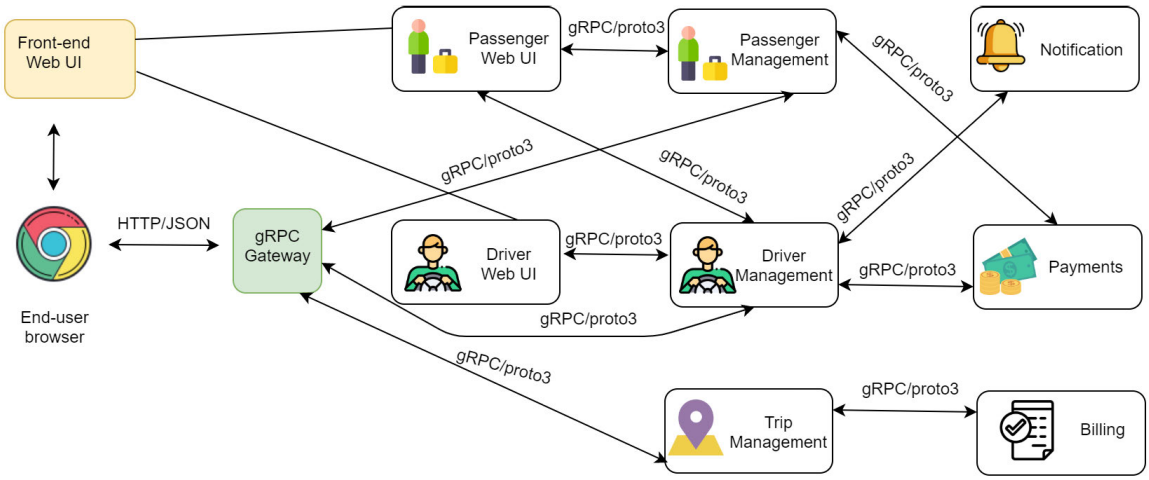


FIGURE 3. Microservice architecture of a taxi hailing system [35].

TABLE 1. Sample scenario for a taxi-hailing system with defined number of instances per microservice.

Microservice name	Number of instances
Billing	50
Payment	15
Notification	125
Driver Web UI	30
Passenger Web UI	25
Driver Management	100
Passenger Management	150
Trip Management	55
Total	**550**

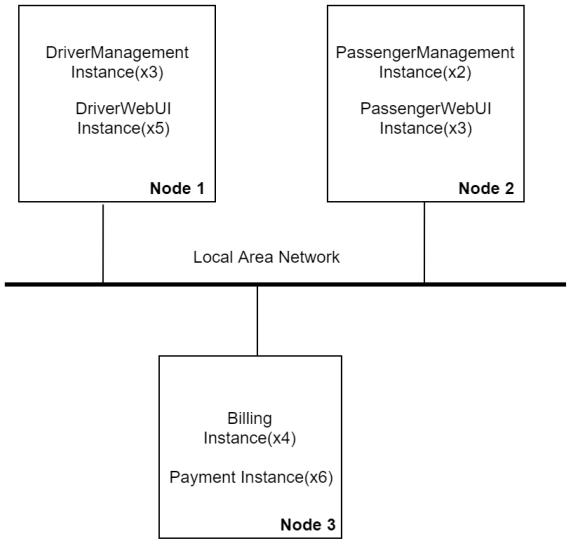


FIGURE 4. A deployment alternative by grouping the same type of service instances.

performed by a human expert who knows the overall system very well. However, existing software systems that migrate to microservice architectures are often large-scale and require a large number of services and nodes. This makes it difficult to perform a feasible deployment process manually for the human expert. Additionally, as the system expands, finding the minimum cost for the deployment process becomes impossible. Considering many parameters such as memory capacities of nodes and services, communication costs between services, and execution cost of services on nodes, various deployment alternatives can be derived for a sample scenario. For example, a deployment alternative for the taxi-hailing system can be defined with driver instances, passenger instances, and billing instances as shown in Figure 4. In this case, six different types of microservices with a total number of 23 instances are deployed to three separate nodes. In this case, the communication cost among driver instances is assumed to be zero, as well as the cost of other instances on the same node. If there is a high frequency communication between two microservice instances deployed on different nodes (e.g., the driver and passenger), the deployment should be re-evaluated in terms of performance and must be compared with different alternatives.

The second example deployment alternative is shown in Figure 5. Here, eight different types of microservices with a total number of 28 instances are deployed evenly among 4 nodes. This means that two different types of services are allocated to each node. Although this deployment approach is simple and easy to understand, the two services assigned to the same node may not communicate frequently with each other, or the services assigned to different nodes may often need to communicate with each other. Another problem is the efficient use of resources. For example, fully utilizing a node with low memory or under-utilization of a high capacity node may adversely affect the system performance and operation cost. Therefore, this deployment alternative may not be feasible in terms of minimizing total communication cost and efficient use of memory capacity.

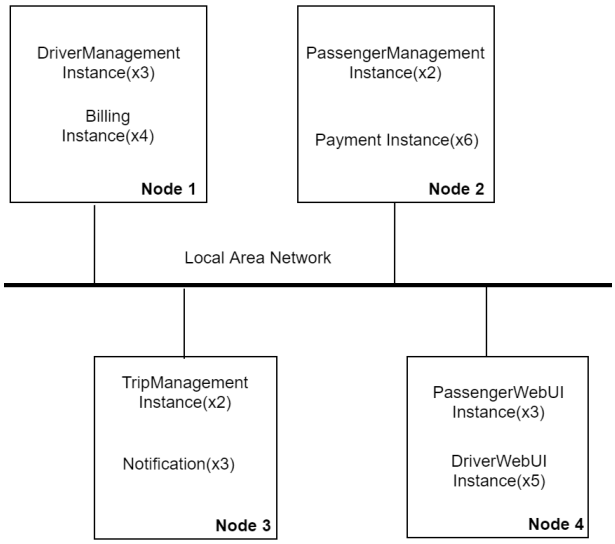


FIGURE 5. Second deployment alternative by grouping services evenly.

Like the deployments given in Figure 4 and Figure 5, various other deployment alternatives can be derived by taking into account different parameters such as number of available nodes, number of services to deploy, communication costs among services, execution costs of microservices on each node. It is hard to obtain optimal performance among these alternatives manually. It is also evident that each deployment alternative can be generated differently in terms of many quality considerations, such as logical allocation for understandability, overhead optimization, and increased use of physical resources. Therefore, a more systematic and formal approach is needed for the automatic generation of deployment alternatives that are very difficult to determine by human experts. Additionally, the automatic comparison of these alternatives with each other is needed in terms of performance. As a deployment approach, container technologies are often used for the deployment of microservice applications in the industry. However, these technologies need configuration files prepared by the user to define the deployment configuration. Obviously, there is no clear approach for guiding the deployment of microservices and resource allocation to optimize the performance in the design phase. Moreover, it is seen that no qualified approach or tool support is proposed in the literature for the selection of deployment alternatives. In the following sections, we describe our approach and tool support for deriving feasible deployment alternatives for microservice-based applications.

V. APPROACH FOR DERIVING AUTOMATED FEASIBLE DEPLOYMENT ALTERNATIVES FOR MICROSERVICES

In this study, we present a concrete approach to identify and evaluate feasible deployment alternatives for microservice-based systems. The proposed approach can be used in the early design phase before the coding and development of the system. Generating alternative deployment models according to runtime scenarios and analyzing the overall system performance at the design phase helps to avoid further deviations and re-work at the following development activities such as detailed design, implementation, testing, and operation, etc.

A. PROCESS STEPS FOR DERIVING FEASIBLE DEPLOYMENT ALTERNATIVES FOR MICROSERVICE APPLICATIONS

We defined a high-level process for feasible deployment alternative generation as follows:

1) DATA EXCHANGE MODEL DESIGN

As the first step, the team extracts a model defining the data types and object sizes to exchange data between microservices during communication.

2) DECOMPOSING THE SYSTEM INTO MICROSERVICES

At the second step, the team decomposes the system into microservices and assigns requirements to these services. It should be noted that not all the services have to be new services. Generally, the team has some legacy services that can be ported to the new architecture. The team needs to define both new and legacy microservices in the design tool to enable feasible deployment generation.

3) COMMUNICATION PATTERN DESIGN

In this step, the team decides the communication pattern among the microservices defined in the second step and the data exchange objects between microservices designed in the first step. As we mentioned before, all of the microservices do not have to use the same communication protocol for data exchange, but we defined a superset data model to assess communication cost in a uniform manner. We are going to give more detail about this uniform data model in the further sections of the paper.

4) DEFINING THE PHYSICAL RESOURCES

In this step, the team defines available resources on the design tool. The physical/virtual servers are defined by means of CPU and memory power and network connection among them.

5) DESIGNING SAMPLE EXECUTION SCENARIOS

In this step, the team defines runtime scenarios according to the microservices and data exchange methods/objects defined in the first two steps. Runtime scenarios define the instance count of each microservice and data update/fetch rates for each communication channel among services. Besides, the execution costs of microservice instances on available resources are defined.

6) GENERATING THE FEASIBLE DEPLOYMENT ALTERNATIVES

After the design of microservices, the data exchange objects, the communication protocols, and the physical infrastructure,

