Received: 27 August 2021 | Revised: 17 January 2022 | Accepted: 21 March 2022

DOI: 10.1002/spe.3088

RESEARCH ARTICLE

WILEY

Micro-IDE: A tool platform for generating efficient deployment alternatives based on microservices

Işıl Karabey Aksakallı[1] | **Turgay Çelik**[2] | **Ahmet Burak Can**[3] | **Bedir Tekinerdoğan**[4]

[1]Department of Computer Engineering, Erzurum Technical University, Erzurum, Turkey

[2]BITES Defence & Aerospace, Ankara, Turkey

[3]Department of Computer Engineering, Hacettepe University, Ankara, Turkey

[4]Information Technology Group, Wageningen University and Research, Wageningen, The Netherlands

Correspondence
Işıl Karabey Aksakalli, Department of Computer Engineering, Erzurum Technical University, Erzurum, Turkey.
Email: isil.karabey@erzurum.edu.tr

Funding information
None.

Abstract

Microservice architecture (MSA) is a paradigm to design and develop scalable distributed applications using loosely coupled, highly cohesive components that can be deployed independently. The applications that realize the MSA may contain thousands of services that together form the overall system. Microservices interact with each other by producing and consuming data. Deploying frequently communicating services to the same physical resource would reduce network utilization, which is vital for reducing costs and improving scalability. Since the physical resources have limited capacity, it is not always possible to deploy communicating services to the same resource. Therefore, automated efficient deployment alternatives need to be generated for MSA in the design phase. To address this problem, we proposed an algorithmic approach to generate efficient microservice deployment configurations to available cloud resources in our previous study. In this study, a tool (Micro-IDE) has been proposed to realize and evaluate this approach. The Micro-IDE tool has been validated using a case study inspired by the Spotify application.

KEYWORDS

automated deployment of microservices, cloud computing, microservice architectures, optimization algorithms, tool platform for deploying microservices

1 | INTRODUCTION

In recent years, many companies are rapidly migrating their applications to microservice architectures (MSAs), especially for migration to cloud-native applications. Cloud providers present resource flexibility and nearly infinite virtual resources in various aspects such as CPU, memory, disk, and network bandwidth based on the pay-as-you-go principle. Although the resources are assumed to be unlimited, they are not free of charge. Therefore, businesses try to minimize the utilized resources for reducing operating costs without jeopardizing the application performance. In this context, the microservice concept helps to efficiently use resources by enabling deployment flexibility to application developers.[1]

The main focus of MSA is to decompose the system into highly cohesive, loosely coupled, scalable units.[2] This decoupling of small modules with well-defined interfaces has several benefits. First of all, it allows microservice-based applications to deploy new service versions without stopping the whole application.[2] The use of microservices also enables the utilization of different technology stacks and programming languages to develop system components. Reducing the

Abbreviations: BDA, big data analytics; CTAP, capacitated task assignment problem; ESB, enterprise service bus; MSA, microservice architecture; MDE, model-driven engineering; SOA, service-oriented architecture; TAP, task assignment problem.

service coupling enables scaling the development process since it reduces the coordination requirement among teams which is critical for setting up geographically distributed teams in different time zones.

MSA has many benefits, but there are several challenges to tackle, including the management complexity of the services. Although microservices should be designed as isolated as possible, the services need to interact with each other during the system execution. Managing communication among thousands of microservices with optimal network costs is a difficult task. Managing the microservices' infrastructure is critical since services must be deployed to limited capacity resources to enable load balancing by considering the available CPU power, memory, and bandwidth. There are available tools and utilities for microservice lifecycle management, including resource usage and on-demand provisioning. However, these management tools are insufficient for providing information at the early design phase, such as the amount of data exchange among microservices.[1] The total communication cost between two microservices is a function of the type and the number and size of the messages exchanged over time.[1] These relationships among the services are critical for the overall performance of the application.

Some service orchestration platforms like Kubernetes,[3] Docker Swarm,[4] and Apache Mesos[5] allow developers to configure CPU and memory thresholds for each container that orchestration platforms use for managing the deployment of services. The missing part for this configuration is the communication cost among each service that would affect overall system performance seriously. For a large-scale and complex microservice application, the manual selection of the deployment configuration becomes less tractable, and it is very hard to create an efficient alternative. An improper and thus inefficient deployment configuration can cause a cascaded increase in communication load on other services that the service interacts with due to the dynamically increasing load on a service. This situation is one of the biggest challenges that limit the scalability of microservice-based applications.

In summary, deployment of microservices according to manually prepared configuration files may cause performance and scalability issues since it is hard to manually define a deployment configuration that minimizes the communication cost among hundreds of microservices without exceeding the resource limits of each node. In our previous study,[6] we have proposed a systematic approach for generating efficient deployment alternatives, and we focused on the model-driven architecture. We modeled and defined the design space with given deployment parameters and automatically derived the feasible deployment solution. The approach was validated using a smaller case study (Uber) compared to the Spotify case study. This study focuses on the challenges of and the development of the tool framework that supports the systematic approach detailed in our previous study. The current study is complementary to the earlier paper but the focus and the contributions are different. We have described the usage of Micro-IDE tool architecture by implementing a Spotify case study. Furthermore, we applied eight types of algorithmic approaches to the case study in addition to the manual deployment approach. Subsequently, we have evaluated the deployment model generation time based on the algorithms, and we compared total communication and deployment costs in terms of the applied algorithms. The proposed tool enables modeling a microservice-based application end-to-end and generating effective deployment configurations by considering the resource constraints of nodes at the early design phase.

The rest of this article is organized as follows: Section 2 gives the necessary information to understand better the study's purpose describing the microservice concept and the efficient deployment generation problem of microservice-based applications. Section 3 clarifies the problem statement and defines a case study to evaluate the proposed tool. Section 4 describes the proposed approach to generate efficient deployment alternatives. Section 5 describes the metamodels that support the approach. Section 6 explains the extraction of algorithm parameters from the design model and the adaptation of the efficient deployment generation algorithms. Section 7 describes the Micro-IDE tool support through the case study. Section 8 presents the evaluation of the approach and tool support by comparing the algorithms used. Section 9 provides the discussion together with the threats to validity. Section 10 describes the related work and, finally, Section 11 concludes the article.

2 | MICROSERVICE ARCHITECTURE

Enterprise software applications often have hundreds of functionalities to meet a variety of business requirements. Until recently, these functions were generally developed in single, monolithic applications. Monolithic applications are subject to cause scalability issues in means of both functionality and development teams. Furthermore, scaling individual functionalities on-demand or updating a single functionality without affecting other functionalities becomes almost impossible in a monolithic application at some point.[7] Any update in a single functionality can trigger cascading updates in other parts of the application in monolithic applications, requiring a complete rebuild, installation, and regression

testing of a broad set of capabilities, including the unchanged ones. To deal with these problems, the concept of "service" is introduced, and service-oriented architecture (SOA) has become popular.[8] There are two types of roles in the SOA concept: service provider and service consumer. The primary motivation of the SOA approach is to design and build services in such a way that they can be seamlessly integrated and easily reused. Thanks to this approach, the decomposed services can be reused in other applications, are easier to manage, have higher reliability, and can be developed in parallel in different teams. However, the management of the services is still an issue, and the decomposition brings extra data exchange load to the system.