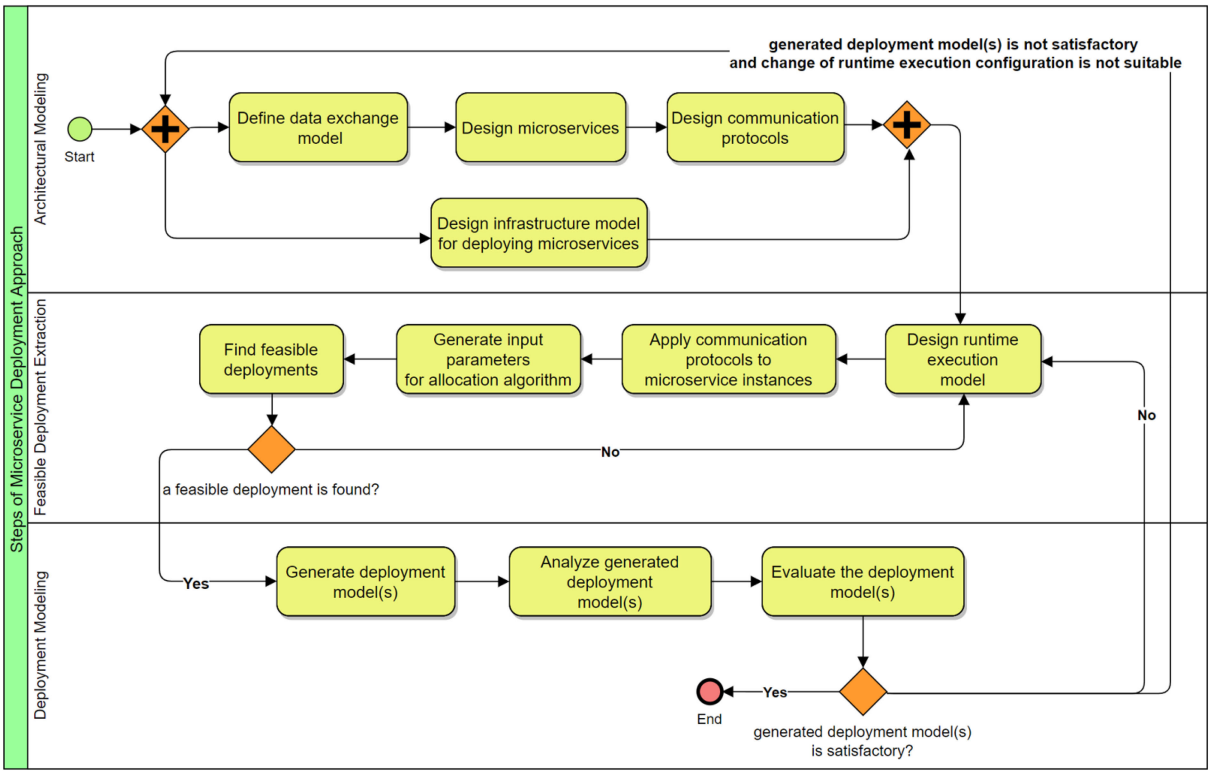


FIGURE 6. Business Process Model and Notation (BPMN) diagram of the proposed approach.

the tool automatically extracts deployment model generation algorithm parameters from the design, executes the CTAP (Capacitated Task Assignment Problem) solver algorithm, and generates the feasible deployment alternatives.

7) ANALYZING THE RESULTS AND COMPARING THE GENERATED MODELS WITH EACH OTHER

At this step, the tool generates a benchmark report for the generated deployment alternative exposing intrinsic information such as the services that generate high traffic load. At this step, the tool also provides the capability to compare two deployment alternatives which can be generated by the same algorithm with different runs, another algorithm or an expert's judgment.

The BPMN diagram of the proposed approach is shown in Figure 6. Firstly, a Microservice Data Exchange Model is designed to define data objects and object sizes. Then, microservices are designed and communication protocols are defined by using the identified Microservice Data Exchange Model. In addition, the communication channels that the microservices will interact with are determined and these channels are modeled in the communication protocols model. Besides these models, the Microservice Infrastructure Model that will host the services can be created independently from the Microservice Data Exchange Model, microservice definition, and communication protocols.

Once the design phase of the system architecture is completed, the generation of a feasible deployment alternative phase begins with the definition of the Microservice Runtime Execution Configuration Model. This model defines the instance count for each microservice and the data update rates for each communication channel among services by using the artifacts defined in the architecture design phase. After all the models are developed, input parameters are generated from these models to feed the feasible deployment model generation process. If the system cannot find a feasible deployment alternative according to given parameters, it directs the designer to the initial steps of the process for updating the models. The designer can modify the data objects, split or merge them, remove unnecessary communication channels among microservices, or even merge/split microservices for better cohesion and low coupling. If a deployment alternative is found, the system generates a deployment model based on the output of the deployment model generation algorithm and presents this alternative to the designer.

Deployment modeling part of the proposed approach represents the seventh process step for analyzing and comparing the generated deployment models. In this part, the system generates a deployment model analysis report that represents insights such as most communicating services and the amount of exchanged data for each communication channel. The system also enables the automatic comparison of different deployment alternatives in terms of communication and

execution costs. This allows designers to analyze the generated deployment models with comparison reports. According to the results of these reports, the designer decides whether the generated model is acceptable or not. If the designer thinks that there are still issues for updating the design or physical resources to further improve the communication and execution costs, he/she can return to the beginning of the design phase and repeat the process.

From a general perspective, finding a feasible deployment alternative for a multi-task system is a combinatorial optimization problem handled in the optimization branch of mathematics. This problem overlaps with the Capacitated Task Assignment Problem (CTAP) in the literature. To implement the task assignment problem, firstly, the input parameters must be clearly defined. These parameters are extracted from the system design, including memory capacities of the available processors, the execution costs on the processors of each task, and the communication costs between tasks. In the proposed approach, tasks correspond to microservices and processors correspond to resources or nodes. Once the necessary parameters are extracted, the tool that supports the approach generates feasible deployment models using the optimization algorithms. Then, the feasibility of the generated deployment models is evaluated in the next step. If the generated deployment models are not satisfactory, an iteration step will be required for analyzing the system design and correcting it according to the feedback provided by the proposed tool. The feasible deployment alternative is determined by minimizing the total cost for the deployment model (e.g., communication and execution costs) without violating the memory capacities of each deployment node. Finding feasible deployment models may require many iterations of the operation steps. The initial deployment model is realized during the development and integration/testing activities, the results are reported back to the designer, the design is revised until a satisfactory alternative is obtained.

In the following sections, the concrete activities defined for the implementation of the proposed approach are described. In each section, the design of the metamodels required for the approach is addressed. Additionally, the relationship between the metamodels that depend on each other is explained.

B. DESIGN MICROSERVICE DATA EXCHANGE MODEL

The Microservice Data Exchange Model defines the data model required to exchange data between microservices. Microservices need to communicate with each other to exchange data. When communication protocols for microservices are investigated, it is seen that there are many communication infrastructures such as gRPC [19], REST [13], graphQL [14], publish/subscribe [6]. In the proposed approach, the data model of the most comprehensive communication infrastructure for microservices is investigated to ensure the applicability of all communication methods. In this context, the gRPC protocol has been implemented since it has the most comprehensive data type set.

The gRPC protocol developed by Google, known as the high-performance remote procedure call, is a modern open-source RPC framework that can execute in any environment [19]. With plug-and-play support for load balancing, monitoring, health control, and authentication, it can effectively connect services within and across data centers. gRPC can use protocol buffers as both in the Interface Definition Language (IDL) and the basic message exchange formats [49]. In proto3, the latest version used in the gRPC, *Message Type* is used to define the call request in message format and the class defined with the variable types in the.proto file is called the *Scalar Value Types*. Therefore, variable types such as *float*, *int32*, *uint32*, *sint64* defined in the Design Microservice Data Exchange Model are extended from the *ScalarValue* class. Furthermore, like other communication protocols, the *Enumerated Data Types* can be defined in this protocol. Other different data types such as *ReservedValue*, *NullValue*, *Map*, and *ArrayDataType* are also extended from the *DataType* class. *ObjectModelElements* inherits some features such as *Struct*, *Any*, and *Oneof*. *Struct* represents any JSON object with the *Message Type*, Any allows a user to send messages as embedded types without.proto definitions and *Oneof* feature is used in a place where only one field can be set at the same time in a message with many fields. A part of the ecore diagram of the Microservice Data Exchange Model is shown in Figure 7. The full ecore diagram of this model is shown in Appendix A.

C. DESIGN MICROSERVICE DEFINITION MODEL

The Microservice Definition Model consists of microservices defined as application participants. Microservices are important artifacts that must be designed for the whole system to work. In the scenario given in Figure 4 and Figure 5, services such as *PassengerManagement*, *DriverManagement*, *Billing*, *Payment*, *Notification*, etc. are microservices. Since microservices communicate with each other using various communication protocols, the components (*Microservice*, *MicroserviceType*, *Version*, *MicroserviceRepository*) of this model are defined in the Microservice Communication Model shown in Figure 8.

D. DESIGN MICROSERVICE COMMUNICATION MODEL

The Microservice Communication Model contains the connection channels which are established for the communication of microservices based on data exchange. The microservices defined in the *Design Microservice Definition Model* communicate with each other using the data objects defined in the *Design Microservice Data Exchange Model*. The ecore diagram of the microservice communication model with various communication patterns is shown in Figure 8.

As shown in Figure 8, common communication methods described for microservices are divided into four groups named *RestOperation* [13], *graphQL* [18], *gRPC* [19], and *pub/sub* relation [6]. Among these communication methods, *gRPC* [19] is the most comprehensive communication

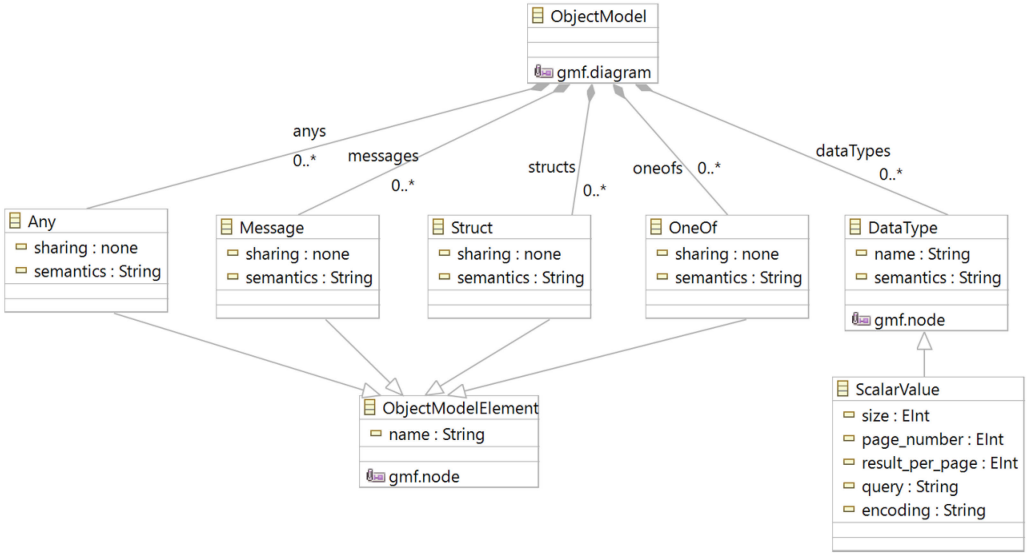


FIGURE 7. A part of the microservice data exchange metamodel.