The MSA initially inspired by the evolution of SOA. Compared with SOA, MSA involves more granular services that are small, autonomous, and loosely coupled. The MSA is based on the idea that a single application can be built as multiple services that can be developed and deployed independently.[9] The independence of the services facilitates scalability, deployment, and well-defined interaction of the services. Both MSA and SOA operate on services as the main component, but they differ significantly in service characteristics. While SOA adopts the "share as much as possible" approach, microservices adopt the "share as little as possible" approach.[10]

Moreover, the communication infrastructure of SOA and MSA is quite different. While SOA solutions generally use a homogenous communication infrastructure (enterprise service bus [ESB]), MSA often uses more lightweight messaging solutions such as HTTP/REST, gRPC,[11] and various communication protocols can be used among different services. The major disadvantage of ESB over MSA's lightweight, heterogeneous communication architecture is that all connected services are affected when there is a problem with the ESB itself. In this case, ESB leads to the whole system crash, and this situation creates a single point of failure.[12] MSA is more fault-tolerant since any problem occurring in a microservice only affects this service. The other services continue to serve if they do not heavily depend on the degraded services. Deciding on the adoption of microservices or SOA approaches heavily depends on the technical requirements. Especially enterprise companies tend to use SOA for specific motivations such as data governance. Microservices are more commonly used in web and mobile applications to ease development. Besides that, the SOA architecture can be used as a steppingstone from monoliths to microservices.[13]

A conceptual model of an MSA is shown in Figure 1. Multiple clients can send requests to the microservices using the API layer, which is the entry point for all the client requests. The API layer also provides the mechanism for the

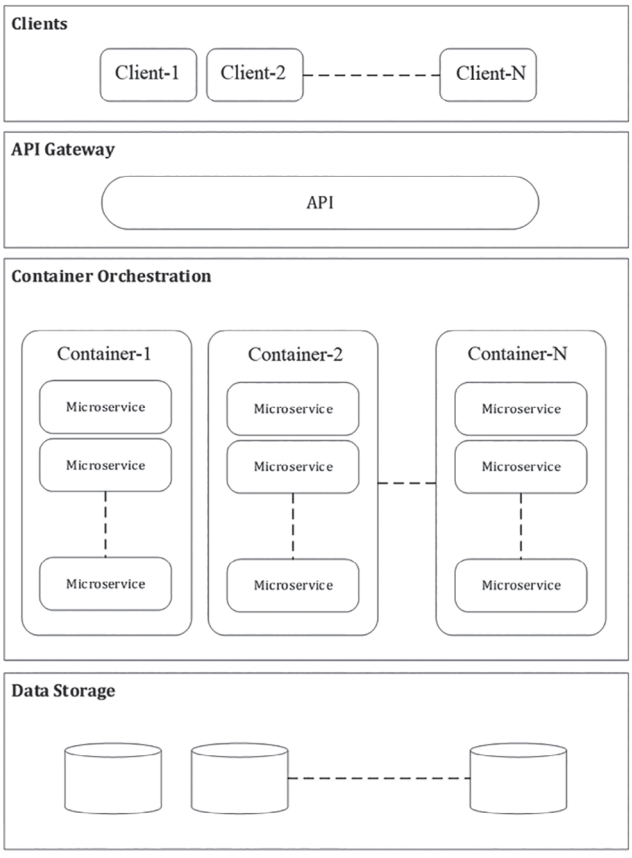


FIGURE 1 Conceptual model of microservice architecture

microservices to communicate with each other. The container orchestration layer includes the different containers that provide a lightweight runtime environment in which the microservices run. Different technology stacks and programming languages can be used for the development of each microservice. Finally, the data storage layer provides persistence through mechanisms, such as databases and files. Each microservice might use a separate persistent data storage, one of the major differences between MSA and SOA.

3 | CASE STUDY AND PROBLEM STATEMENT

We describe a case study in the following subsection and use this case study to describe the problem statement in the second subsection.

3.1 | Case Study—Spotify: Music and podcasts application

We tried to find an easy-to-understand microservice case study and selected the Spotify[14] application, which millions of people use. We performed research on Spotify architecture from public resources,[15] and as far as we have investigated, approximately 18 microservices have been identified in the Spotify application. The application consists of hundreds of microservice instances to provide a smooth experience to millions of concurrent users. Based on this information, we created a hypothetical runtime scenario given in Table 1. The number of instances indicates how many microservice instances are defined for each service type. The third column shows the amount of memory required for the execution of the corresponding microservice in megabytes. According to the values defined in Table 1, 1361 microservices require a total amount of 40,820 MB memory.

TABLE 1 The properties of the microservices for the case study

Microservice name	Number of instances	Required memory per microservice (MB)
Delivery Ingestor	50	20
Transcoder Control	58	25
Transcoder Service	68	15
Ingester	66	10
Merger	64	10
Non-Music Metadata Pipeline	78	30
Vmd2-cms	98	40
Content API	96	20
Scatman	76	25
Indexer	78	25
CurationUI	74	25
Preludex	77	20
NERD	66	30
ENsnaphot	87	30
Google docs	84	45
Group Curation	93	35
Splash Content	92	50
Vmd2-service	89	40
Total	1361	40,820

3.2 | Problem statement

Considering the scenario described in Section 3.1, generating an efficient deployment model with expert judgment manually becomes hard in a system consisting of many microservices. Although, application developers may find efficient deployment configurations, finding an efficient deployment configuration only once is not enough since the configuration will be changed when the application is updated, for example, a service starts consuming new data. Besides, manually finding an efficient deployment for complex microservice-based applications consisting of thousands of services is not tractable. In particular, microservice-based applications' resource usage and performance depend on the efficient deployment of microservices to the available resources. Therefore, developing and operating a large-scale system consisting of hundreds or even thousands of services is not trivial.

The efficient deployment concept means minimizing the total execution cost of microservices on existing nodes and the total communication cost among microservices. Since each node can accommodate the maximum polynomial amount of microservices, and deploying communicating microservices to different nodes will raise communication costs, the deployment problem is an NP-hard multi-objective optimization problem.[2] This problem needs to be solved by optimizing total communication and execution costs on the design phase of the created MSA.

Deploying frequently communicating services to the same physical resources reduces the total communication cost, which is vital in application performance and scalability. In addition to the communication cost among services, CPU and memory usage of the services, execution cost of the services on the different nodes need to be considered to optimize the deployment of microservices to limited capacitated resources. For example, installing many microservices with high resource consumption on the same node will cause problems in performance and scalability. Besides, service upgrades during communication among microservices will change the resource consumption and overall system performance.[1]

A microservice-based application is deployed by considering the required resources of each microservice and available resources. Although container orchestration tools such as Kubernetes,[2] Docker Swarm,[3] and Apache Mesos[4] allow developers to specify resource utilization thresholds (e.g., CPU and memory) for each microservice, there is no guarantee that users will set this threshold correctly. Developers often set these values based on the insights gathered from previous executions of the system or their subjective experience. If only the minimum required amount of resources is set, deploying many microservices together on a single host may be possible. Although all microservices deployed on the same host reduce the total communication cost to zero, this deployment alternative is not feasible since a single physical resource has capacity limits. Even if the resource limits are set correctly for individual microservices, there is no guarantee that the selected values will result in an efficient deployment configuration for the overall application because inter-service costs such as network communication are not considered. To sum up, it is challenging to set the resource requirements for a microservices-based application.

During the deployment phase, the cluster provider tries to balance the load among servers without compromising the performance of the microservice-based application. However, developers' lack of standardization in determining their resource requirements makes it challenging to deploy microservices.

On the other hand, some management tools that use many strategies such as Spread, Bin-pack, Labeled, Random do not use historical data to direct or improve the deployment of microservices.[1] The existing tools select hosts that consider using instantaneous resources to deploy the microservice, and they can rarely find an efficient deployment alternative. In this study, a tool called Micro-IDE is proposed to generate efficient deployment alternatives for microservices. The tool aims at finding efficient deployment alternatives using algorithmic approaches used for solving NP-hard problems. The tool focuses on minimizing total cost considering communication costs of microservices during data exchange, the execution costs on the servers, the amount of memory required, and the amount of memory and CPU necessary for the servers to host microservices. In the following section, we elaborate on and describe the approach in detail.

4 | APPROACH FOR GENERATING MICROSERVICE DEPLOYMENT ALTERNATIVES

Deployment architecture decision generally is deferred to the post-development phase, where it is too late and expensive to change the system design. We have focused on this problem and defined an approach that tries to generate efficient deployment alternatives at the early design phase, even before the actual system implementation is started, and this section explains the approach.

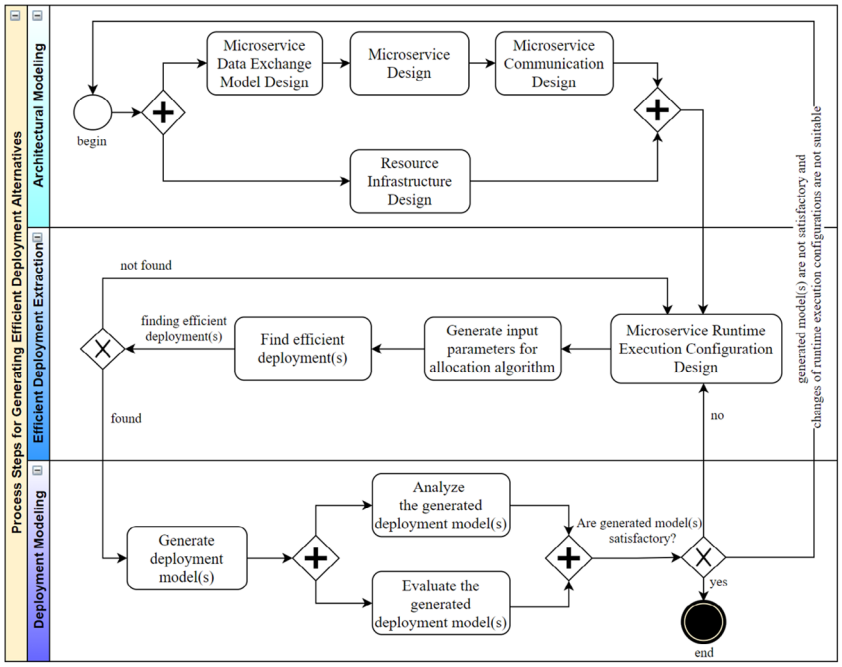


FIGURE 2 BPMN diagram of the proposed approach

The approach is based on generating efficient deployment alternatives with minimal total costs, evaluating the generated alternatives, and analyzing the performance of these alternatives. The approach is supported by a tool (Micro-IDE), which will be discussed in the next section. The steps of the approach are shown in a BPMN diagram in Figure 2.

Summaries of the activity steps in Figure 2 are given below:

1. Microservice data exchange model design: Microservices need to exchange data during communicating with each other. In this step, we enable the designer to create data objects and corresponding types for data exchange. Since there are no homogenous data exchange media in the MSAs, we have researched data exchange platforms and selected the gRPC as the reference model since it has the most comprehensive data model.
2. Microservice design: At this step, the designer defines the properties of the microservices, such as the name, type, version, and endpoints.
3. Microservice communication design: At this step, the designer matches the microservices defined in step 2 with the data exchange model elements defined at step 1.
4. Resource infrastructure design: The designer defines the nodes that the microservices will be deployed at this step. The memory capacities, processor features, and power factors of the nodes can be specified, and the connection of these nodes can be designed as local area network (LAN) and wide area network (WAN). Many nodes with different memory capacities and processor powers can be created using this model. Furthermore, multiple processor units with different frequencies and core counts can be defined in a single node. This step can be designed independently from the previous steps.
5. Microservice runtime execution configuration design: Since this step, only the structural properties of the system are defined. We mean data exchange elements, static properties of the microservices, data production/consumption definitions, and node configurations by structural properties. At this step, the designer identifies the runtime execution scenarios of the designed microservice-based system. The structural properties are not enough to decide on efficient deployment models since the system's runtime properties, such as instance count for each microservice and data update rates, are also important. At this step, the designer defines the number of microservices, the update rate of a microservice instance in each communication, and the execution cost of each microservice instance on each node.
6. Extracting input parameters for generating deployment alternatives: After performing all the steps mentioned above, input parameters such as communication costs among microservices, memory capacities of microservices and nodes, execution cost of microservices on different nodes to generate deployment alternatives are extracted from the developed models.

7. Deriving efficient deployment alternatives: After extracting the input parameters in step 6, these parameters are given as inputs to the algorithms to generate efficient deployment alternatives. According to the selected algorithm, the approach derives one or more deployment alternatives and presents the results to the designer, who can then select an efficient deployment model. If a feasible deployment alternative cannot be found, the system provides detailed information to the designer to optimize the system design.
8. Analyzing the results and comparing the generated deployment models: In this step, the designer can prefer the ideal CTAP algorithm applied in the Micro-IDE tool by comparing the total execution cost and communication cost of the generated deployment models. The system also compares manually created deployment models and algorithmically generated models in terms of execution and communication costs. The Micro-IDE tool provides automatic analysis of the generated deployment models according to different algorithms and quality factors. The proposed tool enables selecting appropriate deployment models for the designed architecture by comparing the generated deployment models.

5 | METAMODELS

To perform the steps of the proposed approach and the implementation of the Micro-IDE tool, several metamodels named microservices data exchange, microservice definition, communication, infrastructure, runtime execution configuration, and deployment metamodels are created. These metamodels are described in the subsections below.

5.1 | Microservices data exchange metamodel

This metamodel defines the model in step 1 described in Section 3 and the data types required to perform data exchange among microservices. Microservices need to exchange data while communicating with each other. We preferred the gRPC communication infrastructure in this metamodel because it has the most comprehensive data types among the other communication protocols such as publish/subscribe, REST, graphQL, and so forth. Choosing the most comprehensive communication infrastructure ensures covering all data types in communication methods used by microservices.

5.2 | Microservice definition metamodel

Microservice definition metamodel is used to design microservices mentioned in step 2 described in Section 3. This model allows the designer to determine many properties of a microservice, such as a name, type, storage unit, version, and endpoint information. The most generated metamodels (communication, runtime execution configuration, and deployment metamodels) need the microservice definition model. The communication infrastructure, the parameters of runtime execution configuration, and the deployment process are performed through the defined microservices in this metamodel.

5.3 | Microservice communication metamodel for determining the relation of microservices

Microservice communication metamodel is designed to create the communication infrastructure model in step 3 described in Section 3. In an MSA, each service has a single functionality that solves a problem in the application. For the application to work as a whole, microservices must communicate using various communication protocols according to each service's nature. Thus, a communication metamodel is created that allows exchanging data between microservices and determining communication protocol. This metamodel requires microservice definition metamodel and microservice data exchange metamodel. The microservices defined in the microservice definition metamodel communicate using the data objects defined in the microservice data exchange metamodel. The CommunicationModel class in the metamodel represents the following protocols used for the communication among microservices: RestOperation, graphQL, gRPC, and pub/sub communication protocols.