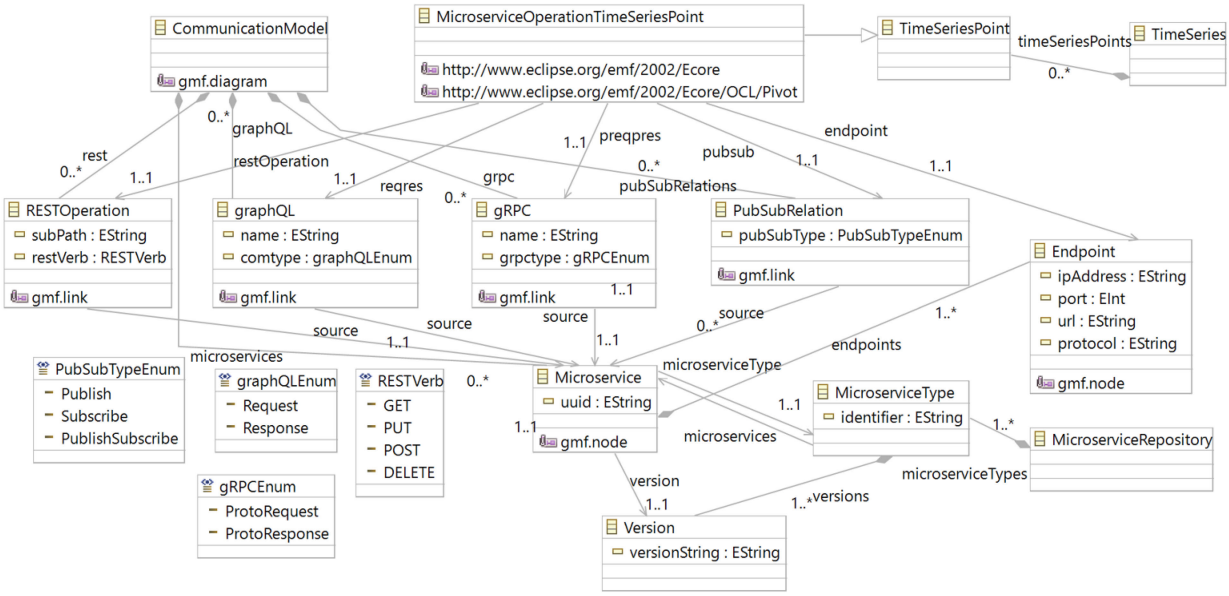


FIGURE 8. Microservice communication metamodel.

protocol in means of data type variation. When many microservices are built in different technologies and programming languages, it is very important to define the service interfaces in a standardized manner in the underlying message exchange format. *gRPC* is a powerful communication method for standardizing service communications using protocol buffers. Therefore, we extracted the gRPC data metamodel to define data exchange of microservices communication.

Publish/subscribe (pub/sub model) is another widely adopted communication method. Pub/Sub model defines an asynchronous communication protocol among loosely coupled distributed services [6]. In a pub/sub model, any published message is delivered to all services that subscribed to that topic. The pub/sub model is suitable for microservices with event-based architecture. The model can also be used to decouple applications to improve performance, reliability, and scalability [6]. For example, considering the pub/sub relationship given through the case study, the *PassengerManagement* microservice can become a member of the *Passenger* object and subscribes to the *Driver*

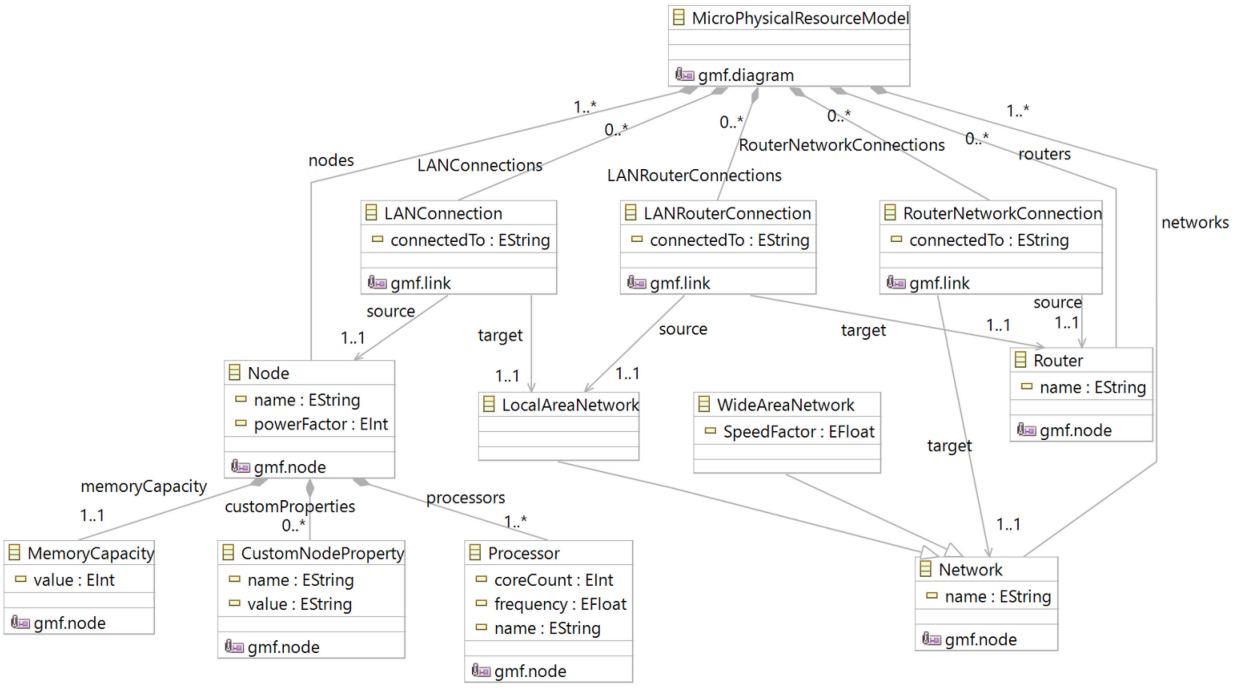


FIGURE 9. Microservice infrastructure metamodel.

object. Similarly, the *DriverWebUI* microservice publishes the *Driver* object while subscribing to the *Passenger* object.

E. DESIGN MICROSERVICE INFRASTRUCTURE MODEL

The Microservice Infrastructure Model defines the processing power and memory capacities of the available nodes and network connections among them.

As an example of physical resources designed to build the Microservice Infrastructure Model, one or more nodes can be defined to have specific parameters in which microservices can be deployed and executed. The properties of these nodes can be updated according to requirements. For instance, we can create a node that has a memory capacity of 1024 MB, 2048 MB, etc, 2.3 MHz frequencies, four cores, and two processes. All nodes can be connected to a common local network to establish a homogeneous connection network for communication between nodes. It is also possible to design a heterogeneous connection model by defining the local area and wide area networks with different communication performances and deploying the nodes to different points of the network by designing nodes with different frequency, number of cores, and memory capacities.

The design of the metamodel is shown in Figure 9. *MicroPhysicalResourceModel* is the main class representing the physical infrastructure of microservices architecture. *Node* class represents one or more physical resources with different attributes. This class has two attributes: *name* and *powerFactor*. While the *name* represents the identity information of the node, *powerFactor* defines the processing power of the node relative to other nodes. The *Processor* class defines one or more processors of the node. In this class, a processor unit can be created with a *name*, *frequency*, and *coreCount* properties. The name represents the symbolic name of the processor unit, *coreCount* allows to define the number of cores the processor unit has, and frequency shows the frequency of the processor in MHz. Each node can have a different memory capacity. Thus, similar to the *Processor* class, the value attribute in *MemoryCapacity* class defines the memory amount of the node in MegaBytes.

In the Microservice Infrastructure Model, *CustomNodeProperty* class is used for the defined attributes depending on user demand (such as disk capacity) besides the characteristics of the processor power and memory capacity assigned to each node. In this class, name-value pairs are represented by name and value attributes. For example, the disk capacity of a node can be defined by the name *"diskCapacity"* and the value *"240GB"*. The *Network* class in the metamodel is the abstract ancestor class of *LocalAreaNetwork (LAN)* and *WideAreaNetwork (WAN)* classes. One or more network definitions can be done in a physical infrastructure. Since the *LocalAreaNetwork* is faster than the *WideAreaNetwork*, the *speedFactor* attribute added to the *WideAreaNetwork* class determines how slow the network is compared to LAN. The *LANConnection* class defines the connection of a node to a local area network, while the *Router* class defines routers used to connect networks to each other.

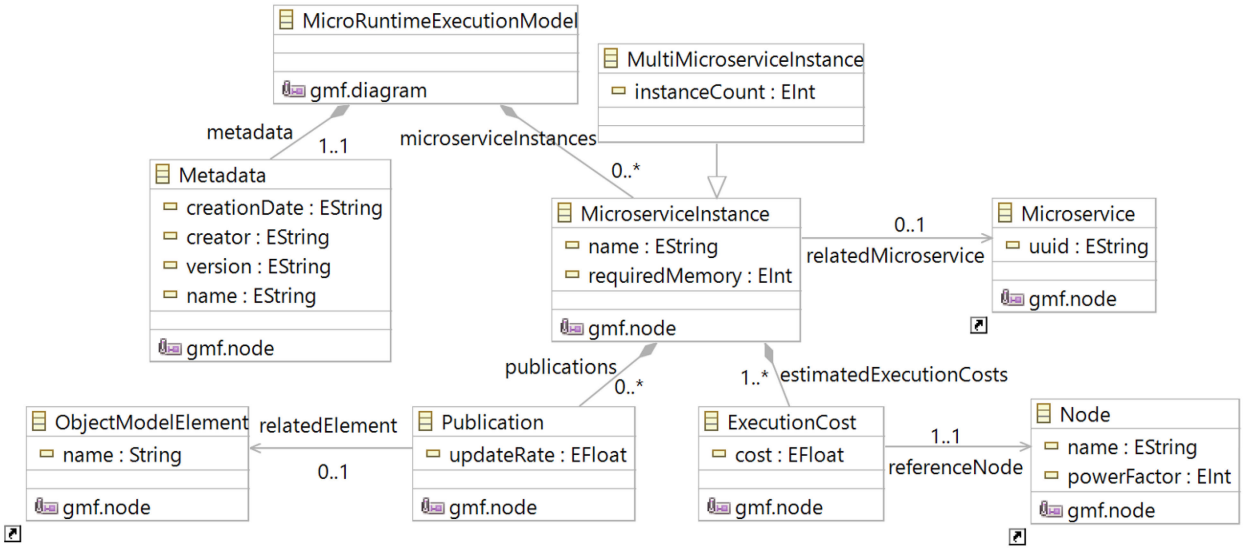


FIGURE 10. Microservice runtime execution configuration metamodel.

The *LANRouterConnection* class connects a local area network to a router, while the *RouterNetworkConnection* class connects a router to a network.