5.4 | Microservice infrastructure metamodel

Microservice infrastructure metamodel is created to model the physical resources mentioned in step 4 described in Section 3. It includes many classes representing the identification of physical resources for deploying services, determining processor features and memory capacities, and setting network connections between resources. The MicroPhysicalResourceModel class seen in the metamodel defines one or more Nodes to which services are assigned. The PowerFactor property in the Node class is used to determine the processing power compared to other nodes. Since a node can have more than one processor and many custom node properties, the relationship between Node and Processor is expressed in the metamodel with a one-to-many ($1 \ldots *$) relationship. The processor's core number, frequency, and processor name can be determined using the Processor class. Additional requirements (such as disk capacity, graphics card speed) of a node can be specified using the CustomNodeProperty class. The memory amount of the processor can be defined as MegaBytes using the MemoryCapacity class. These identified nodes can be connected with multiple networks via LAN or WAN. Therefore, the Network class is defined as an abstract class for the LocalAreaNetwork and WideAreaNetwork classes. WideAreaNetwork (WAN) class has speedFactor property to determine how many times the WAN is slower than LAN. The Router class defines the routers used to connect networks. The LANConnection class is defined as a GMF link in the metamodel and represents nodes' connection with the local area network. Similarly, the LANRouterConnection class represents the connection of a local area network to a router, and the RouterNetworkConnection class represents the connection of a router to a network.

5.5 | Runtime execution configuration metamodel

Runtime execution configuration metamodel defined in step 5 of the approach requires all the metamodels described in the previous subsections for execution. The MicroRuntimeExecutionModel class includes Metadata and MicroserviceInstance classes to define a single microservice. The MultiMicroserviceInstance class represents multiple microservice instances, and the Publication class represents the update rate of an ObjectModel. The MicroserviceInstance class contains the Publication and ExecutionCost classes used to determine the update rate of data related to a microservice and the execution cost of the services on each node, respectively. The MicroserviceInstance class needs the microservice definition metamodel for identifying microservice instances associated with the microservices. For instance, DeliveryIngestor, which is a service of the Spotify application, is created once in the microservice definition metamodel. The Publication class needs the ObjectModelElement class defined in microservice data exchange model to determine the data related to the service. In the runtime execution configuration metamodel, many properties for the execution such as the microservice instance name (name), instance count (instanceCount), and the required memory (requiredMemory) of the microservice, and the associated microservice name (relatedMicroservice) in microservice definition metamodel are determined. The relatedMicroservice property can be selected by listing the microservices defined in the microservice definition metamodel. The relatedElement property in the Publication class is chosen using the ObjectModelElement defined in the microservice data exchange metamodel. The updateRate property in the Publication class refers to the update frequency of the selected ObjectModelElement. Also, the ExecutionCost class represents the execution cost of a microservice on a node. It can be defined as a different execution cost of a microservice for each node defined in the microservice infrastructure metamodel. The designer can concretely define each of the DeliveryIngestor instances to update the Address object twice per second. Depending on the processing power, the execution cost for each DeliveryIngestor instance can be defined as 10 out of 25 for one node, 15 out of 25 for another node, and so forth.

5.6 | Microservice deployment metamodel

Microservice deployment metamodel—used to define the model mentioned in step 6 described in Section 3—includes members, nodes defined in microservice infrastructure metamodel, and microservice instances defined in microservice runtime execution configuration metamodel. The Member class contains zero or more microservices deployed on a node. The DeployedMicroservice class includes the deployed microservice's name and the instance count of the microservice. Additionally, this class has a reference link to the MicroserviceInstance class defined in the microservice runtime execution configuration metamodel to define microservice instances associated with microservices defined in the microservice

definition metamodel. The overall metamodels mentioned in this section and their relationships are shown in the Appendix. Furthermore, each metamodel has been explained in our previous work with more details.[6]

6 | MAPPING DESIGN ARTIFACTS TO CTAP PARAMETERS

After all the models have been created, the input parameters required to deploy microservices are extracted from these models. These parameters include the memory capacities of the services, execution costs of the services on the nodes, and the memory capacities of the nodes. The efficient deployment problem for the microservices matches the "capacitated task assignment problem (CTAP)" in the literature. CTAP is the problem of assigning n jobs to m nodes with minimum execution and communication costs. This NP-hard problem is an extension of the task assignment problem (TAP). Unlike TAP, the memory capacities of resources are limited in CTAP, and communication costs between tasks are considered during the assignment. Although this problem is used as a model in distributed computer systems, it is also adopted for many optimization problems such as job scheduling, vehicle routing, shortest route finding, telecommunication network design, cargo loading. We adopt the CTAP in the proposed tool to solve the efficient deployment problem of MSAs. With the adoption of the mathematical model of the CTAP, we aim to minimize the communication cost between microservice instances and the execution cost of the services on the nodes. While the total cost is minimized, the total memory capacity and processor power of the services deployed to the related node should not exceed the node's memory capacity and processor power. The model components matched with the parameters in this problem are expressed as follows:

S, set of m microservices $= s_1, s_2, \ldots, s_m$
S_i^j, j th instance of i th microservice
N, set of n nodes $= n_1, n_2, \ldots, n_n$
N_c, the number of nodes
N_m, the number of microservices
k_i, the number of microservice instances
M_n, memory capacity of nodes n
mem_i, amount of memory needed for i th microservice
x_{in}, cost of executing s_i microservice on the n th node
a_{in}^j, S_i^j is assigned to Node n
E, a set of communication between microservices, whereby each communicating service combination (i, j) has a communication cost c_{ij} if microservices s_i and s_j are assigned to different nodes. Communication cost is negligible if two microservices are assigned to the same node.

The minimum cost achieved by using these parameters is formulated as follows:

$$\text{Total minimum cost} = \sum_{i=1}^{N_m}\sum_{k=1}^{k_i}\sum_{n=1}^{N_c} a_{in}^j x_{in} + \sum_{(i,j)\in E}\sum_{x=1}^{N_m}\sum_{y=1}^{k_x} \sum_{n=1}^{N_c} a_{in}^j (1 - a_{xn}^y) c_{ix}$$

Subject to,

$$\sum_{n=1}^{n} a_{in}^j = 1, \quad i \in S, j \in k_i,$$

$$\sum_{j=1}^{k_i}\sum_{n=1}^{N_c} a_{in}^j = 1, \quad i \in S,$$

$$\sum_{i=1}^{m}\sum_{j=1}^{k_i} mem_i a_{in}^j \le M_n, \quad n \in N,$$

$$\sum_{i=1}^{N_m}\sum_{j=1}^{k_i} x_{in} a_{in}^j \le C_n, \quad n \in N,$$

$$a_{in}^j = 0, 1, \quad n \in N, i \in S (a_{in}^j = 1, \text{ if } S_i^j \text{ is assigned to Node } n, 0 \text{ otherwise}).$$

In the adopted CTAP problem, it is assumed that m microservices in the system and microservice i needs mem_i units of memory. There are c nonequivalent nodes in the storage system, Node n has a total of M_n unit memory and C_n unit processing capacity. Running microservice i on node n costs x_{in}. a_{in}^j represents the i th microservice's j th instance is assigned to Node n. To calculate total execution cost of i th microservice, a_{in}^j and x_{in} are multiplicated for all i, j, and n values, then these results are summed up. In this formula, if it is assigned to Node n, a_{in}^j is equal to 1, otherwise it means that S_i^j is not assigned to Node n and the formula returns 0.

This NP-hard problem is solved using various algorithms such as Genetic, Hungarian, Next Fit, Best Fit, and so forth. The aim is to find a deployment of microservices to available nodes at minimum cost.

7 | MICRO-IDE TOOL SUPPORT

In this section, a tool framework that supports the approach described in Section 3 is illustrated, and all metamodels are integrated for runtime implementation. The Micro-IDE tool is based on metamodels described in Section 5 and algorithmic approaches adopting the CTAP algorithm. This tool operates in plug-in sets on the Eclipse-IDE platform. These plug-ins are built on Eclipse Framework plug-ins such as Eclipse Modeling Framework (EMF),[16] Graphical Editing Framework (GEF),[17,18] and Graphical Modeling Framework (GMF).[19] EMF is an Eclipse plug-in set used to design the data model and derive code and other outputs based on the model. There is a distinction between metamodel and model. While a metamodel describes the structure of the model, a model is a concrete instance of this metamodel. EMF allows the developer to create metamodels with different extensions such as XMI, Java comments, UML, or an XML schema. Besides, the default application ensures continuity of model data using a data format called XML Metadata Interchange. The GEF is an Eclipse project that provides end-user components and views associated with graphical applications. The GMF is an Eclipse modeling project that aims to provide a productive bridge between EMF and GEF. Creating an Ecore model using GMF is derived from the domain model, domain gen model, graphical def model, tooling def model, mapping model, and diagram editor gen model via the GMF dashboard. In this study, a text editor named Emfatic[20] is used to generate Ecore models. Besides, EuGENia[21] GMF tool is used, which is more practical than the GMF dashboard for generating all needed models such as .gmfgraph, .gmftool, and .gmfmap models using generated Ecore models. The layered architecture of the Micro-IDE tool is shown in Figure 3.

In the following subsections, the general perspective of Micro-IDE tool architecture is described (Section 7.1), and Micro-IDE tool design is defined according to a selected case study (Section 7.2). Besides, the generated deployment alternatives using the Micro-IDE tool are examined through the case study (Section 7.3).

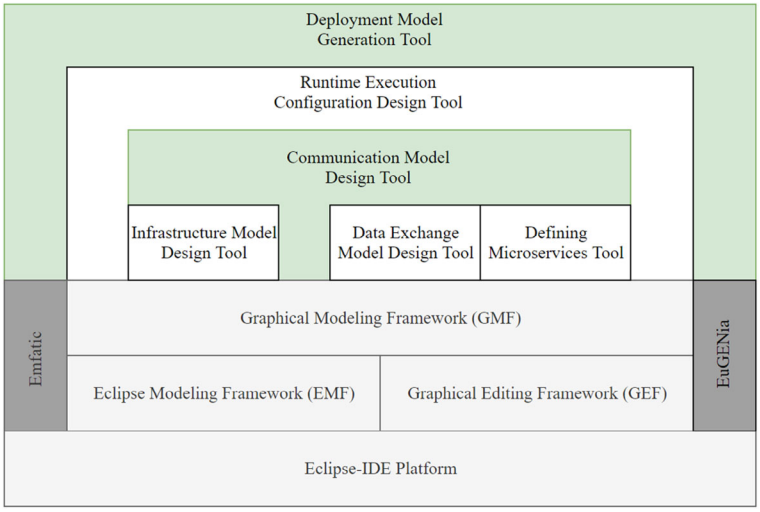


FIGURE 3 Layered architecture of Micro-IDE tool framework

7.1 | Tool architecture

The Micro-IDE tool consists of five fundamental parts: model navigator, model editing pane, item palette, and properties view, as shown in Figure 4. The Model Navigator on the left side of Figure 4 shows the created metamodels and the elements used in these models. The Model Editing Pane in the middle provides a drawing area for designing objects and connections. The Properties View at the bottom serves to list the properties of the elements drawn in the Model Editing Pane and the metamodels in the Model Navigator. The Item Palette on the right offers the necessary components for drawing in the Model Editing Pane.

7.2 | Usage of Micro-IDE for implementing the case study

In this section, the music application developed by the Spotify company is used to illustrate the Micro-IDE tool. The created models for this case study and the generated deployment model using this tool are described below.

7.2.1 | Designing microservice data exchange metamodel—Object models and data types

While designing the microservice data exchange model for the Spotify application, object classes used in this application have been determined as the first step. Object classes, attributes, and data types for the case study are shown in Figure 5. Since user information is required for Spotify membership, a User object class has been created. This class contains *userNo* , *password* , and *email* attributes. In our scenario, the User class inherits some object classes called Musics, Podcasts, member payment information (Interests), search list (Browse), and so forth. Since these classes are user-specific, the User class is designed as a root class. The Subscribing and *PremiumTypeEnum* object classes, defined as *EnumeratedDataType* , allow selection of payment type and membership type, respectively, in the *PaymentInfo* class. The *Subscribing* class includes the payment type that enables the user to subscribe by credit card, phone bill, or prepayment. Similarly, *PremiumTypeEnu* m object class consists of the membership type called individual, student, or family. A member is charged according to the

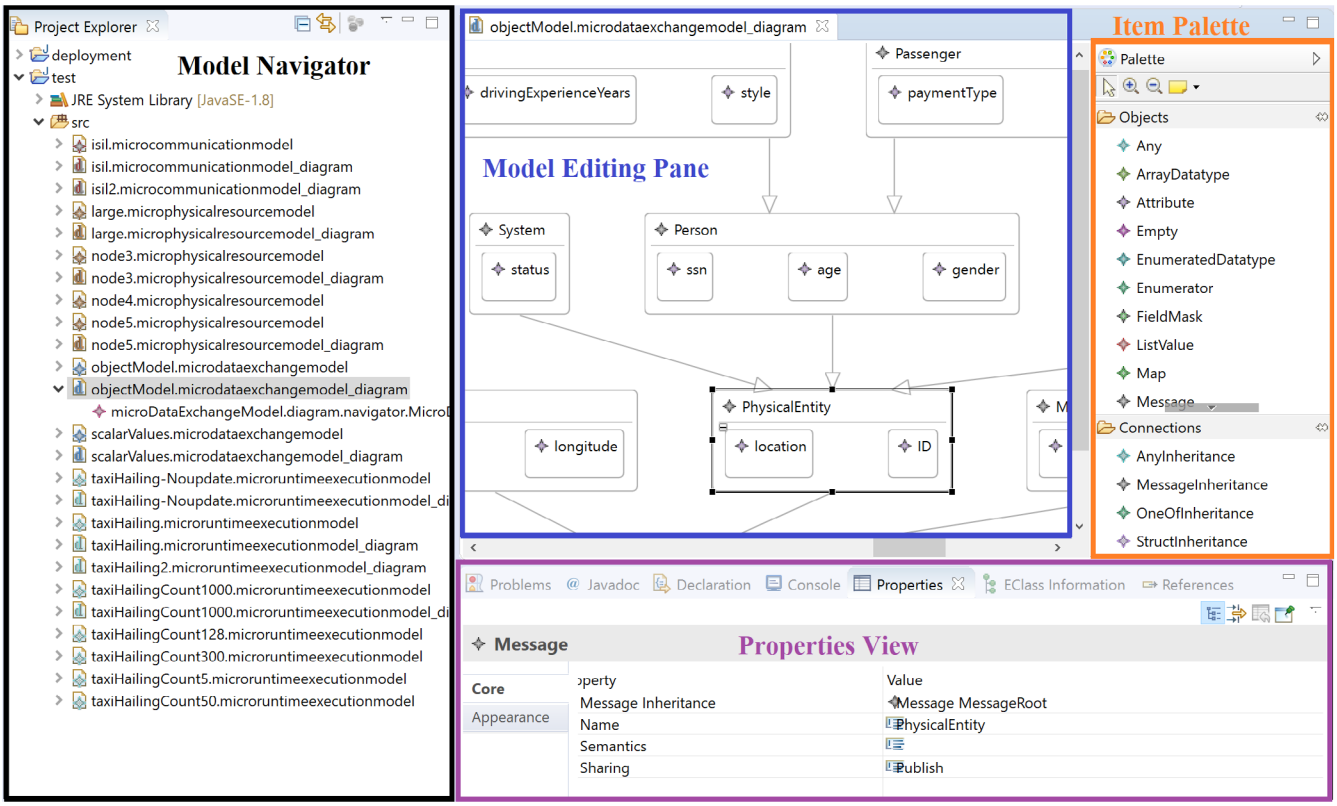


FIGURE 4 Fundamental parts of Micro-IDE tool

FIGURE 5 Microservice data exchange model for the case study—Object classes, attributes, and their data types

selected property. Furthermore, *Artist*, *Album*, and *Song* classes whose root class is Musics include *name*, *genre*, *rating*, and *listenCount* properties.