F. DESIGN MICROSERVICE RUNTIME EXECUTION CONFIGURATION MODEL

The structural features of the system are defined by the establishment of a Microservice Data Exchange Model, a Microservice Definition Model, a Microservice Communication Model, and a Microservice Infrastructure Model. In addition to these structural definitions, we also need to know how many components will be presented in the system, how often the data model elements of the components are updated (update rate), and the execution costs of each element on each target node to be able to derive deployment alternatives. In this context, defining the runtime execution configuration activity is an important part of the proposed approach and it requires all the models explained in the previous sections. For example, the user can design a microservice named *DriverManagement* to update the *Vehicle* object five times per second. Depending on the processing power, the execution cost for each *DriverManagement* instance can be defined as 5 out of 10 for one node and 8 out of 10 for another node.

The metamodel containing the elements necessary to model the runtime execution configurations are shown in Figure 10. Here, the *MicroRuntimeExecutionModel* class is the root class of the metamodel and defines an execution configuration. A runtime execution configuration consists of *Metadata* and a group of *MicroserviceInstance* instances. The *Metadata* class contains the *name*, *version*, *creator*, and *creationDate* attributes that define the configuration of execution. *MicroserviceInstance* represents an instance of a microservice defined in the Microservice Definition Model. Since the execution cost of each microservice instance on different nodes may be different, *estimatedExecutionCost* attribute has been defined in the *MicroserviceInstance* class. The *requiredMemory*, which is another attribute of the *MicroserviceInstance* class, represents the estimated amount of memory the microservice instance needs during execution. Similar to the execution cost, the *requiredMemory* attribute can also be estimated at design time. Most probably, there will be more than one instance of the same microservice in an execution scenario. For example, in a taxi hailing system there will be hundreds of taxis. The naive modeling approach is to force the designer to add a *MicroserviceInstance* to the execution model for each instance, but it is obviously not a user friendly approach. We defined a class named *MultiMicroserviceInstance* class that has an *instanceCount* attribute that defines the number of instances in the runtime execution configuration. The designer can add one *MultiMicroserviceInstance* class to the scenario with *instanceCount* set to 100 instead of adding 100 *MicroserviceInstance* classes one by one.

The *RelatedMicroservice* relationship also represents the relationship between a *Microservice* defined in the Design Microservice Definition Model and the *MicroserviceInstance*. The *MicroserviceInstance* has zero or more communication classes representing the update rate. The *updateRate* property in the communication classes represents the number of times a microservice instance is updated in one second.

1) ESTIMATING RUNTIME EXECUTION PARAMETERS OF MICROSERVICES

The presented approach can be used early in the software life cycle to determine the feasible deployment design alternatives early on. However, the approach can also be

adopted later in the life cycle to enhance the deployment architecture of the system.

This also includes the later phases such as integration and testing where the services are already developed and their memory and CPU requirements can be measured with high accuracy by actually running the services. Independent of the stages in which the presented approach is used, it is a good practice to use estimated the CPU and memory requirements of service at requirements based on the expected workload and service characteristics. To estimate resource requirements of the microservices before the service is actually developed require a comprehensive analysis including:

- The detailed use case and usage analysis based on business analysis.
- Determination of service quality requirements by analysis of business requirements.
- The specific costs and characteristics of computing, space, and networking.
- Past experiences obtained from the previous deployment patterns for similar case studies.

Resource requirements of the services should be continuously measured and monitored during the development lifecycle. Since continuously measuring and monitoring resource usage of the services manually is not tractable in means of effort and schedule, the measurement and monitoring process should be automated and integrated into CI/CD process as early as possible by using automated test frameworks.

G. GENERATING INPUT PARAMETERS FOR RESOURCE ALLOCATION ALGORITHM

The design of the application environment is completed by defining the structural characteristics, the preparation of the physical infrastructure environment, and execution configurations of the application participants as explained in the previous sections. This section describes the allocation of microservice instances into the designed nodes by taking into account constraints defined in the design such as memory requirements, execution costs of microservice instances on each node, communication costs between microservices, and processing power. Within the scope of the study, it is seen that this allocation problem matches the Capacitated Task Assignment Problem (CTAP) [29] as we mentioned before.

1) CAPACITATED TASK ASSIGNMENT PROBLEM (CTAP)

The problem of finding the feasible allocation is mentioned in the literature as a Capacitated Task Assignment Problem (CTAP) [29]. In CTAP, there are m tasks and the i^{th} task requires m_i unit memory. There are n non-equivalent processors in the environment, processor p has a total M_p unit memory and C_p unit processing capacity. The execution cost of task i on processor p is X_{ip}. Similarly, the total communication cost between tasks i and j is c_{ij}. When calculating communication costs, the frequency of inter-task communication should be taken into account in addition to the size of the transmitted data. High communication frequency between tasks i and j will result in higher communication costs c_{ij}. The goal of the optimization problem is to deploy tasks in processors with minimal communication and execution costs without exceeding memory and processing power constraints. The decision variable of the problem is: $a_{ip} = 1$, if i is assigned to processor p, otherwise it is 0. According to this definition, the CTAP can be expressed mathematically as an optimization problem with binary decision variables, as seen in the algorithm space part of Figure 11.

The adaptation of CTAP parameters extracted from design to algorithm space is shown in Figure 11. The corresponding CTAP parameters for the proposed approach are described as follows: The microservices defined in Design Microservice Definition Model correspond to task classes. Each microservice instance defined in the Design Microservice Runtime Execution Configuration Model corresponds to a task. The *instanceCount* attribute of a microservice represents the number of tasks (m) for a microservice. Each node defined in the Design Microservice Infrastructure Model corresponds to a processor. The *memoryCapacity* attribute of each node p defined in the Design Microservice Infrastructure Model corresponds to the M_p memory capacity of processor p. The *powerFactor* attribute of node p defined in the Design Microservice Infrastructure Model corresponds to the processing power of processor p. The cost attribute of *ExecutionCost* class defined in the Design Microservice Runtime Execution Configuration Model represents execution cost for node p which corresponds to the cost of executing task i on the processor p (X_{ip}). This class is linked to the *Node* class defined in the Design Microservice Infrastructure Model to identify the costs of each microservice instance on each Node p. The last parameter is c_{ij} that defines the communication cost for task i and j. In our approach, the communication cost is calculated by multiplying the size of the data exchange object defined in the Design Microservice Data Exchange Model and the *updateRate* of the publication defined in the Design Microservice Runtime Execution Configuration Model. The pseudocode of the calculation of execution costs and communication costs is detailed in Appendix B.

The problem definition for the proposed approach is formulated in Figure 12. Like CTAP definition, there are μ^c microservices and μ_i denotes the i^{th} service, which m_i unit memory and has k_i instances in total, and μ_i^j denotes j^{th} instance of i. There are P_c non-equivalent nodes (processors) in the environment, node p has a total M_p unit memory and C_p unit processing capacity. The execution cost of i on node p is x_{ip}. The total communication cost between microservices i and j is c_{ij} if they are allocated on different nodes. The goal of the optimization problem is to deploy microservices

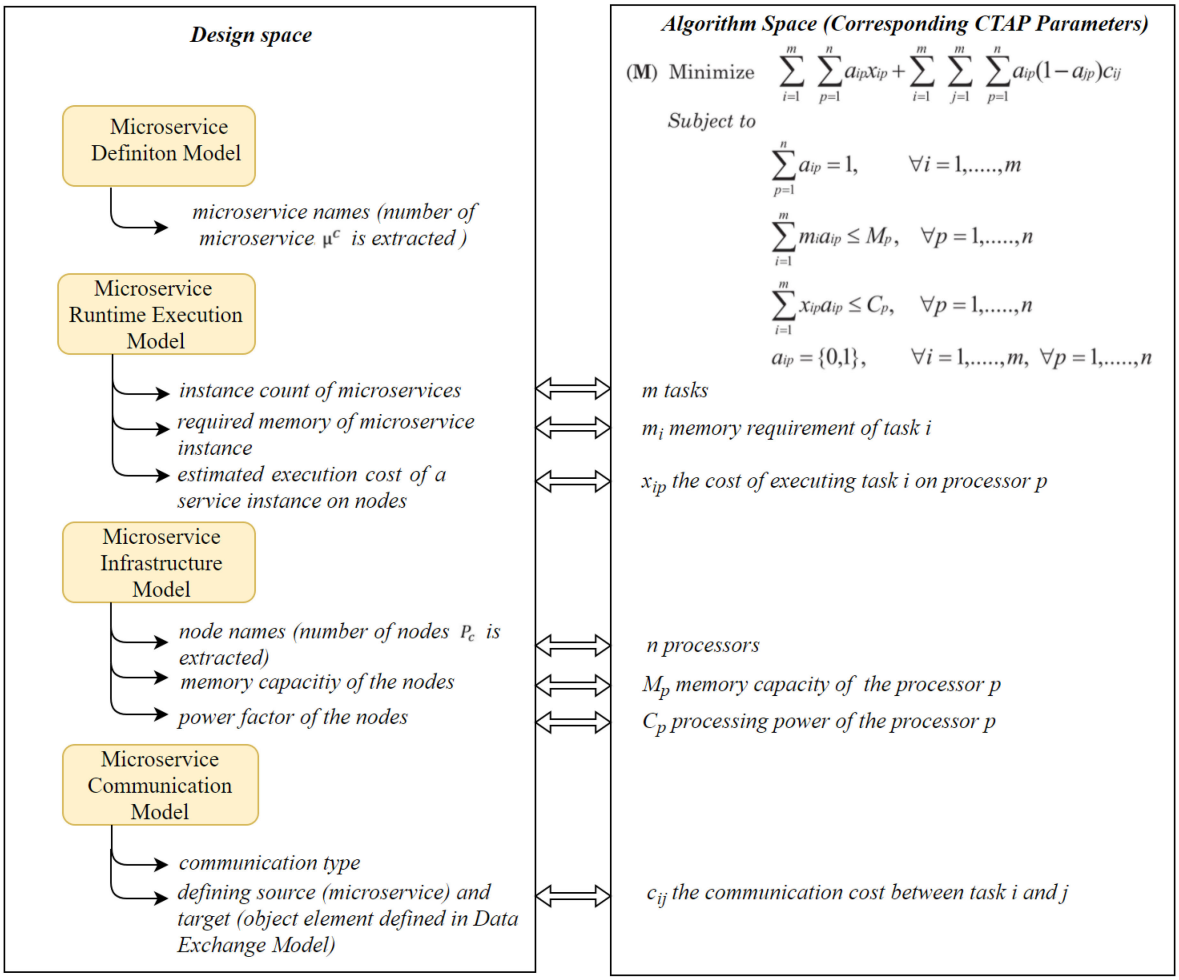


FIGURE 11. The mapping between design and algorithm spaces for the proposed approach.

in nodes with minimal communication and execution costs without exceeding memory and processing power constraints. The decision variable of the problem a_{ip}^{j} equals 1 if μ_i^j is assigned to node p, otherwise, it is 0.