7.2.2 | Identification of microservices and determining the communication patterns

Within the scope of this case study, 18 microservices shown in Table 1 are defined, and the gRPC communication pattern is created to communicate the services with each other. Communication patterns designed on the microservices are shown in Figure 6. Once the communication pattern is selected, gRPCtype (protoRequest-ProtoResponse), name of the relationship, source (microservice name), and target (object class) properties must be identified.

In the scope of this study, the identified communication patterns among microservices are represented in Figure 7. In the figure, Merger, PreludexVoting, TranscoderControl, Ingester, contentAPI, GroupCuration, TranscoderService, CurationUIs, NonMusicMetadataPipeline, vmd2-cms, Scatman, and NERD represent the microservices. The message types (Podcasts, Interests, Song, PaymentInfo, etc.) refer to the objects used by microservices.

7.2.3 | Establishment of physical infrastructure for the deployment of microservices

The physical infrastructure required for deploying microservices can be established by creating the microservice infrastructure model. In the scope of this case study, we designed an infrastructure model consisting of four nodes with different processors and different memory capacities. As shown in Figure 8, some nodes may have more than one processor, such as Node-4. These four nodes are connected via a local area network connection. The proposed tool allows creating the desired number of nodes and heterogeneous LAN/WAN connections.

7.2.4 | Defining microservice instances using microservice runtime execution configuration metamodel

In microservice runtime execution configuration metamodel, many properties such as the number of instances (instanceCount) of a microservice, related microservice for the microservice instance (relatedMicroservice), and the

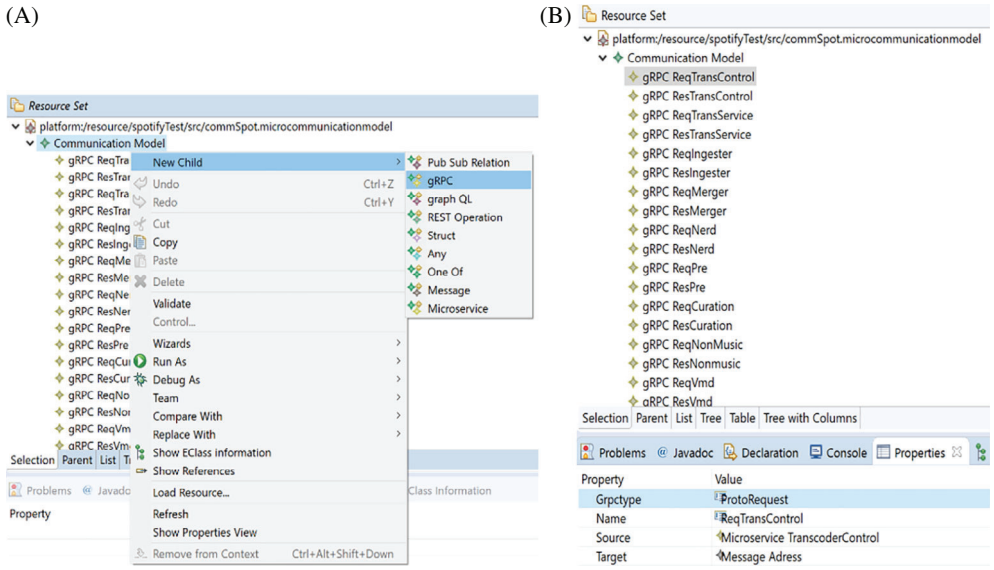


FIGURE 6 (A) Establishing a communication pattern on microservices. (B) Properties of the selected communication pattern

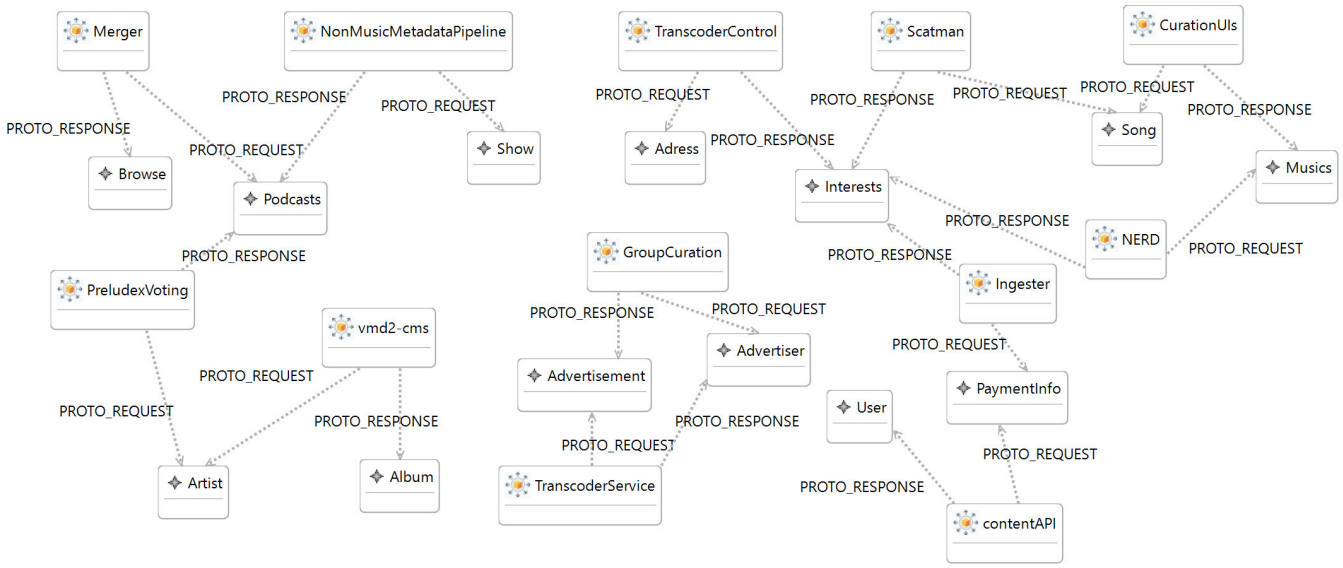


FIGURE 7 Identified communication patterns among microservices for the case study

amount of memory needed to store the microservice instance (requiredMemory) can be defined. Besides, Publication and ExecutionCost classes are added as an attribute in a microservice instance. Thus, the frequency of updating the data (updateRate) and the execution cost of a microservice instance on each node are determined. A part of the sample microservice runtime execution configuration model designed for the Spotify application is given in Figure 9. MultiMicroserviceInstances are defined for 18 microservices described on the microservice definition model.

This model uses the microservice infrastructure model to determine the execution cost of the microservices on the nodes, the microservice data exchange model to establish relations with microservices and objects, and the microservice definition model to determine the corresponding microservice of an instance. An example scenario in which the number of microservices is listed for the case study is given in Table 1.

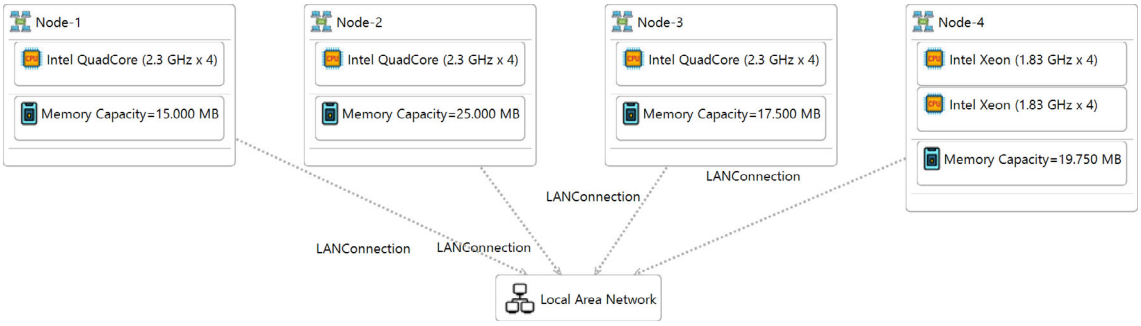


FIGURE 8 Four nodes physical infrastructure model created for the case study

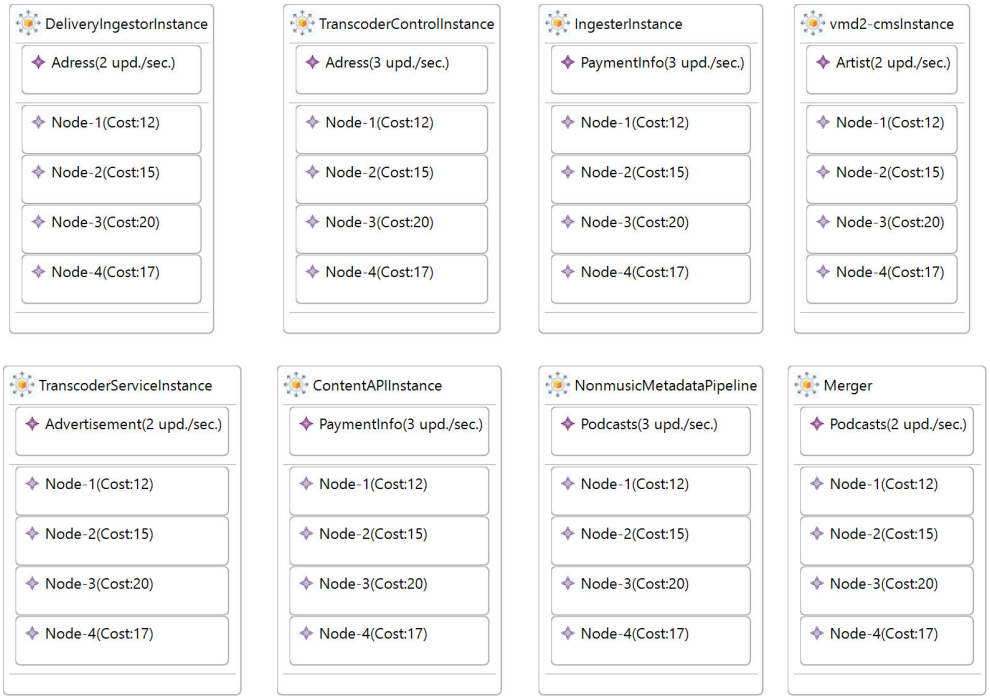


FIGURE 9 A part of the microservice runtime execution configuration model for the case study

7.3 | Generating efficient deployment alternatives for the case study

After determining the physical and structural requirements for the Micro-IDE tool to deploy the microservices, the automated deployment process for the microservice instances described in Table 1 is performed using algorithmic approaches. The pseudocode of the algorithm for implementing the efficient deployment alternatives is given in below:

```
GENERATE_FEASIBLE_DEPLOYMENT (phy_resources,runtime_exec_config)
processors EXTRACT_PROCESSORS (resources)
microservices EXTRACT_TASKS (exec_config)
assignment_tables EXECUTE_CTAP (microservices, processors)
CREATE_DEPLOYMENT_MODEL (allocation_table)
```