The objective of the M mathematical model can be explained as follows: "Assign microservices to nodes (processors) to minimize the sum of the total execution cost and the total communication cost. If two services are assigned to the same node, the communication cost between them is assumed to be zero. When making the assignments, make sure that the total memory requirement of the services assigned to each node does not exceed the amount of node memory and the total processing power requirement does not exceed the processing power of the node. This problem is known to be NP-hard in the literature so the optimum solution cannot be found in linear time. For this reason, different optimization algorithms that generate feasible but not guaranteed to be best deployment alternatives are defined to solve this problem. With the help of these algorithms, it is aimed to deploy microservices that have high communication costs to the same nodes as much as possible. Within the scope of this study, the list of the parameters mentioned above, and the parameters used in the application environment are given in Table 2.

The most explicit difference of the proposed approach from the basic CTAP problem is the definitions of microservices and their instances. Since different numbers of instances from a microservice can be defined, it must be expressed that how many microservice instances for a microservice type are defined using k_i parameter. Accordingly, the execution and communication costs of each instance μ_i^j on the node p are calculated using the value of a_{ip}^{j} parameter.

H. GENERATE DEPLOYMENT CONFIGURATION

After the physical resource and structural model of the system designed, runtime execution configuration models defined, parameters extracted and deployment model is generated

$$(\boldsymbol{M})\ Minimize \sum_{i=1}^{\mu^c}\sum_{j=1}^{k_i}\sum_{p=1}^{P_c} a_{ip}^{j}.x_{ip} + \sum_{i=1}^{\mu^c}\sum_{j=1}^{k_i}\sum_{x=1}^{\mu^c}\sum_{y=1}^{k_x}\sum_{p=1}^{P_c} a_{ip}^{j}.\ (1 - a_{xp}^{y})c_{ix}$$

Subject to,

$$\sum_{p=1}^{P_c} a_{ip}^{j} = 1 \qquad \forall_i = 1, \ldots . \mu^c ; \ \forall_j = 1, \ldots . k_i$$

$$\sum_{j=1}^{k_i}\sum_{p=1}^{P_c} a_{ip}^{j} = k_i \qquad \forall_i = 1, \ldots . \mu^c$$

$$\sum_{i=1}^{\mu^c}\sum_{j=1}^{k_i} m_i . a_{ip}^{j} \leq M_p \qquad \forall_p = 1, \ldots . P_c$$

$$\sum_{i=1}^{\mu^c}\sum_{j=1}^{k_i} x_{ip} . a_{ip}^{j} \leq C_p \qquad \forall_p = 1, \ldots . P_c$$

$$a_{ip}^{j} = \begin{cases} 1, \ \mu_i^j \ is\ assigned\ to\ node\ p \\ 0, otherwise \end{cases}$$

FIGURE 12. The adaptation of CTAP parameters to our approach.

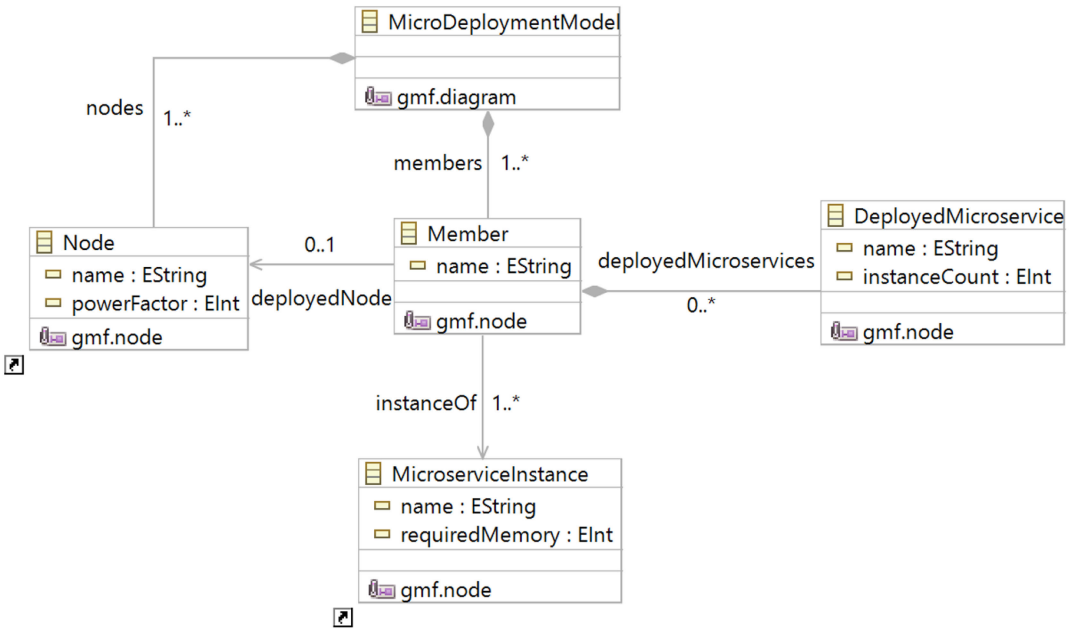


FIGURE 13. Design MicroDeployment metamodel.

with CTAP solvers, it is time to generate the deployment model according to algorithm output. The CTAP solvers generate task-processor mapping as output. In the *MicroDeployment* metamodel, there is a one or many relationship (1..*) between the *Node* (from Microservice Infrastructure Model) and the *Member* class. This represents the transfer of one or more members to one of the nodes. Moreover, one or more microservice instances defined in the *Design*

TABLE 2. Deriving CTAP adaptation parameters from the design.

CTAP Parameter	Derivation from the design
μ_i	i^{th} microservice defined in Design Microservice Definition Model
μ^c	Number of microservices defined in Design Microservice Definition Model
k_i	Number of microservice instances defined in Design Microservice Runtime Execution Configuration Model for μ_i
μ_i^j	j^{th} instance of μ_i idefined in the Design Microservice Runtime Execution Configuration Model
P_c	Number of nodes (processors) defined in Design Microservice Infrastructure Model
a_{ip}^j	μ_i^j is assigned to Node p
M_p	The memory capacity of Node p defined in Design Microservice Infrastructure Model
C_p	The power factor (the processing power) of Node p
m_i	The required memory of μ_i
x_{ip}	The estimated execution cost of μ_i on Node p
c_{ij}	c_{ij} represents the communication cost if μ_i and μ_j assigned to different nodes. Communication cost is negligible if two microservices are assigned to the same node. It is calculated by using: • Publications defined in Design Microservice Runtime Execution Configuration Model • Subscriptions defined in *Publish/Subscribe*, *gRPC*, *graphQL* or *Rest* operations of Design Microservice Communication Model • Object model elements defined in Design Microservice Data Exchange Model

Runtime Execution Configuration Model can be deployed on a *Member*. The illustrated *MicroDeployment* metamodel is shown in Figure 13.

VI. TOOL SUPPORT AND CASE STUDY

In this section, the *Micro-IDE* tool, a development environment that combines the metamodels described in the previous sections is presented. The *MicroDeployment* model is developed on the Eclipse platform and allows the realization of all the metamodels and CTAP algorithms as plugins. It also allows for adding different optimization algorithms as plugins without modifying the original tool.

The taxi-hailing application, inspired by Uber system [35], is tested on the case study described in detail in Table 1. For this purpose, all structural definitions are conducted including the Microservice Data Exchange Model, Microservices Definition Model, Microservice Communication Model, and Microservice Infrastructure Model (which is assumed to have 10 heterogeneous nodes with different memory capacities and processors). After these operations, the number of previously identified microservice instances is determined in the Microservice Runtime Execution Configuration Model. After all necessary model designs are made, the designer runs the Deployment Model Generator tool and selects the Microservice Runtime Execution Configuration Model and the Microservice Infrastructure model. Since the Microservice Runtime Execution Configuration Model refers to other structural definition models (Microservice Data Exchange, Microservice Definition, etc.), the designer is not forced to select these models again. Additionally, the designer selects the algorithm that will be used for generation of the feasible deployment alternative. In this study, the genetic algorithm proposed by Mehrabi *et al.* [27] is implemented and integrated into the tool as a sample CTAP solver. After the algorithm parameters extracted from the design and CTAP solver is executed, results of the algorithm which is a task-processor assignment table is used for generating the deployment design. The steps summarized above for the feasible deployment model generation is presented in Appendix B as a pseudocode. The main method of the pseudocode, *GENERATE_FEASIBLE_DEPLOYMENTS* met-hod uses two parameters, (1) *phy_resources* that represent the Microservice Infrastructure Model and (2) *exec_config* represents the Microservice Runtime Execution Configuration Model. *EXTRACT_PROCESSORS* method extracts the properties of the processor specified for each node in the Microservice Infrastructure model. *EXTRACT_TASKS* method extracts the execution cost of microservices on nodes and the cost of inter-service communication obtained from the Microservice Runtime Execution Configuration Model. The outputs of these two methods are taken as a parameter in the *EXECUTE_CTAP* method. According to the cost information obtained from these methods, the assignment of microservices to processors is recorded in the *assignment_tables*. Finally, more than one deployment alternatives are generated with the *CREATE_DEPLOYMENT_MODELS* method using the parameters in the *assignment_tables*.

TABLE 3. Time to generate values according to microservice instances and number of nodes using CTAP algorithm.

Total number of microservice instances	Number of nodes	Time to generate (seconds)
5	3	0
50	4	3
128	4	15
300	5	104
550	10	315
960	10	1433

Figure 14 shows a sample deployment model generated using this algorithm after applying the sample scenario in Table 1. After running the algorithm, 550 microservice instances are automatically deployed to ten nodes. The number of microservice instances is indicated in parentheses near the name of microservices. Services not specified in parentheses are assigned as one.

Table 3 shows the deployment model generation times obtained by applying the *Micro-IDE* tool on the nodes in a different number of microservice instances. Experiments are carried out on a computer that has Intel Core i7-6700HQ CPU, 2.60 GHz, and 16 GB RAM. In this experiment, the estimated execution cost of all microservice instances,

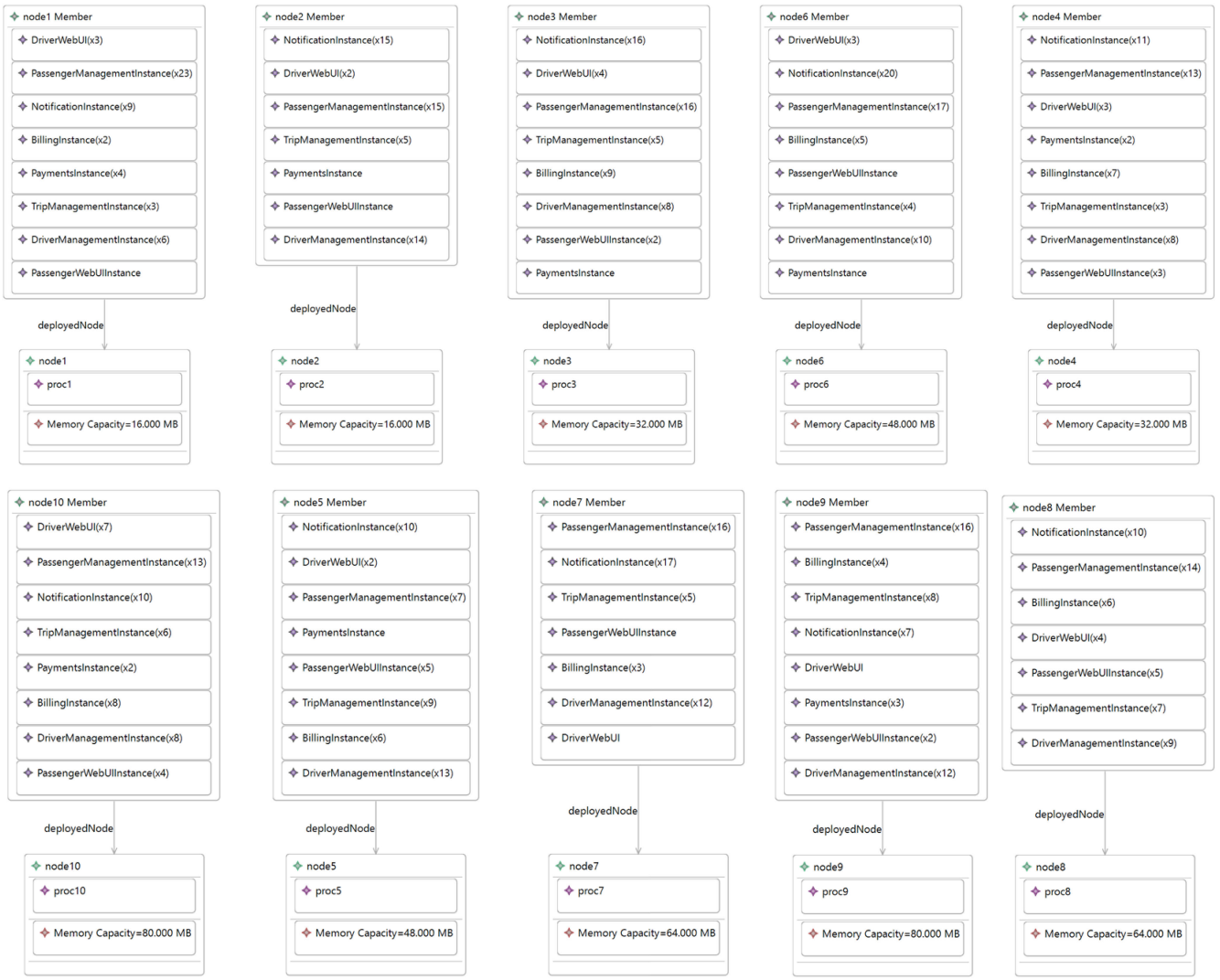


FIGURE 14. Generated feasible deployment alternative including 550 microservice instances using the genetic algorithm.

the memory and communication costs required for the operation are kept constant and the time taken for the deployment model generation is evaluated. As a result of the experiments performed on different numbers of nodes and service instances, it is seen that the generation time increases as the number of service instances increases. However, the time required for the generation process is often much lower than expert judgment. Therefore, we can say that the proposed approach generates a deployment model in a reasonable time. When the generated deployment model is examined (see Figure 14), the algorithm gives the priority to deploying the service instances that are in frequent communication with each other on the same node. For example, in the performed scenario, two microservice instances that have the highest communication costs are *DriverManagementInstance* and *PassengerManagementInstance*. Therefore, the allocation of these service instances on the same nodes decreases the communication cost to negligible values when compared with communication over the network. It is also observed that the memory capacity of the instances assigned to the nodes does not exceed the memory capacity of the nodes. When the memory capacity of the node is exceeded, service instances are deployed to other nodes. Furthermore, the total execution cost of the microservices on the nodes is also aimed to be minimized algorithmically.

In addition to analyzing time to generate a deployment model, a detailed benchmark report is provided to the developer. The benchmark report includes various evaluations such as the total communication and execution costs, the memory utilization rates of each node, and the communication cost between each service pair. Hereby, it enables the developer to choose a feasible deployment alternative over the generated deployment alternatives or to update the design to improve resource usage.

When analyzing the feasibility of the deployment models two approaches can be used. The first approach is to intuitively examine deployment alternatives by an expert who knows the architecture well. In this approach, the feasible deployment alternative for the architecture is evaluated manually by the expert. Most intuitive manual evaluation strategy is that the expert checks if the frequently communicating microservices are deployed to the same resource. From the overall performance improvement perspective, this judgment strategy is not so healthy since some microservices need to be deployed to different resources even if they are frequently communicating. Hence we need a more holistic deployment model analysis approach, which can be achieved with tool support. The second approach is to generate deployment models automatically using algorithmic approaches and compare deployment alternatives automatically in terms of communication and execution costs and total memory capacities of the assigned services in the nodes. Our proposed approach and tool support provide the needed holistic model for the second approach.

The communication and execution costs of each microservice instance for the deployment model generated with the genetic algorithm and manual deployment by the expert are listed in Table 4. The communication cost refers to the total communication cost between microservices defined in the architecture. The size of the data exchange object and the frequency of communication between services are taken into account when deriving the total communication cost. The execution cost refers to the total execution cost of microservices running on the nodes. The last column for each cost shows the improvement rate for the results of the genetic algorithm according to expert deployment.

When the deployment model generated by the genetic algorithm and the expert deployment are compared, it is seen that the expert deployment shows relatively lower performance in terms of communication and execution costs, as seen at the last row of the Table 4. When the communication costs are examined, the costs of the *PaymentInstance*, *NotificationInstance*, and *TripManagementInstance* services are stated as zero. Because a single communication cost is calculated between two services that exchange data with each other, this value is only shown in the service of the publisher. For example, in the specified scenario for the case study, *TripManagementInstance* service subscribes to *ManageTrip* message (the object where the start and end of the trip location are stored), while *BillingInstance* publishes this message. Therefore, the cost of this data exchange is only charged to the publisher (*BillingInstance*).

While some communication and execution costs of the microservice instances using expert deployment seems to be better than the genetic algorithm, the aim of the proposed approach is to achieve minimum total execution and communication cost of the system algorithmically. Besides the 2.06% improvement of the total execution cost, the genetic algorithm improves the total communication cost by 1.51%.

The result of the improvement rates varies according to the developed architecture, the case study, and the selected CTAP solver algorithm. The aim is to obtain close or better results to expert judgment. Furthermore, as the service instances increase, it becomes difficult for the expert to find feasible deployments. For this reason, the algorithmic approach is advantageous compared to expert judgment as it offers multiple deployment alternatives to the developer in a shorter time. In this experiment, we used a genetic algorithm to compare with human expert performance. Please note that our approach and tool are both algorithms agnostic and different CTAP solvers can be integrated into the system hassle-free. A key future work for our study is to implement and integrate other CTAP solver algorithms, analyze and compare their performance for different cases in means of performance optimization and time to generate a deployment result.

VII. RELATED WORK

Microservice architectures contribute to the Continuous Integration and Continuous Deployment by enabling isolated update of application services, providing a modular application design and enabling use of diverse technology stacks for service development. One of the main technical problems for microservices is the allocation of microservices in limited resources as close to optimum as possible. In addition to this requirement, the deployment of services should be fast, reliable, and cost-effective. The rapid, reliable, cost-effective deployment of each service to the available resources is reflected in the literature as one of the challenges of microservice architectures.

When the studies in the literature are examined, it is seen that a deployment approach considering the communication parameters of microservices, memory capacities, execution costs on nodes is not available. In [10], conducted by an author involved in our study, a deployment approach has been proposed for parallel and distributed systems, and tool support [9] to examine this approach has been developed. This approach has also been implemented for Data Distribution Service-based systems [34].

When the other tools proposed in this field are searched in the literature, it is seen that special components and infrastructures are used rather than algorithmic approaches. For instance, Gabbrielli *et al.* [15] report that microservices have a suitable architectural structure for the development of distributed systems in the cloud, but due to the dynamic nature of these services, appropriate methods are required for their automatic deployment. In their study, they proposed a tool called JRO written in Jolie for the automatic and optimized deployment of microservices. JRO tool offers an API to access all data related to services and platforms running on the managed system, deploy, start, terminate and remove services, as well as monitor resource consumption and performance. While this tool reconfigures the service deployment of already developed systems on runtime, we focus

TABLE 4. The communication and execution costs values for the algorithmic and manual approach.

Microservice instance	The execution cost (unit)			The communication cost (Mbytes/second)		
	Expert deployment	Genetic algorithm	Impr (%)	Expert deployment	Genetic algorithm	Impr (%)
DriverManagementInstance (x100)	2325	2306	0.81	2.265	2.234	1.36
DriverWebUIInstance (x30)	695	660	5.03	0.293	0.292	0.34
PaymentInstance (x15)	350	319	8.85	0	0	0
BillingInstance (x50)	1160	1172	-1.03	0.00645	0.00647	-0.3
PassengerManagementInstance (x150)	3485	3455	0.86	1.4019	1.4256	-1.69
NotificationInstance (x125)	2905	2851	1.85	0	0	0
TripManagementInstance (x55)	1275	1270	0.39	0	0	0
PassengerWebUIInstance (x25)	705	600	13.4	0.3604	0.3031	15.89
Total costs	**12.900**	**12.633**	**2.069**	**4.3281**	**4.2626**	**1.51**

on optimizing the resources and deployment of microservices during the early design phase when the system is not developed yet and the cost of change is lower. In another study, Wan *et al.* [36] formulated the application deployment problem by examining Docker's features, microservice-based application requirements, and available resources in cloud environments. Then, they proposed a framework and developed an algorithm for container placement and task assignment. In the developed framework, application requests are processed as microservices on Execution Containers (ECs). Resource allocation and resource management for applications are performed on Microservice Controllers (MCs). The authors design a communication-efficient framework while performing an efficient usage of resources. However, this framework focuses on container placement rather than deployment of microservices and it cannot inform the developer while designing the communication patterns of microservices or the memory requirements of the resources and services.

Ciuffoletti [11] provides the automatic deployment of microservices with a machine implementing the monitoring infrastructure. The application consists of two multi-threaded Java applications that implement two components. The application is implemented on the Docker hub. The contribution of the study is defined as the identification of all steps from the model defining the infrastructure to the machine deploying the probes in the design of a demand-based monitoring service. In the ''monitoring as a service'' methodology proposed in this study, communication costs between microservices during deployment are not taken into account while only the metrics of the resources given by a container is measured. In another study dealing with the deployment problem of microservices, Zheng *et al.* [37] propose a new platform called BigVM to separate SaaS developers from deployments and close the gap between best practices and real-world applications. BigVM provides SaaS developers with microservice-focused deployment kits to enable the creation, customization, and deployment of SaaS solutions on a multi-tier microservice-based form. Similarly, this platform focuses only on the optimized use of resources in terms of CPU workload and file I/O operations.

Some studies in the literature address the deployment challenges in transitioning from monolithic architectures to microservices and compare these architectures in terms of deployment performance. Berger *et al.* [8] successfully implemented the Docker-based containerized software paradigm for both software development and deployment of the software by converting their monolith architectures into microservices. Considering the experiences in the development and deployment process of containerized software, it is seen that the development of containerized software in the automotive industry is much more successful than the monolithic configuration due to its deployability, automation, and traceability features. In this study, existing technologies are used as a deployment strategy rather than proposing an algorithmic approach and it is stated only that Docker records have an important role in accelerating the deployment procedure in the web environment. Similarly, Singh and Peddoju [32] deploy their proposed architecture on Docker containers and test it using a social networking application. The authors analyzed the difficulties that occur during the continuous integration and deployment of microservices and proposed an automated system that helps the deployment and continuous integration of microservices to cope with these challenges. The experiments are compared with the monolithic approach in terms of parameters such as response time, throughput, and deployment time. As a result of the experiments, the deployment with the proposed design reduces the time and effort for the deployment and continuous integration of the microservices. At the same time, due to the low response time and high bit flow, it has been found that the microservice-based application outperforms the monolithic design. In this study, rather than the optimal deployment of microservices to servers, a deployment approach has been developed to overcome the challenges of service discovery, continuous integration, and continuous delivery.

In another study, the process of migrating the core business software of MGDIS SA software company from a monolith to a web-based microservice application is discussed. Gouigoux and Tamzalit [17] have emphasized three important questions for a successful transition to the Web-oriented

architecture during this transformation: (1) How to split a monolith architecture into the most appropriate granular microservices communicating via APIs, (2) How to determine the best deployment and (3) most efficient orchestration of microservices. In the study, it is observed that the Quality Assurance (QA) effort decreases as the granular structure increases in the transition from monolith to microservices, and the deployment cost increases linearly in the absence of any automation, and asymptotically in the case of automation. In this study, no optimization has been performed to achieve the minimum deployment cost in a microservices architecture.

Guo *et al.* [20] proposed a new Cloudware PaaS platform called CloudwareHub based on microservice architecture and lightweight container technology. CloudwareHub is a platform that provides users and developers with tools for developing, testing, deploying, and executing software in the cloud. Authors are able to deploy traditional software that provides services to users via a browser directly on this platform without any changes. Additionally, this platform supports scalability, automatic deployment, disaster recovery, and elastic configuration thanks to microservice architectures. Since this platform performs automatic deployment over existing technologies such as Docker, Docker Swarm, etc., and does not propose a deployment generation approach taking into account the deployment cost, it differs from our approach.

Leitner *et al.* [24] introduced a graph-based approach to model deployment costs, including computational I/O costs during the deployment of microservice-based applications to the public cloud. The model called CostHat supports services that use new cloud programming paradigms such as microservices deployed to traditional IaaS and PaaS clouds, and AWS Lambda [4]. Based on a network model, CostHat includes tools to alert costly code changes in the Integrated Development Environment (IDE). The CostHat platform does not present automatic deployment alternatives, it only offers to developers the opportunity to evaluate the deployment cost during development of the microservice-based applications.

VIII. CONCLUSION

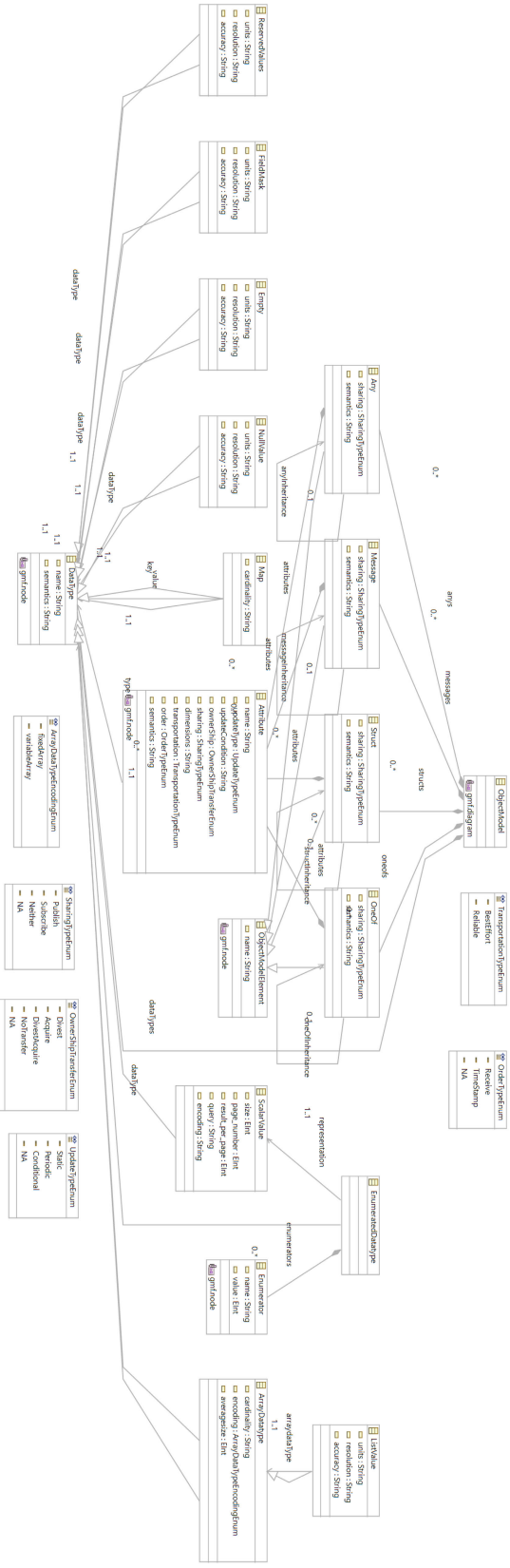
Independent deployment of microservices to cloud resources enables efficient use of resources by scaling high-demand services. At this point, the feasible deployment of microservices on cloud servers becomes an important problem. In the current microservice deployment approaches, the generation of the deployment model is performed either by expert judgment or postponed to the deployment phase. Although expert judgment can be sufficient up to a certain problem size, the expert needs to know the system very well and the manual deployment process becomes time-consuming and not tractable in large systems. Furthermore, postponing the deployment model design and evaluation negatively affects the development process and requires returns to the design phase of the project lifecycle to find feasible deployment alternatives. For these reasons, we proposed an algorithmic approach to enable automated feasible deployment of microservices to limited capacitated resources at early design phase. To find feasible deployment alternatives, we developed a method for extracting parameters from the design and converting the problem to the well-known and extensively studied CTAP problem, and generating deployment models according to outputs of the CTAP solvers. One of the most important benefits of this approach is the early analysis of the system design to determine feasible deployment alternatives and update the system architecture in the design phase. Changing the deployment model in the development or later phases instead of the early design phase is exponentially more expensive since this will lead to rework in the product lifecycle elements such as architectural design, detailed design, implementation, testing elements, and documentation. These reworks will increase the project cost and may even lead to schedule delays.

In order to validate and evaluate the proposed approach, a tool environment based on the Eclipse framework is developed. The model allows defining the resources, microservices to work on them, interactions among the microservices, and runtime scenarios to define the expected service instances and dynamic traffic among them. To illustrate the approach, a taxi-hailing system inspired by Uber is selected as a case study. To implement the case study, the following models are designed: the Microservice Definition Model that defines the microservices, the Microservice Data Exchange Model that defines the data structures used by the services, the Microservice Communication Model that identifies the communications among the microservices, the Microservice Infrastructure Model that defines the servers, and the Microservice Runtime Execution Configuration Model that defines microservice instances and traffic load among them. Based on these models, the parameters required for the CTAP algorithm are extracted from the design models to generate feasible deployment alternatives. Experimental results show that generation times of the deployment alternatives are reasonable for evaluation at the design phase. The tool also supports comparison and benchmarking of the generated deployment models.

The goal of this study was to present a systematic approach for deriving the automated deployment configuration alternatives using algorithmic solutions. Yet, an interesting study would be to investigate the use and performance of the implementation of other optimization approaches for CTAP, such as evolutionary computation, linear programming, and expectation-maximization algorithms. Additionally, dynamic parameters of runtime execution environment can also be investigated to create further approaches, which consider load balancing of nodes, changing user demand, and network traffic.

APPENDIX A

The ecore diagram of the microservice data exchange model.



APPENDIX B

Algorithm 1 Algorithm for generating feasible microservice deployment model

```
GENERATE_FEASIBLE_DEPLOYMENT (phy_resources, exec_config)
    processors← EXTRACT_PROCESSORS (phy_resources)
    tasks← EXTRACT_TASKS (exec_config)
    allocation_table←EXECUTE_CTAP (tasks, processors)
    CREATE_DEPLOYMENT_MODEL (allocation_table)

EXTRACT_PROCESSORS (resources)
    create empty list processors
    for each node in resources do
    processors← CREATE_PROCESSOR (node)
    append processor to processors
    end for
    return processors

EXTRACT_TASKS (exec_config)
    create empty list tasks
    for each source_module_inst in exec_config do
    tasks← CREATE_TASK (source_module_inst)
    for each target_module_inst in exec_config do
    comm_cost← GET_COMM_COST(source_module_inst,target_module_inst)
    SET_COMM_COST(task, comm_cost,target_module_inst.ID)
    end for
    append task to tasks
    end for
    return tasks

CREATE_DEPLOYMENT_MODEL (allocation_table)
    for each task in allocation_table do
    deployed_module←CREATE_DEPLOYED_MODULE(task)
    deployed_module.processor←GET_PROCESSOR(allocation_table,task)
    end for

GET_COMM_COST (source_module_inst, target_module_inst)
    communication_cost=0
    if source_module_inst not equals target_module_inst then
    for each publication of source_module_inst do
    data_exchange_object←publication.data_exchange_object
    if IS_SUBSCRIBER (target_module_inst, data_exchange_object)
    then
        communication_cost += CALCULATE_COST (publication)
    end if
    end for
```

Algorithm 1 Algorithm for generating feasible microservice deployment model (continued)

```
        end if
        return communication_cost

IS_SUBSCRIBER (target_module_inst, data_exchange_object)
        module←target_module_inst.related_module
        if module subscribes to data_exchange_object or a parent of it
        then return TRUE
        end if
        else return FALSE
        end else

CALCULATE_COST (publication)
        data_exchange_object←publication.data_exchange_object
        update_rate←publication.update_rate
        return update_rate x SIZEOF (data_exchange_object)

SIZEOF (data_exchange_object)
        size=0
        if data_exchange_object has a parent_object then
            size += SIZEOF (parent_object)
        end if
        if data_exchange_object is an Object Class then
            for each attribute of data_exchange_object do
            size += SIZEOF (attribute.datatype)
            end for
        end if
        return size

SIZEOF_DATATYPE (parameter.datatype)
        size=0
        if parameter.datatype is a Message then
            size = ((Message) parameter.datatype).size
        end if
        else then
// calculates the size of enumerated, array, fixed, list and
// variant data types recursively until reaching message type elements
        end else
        return size
```

REFERENCES

[1] (2019). *Apache Mesos*. Accessed: Jul. 31, 2019. [Online]. Available: http://mesos.apache.org/

[2] (2019). *AWS*. Accessed: Sep. 9, 2019. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[3] (2019). *AWS Fargate*. Accessed: Sep. 9, 2019. [Online]. Available: https://aws.amazon.com/tr/fargate/faqs/

[4] (2019). *AWS Lambda*. Accessed: Sep. 13, 2019. [Online]. Available: https://aws.amazon.com/tr/lambda/

[5] (2019). *AWS Lambda*. Accessed: Sep. 13, 2019. [Online]. Available: https://docs.aws.amazon.com/en_us/whitepapers/latest/microservices-on-aws/introduction.html

[6] (2019). *AWS*. Accessed: Sep. 12, 2019. [Online]. Available: https://aws.amazon.com/tr/pub-sub-messaging/

[7] (2019). *AWS*. Accessed: Sep. 9, 2019. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

[8] C. Berger, B. Nguyen, and O. Benderius, "Containerized development and microservices for self-driving vehicles: Experiences & best practices," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, Apr. 2017, pp. 7–12.

[9] T. Celik and B. Tekinerdogan, "S-IDE: A tool framework for optimizing deployment architecture of high level architecture based simulation systems," *J. Syst. Softw.*, vol. 86, pp. 2520–2541, Oct. 2013.

[10] T. Çelik, B. Tekinerdogan, and K. M. Imre, "Deriving feasible deployment alternatives for parallel and distributed simulation systems," *ACM Trans. Model. Comput. Simul.*, vol. 23, no. 3, pp. 1–24, Jul. 2013.

[11] A. Ciuffoletti, "Automated deployment of a microservice-based monitoring infrastructure," *Procedia Comput. Sci.*, vol. 68, pp. 163–172, Jan. 2015.

[12] (2019). *Consul*. Accessed: Sep. 13, 2019. [Online]. Available: https://www.consul.io

[13] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, *Microservices From Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. New York, NY, USA: IBM Redbooks, 2016. [Online]. Available: https://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf

[14] (2019). *Docker Swarm*. Accessed: Jul. 31, 2019. [Online]. Available: https://docs.docker.com/engine/swarm/

[15] M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi, "Self-reconfiguring microservices," in *Theory and Practice of Formal Methods*. Cham, Switzerland: Springer, 2016, pp. 194–210.

[16] (2019). *Google Kubernetes Engine*. Accessed: Sep. 9, 2019. [Online]. Available: https://cloud.google.com/kubernetes-engine/

[17] J.-P. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a Web oriented architecture," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, Gothenburg, Sweden, Apr. 2017, pp. 62–65.

[18] (2019). *GraphQL*. Accessed: Sep. 12, 2019. [Online]. Available: https://graphql.org/

[19] (2019). *gRPC*. Accessed: Sep. 12, 2019. [Online]. Available: https://grpc.io/

[20] D. Guo, W. Wang, G. Zeng, and Z. Wei, "Microservices architecture based cloudware deployment platform for service computing," in *Proc. IEEE Symp. Service-Oriented Syst. Eng. (SOSE)*, Oxford, U.K., Mar. 2016, pp. 358–363.

[21] (2019). *Haproxy*. Accessed: Sep. 9, 2019. [Online]. Available: http://www.haproxy.org

[22] (2019). *Jenkins* Accessed: Sep. 12, 2019. [Online]. Available: https://www.jenkins.io

[23] (2019). *Kubernetes*. Accessed: Jul. 31,2019. [Online]. Available: https://kubernetes.io/

[24] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proc. 9th Int. Conf. Utility Cloud Comput.*, Shanghai, China, Dec. 2016, pp. 165–174.

[25] (2019). *Marathon*. Accessed: Oct. 27, 2019. [Online]. Available: https://mesosphere.github.io/marathon/

[26] (2019). *T. Mauro*. Accessed: Oct. 30, 2019. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices

[27] A. Mehrabi, S. Mehrabi, and A. D. Mehrabi, "An adaptive genetic algorithm for multiprocessor task assignment problem with limited memory," in *Proc. World Congr. Eng. Comput. Sci.*, San Francisco, CA, USA, 2009, p. 115.

[28] M. Neppelenbroek, M. Lossek, R. Janssen, and T. de Boer. (2011). *Twitter an Architectural Review*. Accessed: Nov. 4, 2019. [Online]. Available: http://www.timdeboer.eu/paper_publishing/Twitter_An_Architectural_Review.pdf

[29] T. Pirim, "A hybrid metaheuristic algorithm for solving capacitated task allocation problems as modified XQX problems," Ph.D. dissertation, Univ. Mississippi, Oxford, MS, USA, 2006.

[30] (2015). *E. Reinhold*. Accessed: Sep. 8, 2019. [Online]. Available: https://mesosphere.github.io/marathon/

[31] (2016). C. Richordson. *Microservices From Design to Deployment, in: NGINX*. Accessed: Sep. 8, 2019. [Online]. Available: https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/

[32] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *Proc. Int. Conf. Comput., Commun. Autom. (ICCCA)*, Greater Noida, India, May 2017, pp. 847–852.

[33] SmartBear. (2015). *Why You Can't Talk About Microservices Without Mentioning Netflix*. Accessed: Sep. 8, 2019. [Online]. Available: https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/

[34] B. Tekinerdogan, T. Çelik, and Ö. Köksal, "Generation of feasible deployment configuration alternatives for data distribution service based systems," *Comput. Standards Interfaces*, vol. 58, pp. 126–145, May 2018.

[35] (2015). *Uber*. Accessed: Jul. 31, 2019. [Online]. Available: https://www.uber.com

[36] X. Wan, X. Guan, T. Wang, G. Bai, and B. Y. Choi, "Application deployment using microservice and Docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.

[37] T. Zheng, Y. Zhang, X. Zheng, M. Fu, and X. Liu, "BigVM: A multi-layer-microservice-based platform for deploying SaaS," in *Proc. 5th Int. Conf. Adv. Cloud Big Data (CBD)*, Aug. 2017, pp. 45–50.

[38] A. Deb. (2016). *Application Delivery Service Challenges in Microservices-Based Applications*. Accessed: Dec. 10, 2020. [Online]. Available: http://www.thefabricnet.com/application-delivery-servicechallenges-in-microservices-based-applications/

[39] E. Wilde and C. Pautasso. (2011). *REST API Tutorial*. Accessed: Dec. 15, 2020. [Online]. Available: https://restfulapi.net/

[40] (2020). *Omg, Omg-DDS*. Accessed Dec. 15, 2020. [Online]. Available: https://www.omg.org/omg-dds-portal/omgwiki

[41] Amazon. (2019). *Amazon Simple Notification Service*. Accessed: Dec. 15, 2019. [Online]. Available: https://aws.amazon.com/sns/

[42] (2019). *Apache, Kafka*. Accessed: Dec. 15, 2019. [Online]. Available: https://kafka.apache.org

[43] Amazon. (2019). *Amazon Simple Queue Service*. Accessed: Dec. 15, 2019. [Online]. Available: https://aws.amazon.com/tr/sqs/

[44] (2019). *RabbitMQ*. Accessed: Dec. 15, 2020. [Online]. Available: https://www.rabbitmq.com/

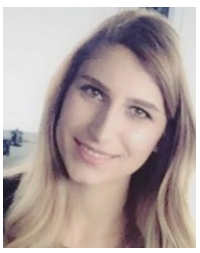
[45] (2020). *Apache, Activemq*. Accessed: Dec. 15, 2020. [Online]. Available: http://activemq.apache.org/

[46] (2020). *Java, Oracle*. Accessed: Dec. 15, 2019. [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html

[47] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2018, pp. 198–206.

[48] (2020). *Ngnix*. Accessed: Dec. 20, 2020. [Online]. Available: https://www.nginx.com/blog/building-microservices-inter-process-communication//

[49] (2020). *Language Guide (Proto3)*. Accessed: Dec. 20, 2020. [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto3



ISIL KARABEY AKSAKALLI graduated from Gazi University, in 2013. She received the M.Sc. degree from Atatürk University, in 2015. She is currently pursuing the Ph.D. degree with Hacettepe University. She started to work as a Research Assistant with Atatürk University, in 2014. She was appointed as a Research Assistant with Erzurum Technical University. Her research interests include microservice architecture, optimization methods, distributed systems, machine learning, and deep learning techniques.

TURGAY CELIK received the B.S., M.Sc., and Ph.D. degrees in computer engineering from Hacettepe University, Turkey, in 2003, 2005, and 2013, respectively. From 2003 to 2005, he served as a Research Assistant with Hacettepe University. Since 2005, he has been a Lead Software Engineer with MilSOFT Inc., Turkey. He has ten years of professional experience in software engineering research and software development. His research interests include distributed systems, infrastructure and middleware technologies, modeling and simulation, software architecture modeling, model-driven software development, software design optimization, and software performance profiling and optimization.

AHMET BURAK CAN (Member, IEEE) received the B.S. and M.S. degrees in computer science and engineering from Hacettepe University, Turkey, and the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, USA. He is currently affiliated with the Department of Computer Engineering, Hacettepe University. His main research interests include computer vision, distributed systems, and network security.

BEDIR TEKINERDOGAN received the M.Sc. and Ph.D. degrees in computer science from the University of Twente, Netherlands, in 1994 and 2000, respectively. He is currently a Full Professor and the Chair of the Information Technology Group with Wageningen University, The Netherlands. He has more than 25 years of experience in software/systems engineering. He has been active in dozens of national and international research and consultancy projects with various large software companies, whereby he has worked as a Principal Researcher and leading software/system architect. He is the author of more than 300 peer-reviewed scientific articles.