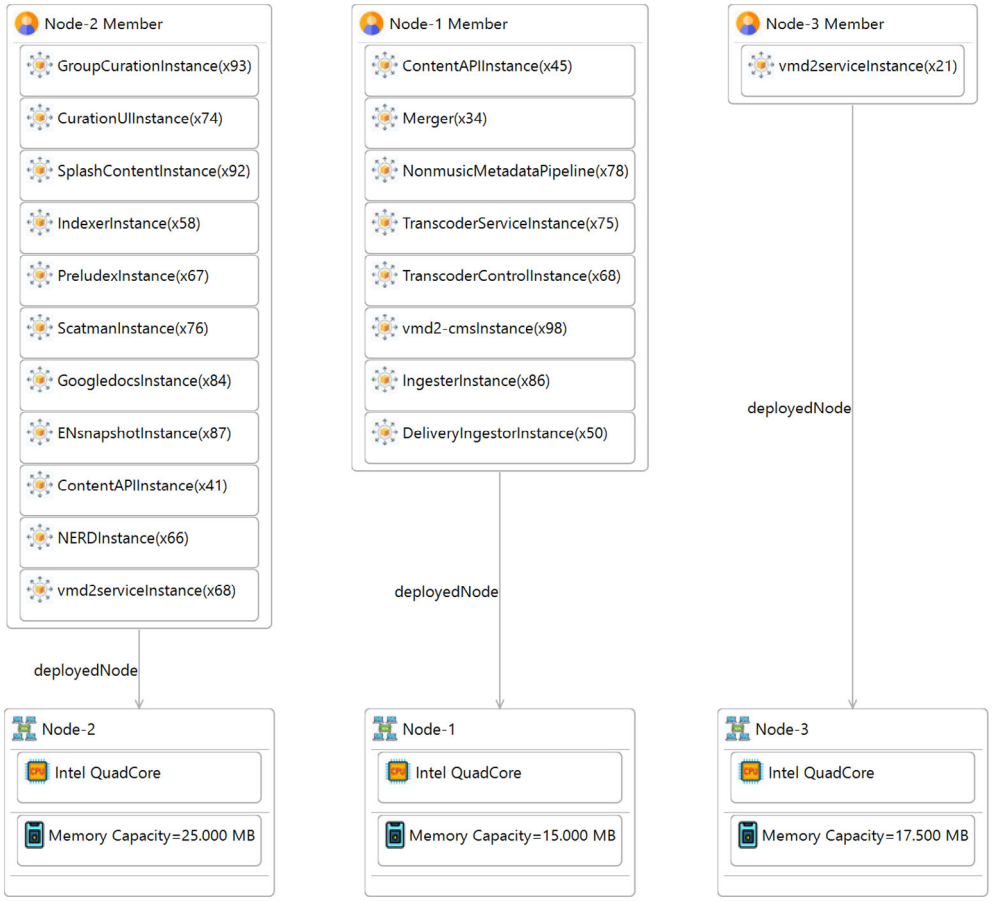


FIGURE 10 A sample deployment alternative generated by the Next Fit algorithm

In the first line of the algorithm, *GENERATE_EFFICIENT _DEPLOYMENTS* gets two parameters to use the microservice infrastructure model and the microservice runtime execution configuration model. *EXTRACT_ PROCESSORS* represents the processor specified for each node in the microservice infrastructure model in the second line. Using this method, feature extraction of the processors in the microservice infrastructure model is performed. In the third line, the microservice instances are extracted from the microservice runtime execution configuration model using the *EXTRACT_TASKS* method. This method extracts the execution costs of microservices on the nodes and the communication costs among the microservices corresponding to the concept of a task in the CTAP algorithm. In the fourth line, the CTAP algorithm is applied with the *EXECUTE_CTAP* method. The method takes processors and microservices as parameters. According to the cost information received from the methods, the deployment of microservices to the processors is stored in the table named *assignment_tables*. Each member assigned to the processors in the *assignment_tables* represents an abstract definition of an efficient deployment alternative. In the last line, alternative deployments are generated with the *CREATE_DEPLOYMENT_MODELS* method using the parameters in this table. The details of each method and overall pseudocode for generating efficient deployment alternatives can be found in our previous work.[6] In this study, various algorithms adapted to the CTAP algorithm are used to deploy microservice instances defined on the microservice runtime execution configuration model. Improvement rates of the algorithms are calculated according to total execution and communication costs. In Figure 10, a sample deployment alternative using the Next Fit algorithm can be seen.

8 | EVALUATION

In this section, the Micro-IDE tool is evaluated on the case study, and the efficiency of generated deployment alternatives has been discussed using algorithmic approaches. Besides, the algorithms have been compared in terms of the generation time of deployment alternatives according to different scenarios.

8.1 | Performance comparison of applied algorithms

After all necessary models for the case study are designed, the deployment model generator wizard can be used by selecting Microservice Runtime Execution Configuration and microservice infrastructure model, as shown in Figure 11.

Furthermore, an interface where the generated models are analyzed according to the communication model, data exchange model, physical infrastructure model, and the number of microservice instances has been developed in the Micro-IDE tool. The .results file, which reports the detailed analysis of the generated deployment model, is obtained by selecting the Microservice Runtime Execution Configuration and microservice infrastructure model, as shown in Figure 12. This file (.results) gives information about the memory capacities of the microservices, the object sizes used by the services, the communication costs between services, and the memory capacities of the physical resources selected on the tool.

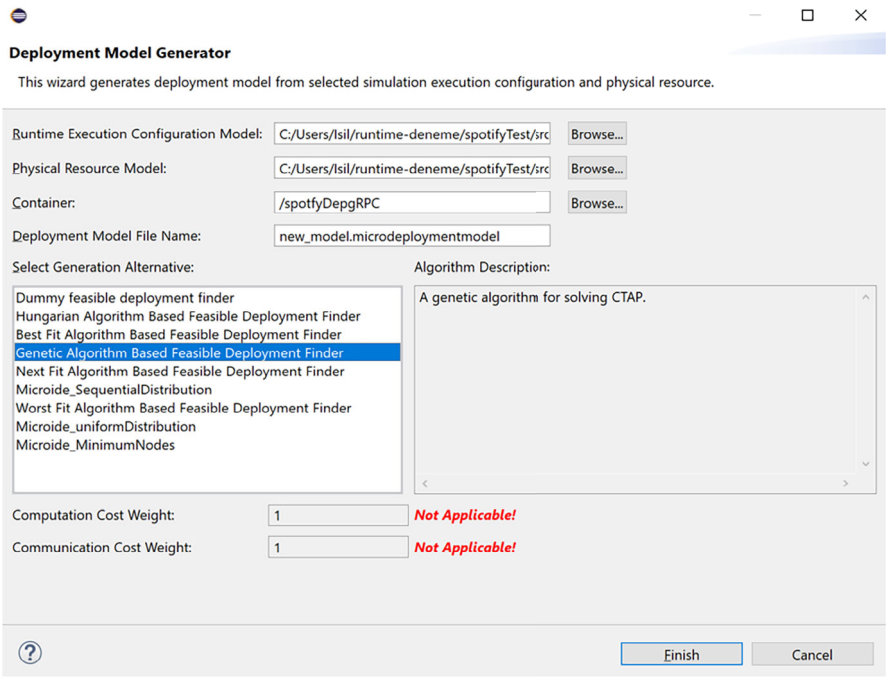


FIGURE 11 The model generation interface of the Micro-IDE tool

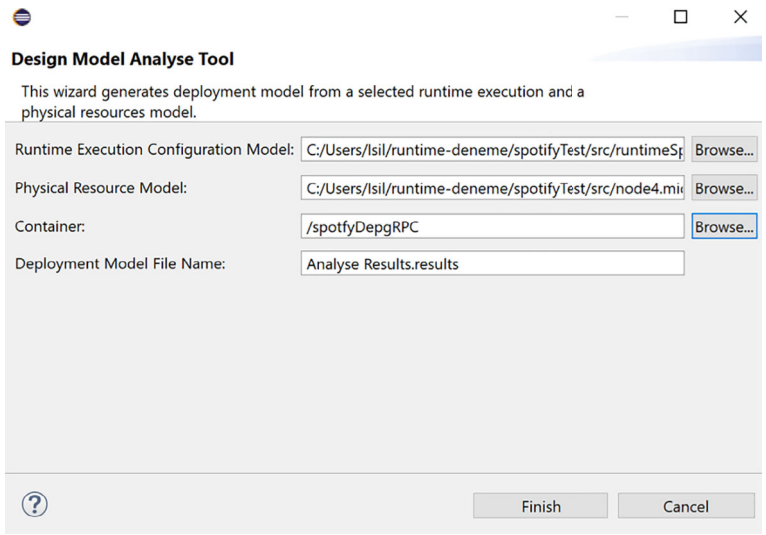


FIGURE 12 The model analysis interface of the Micro-IDE tool

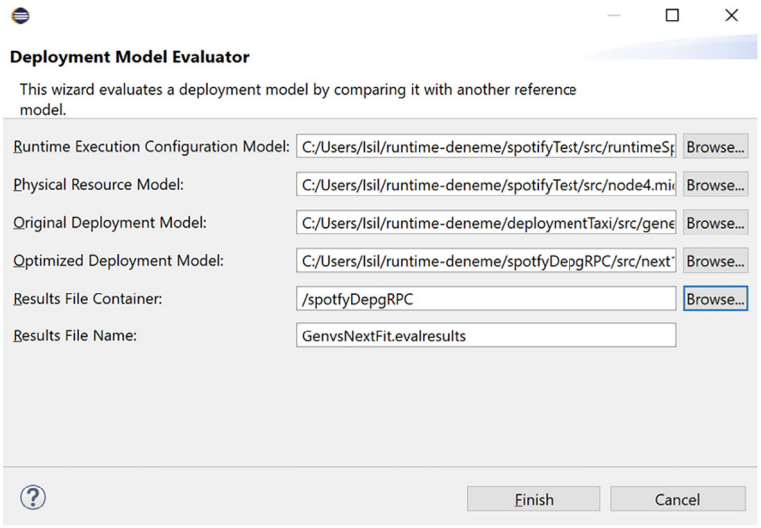


FIGURE 13 The model evaluator interface of the Micro-IDE tool

TABLE 2 The communication and execution costs values for the generator

Algorithm	Total communication cost (MB/s)	Total execution cost on the nodes
Hungarian	265.350	21.425
Genetic	266.546	21.970
Next Fit	195.017	21.603
Best Fit	214.883	22.390
Worst Fit	260.681	21.881
Sequential Distribution	266.945	21.772
Uniform Distribution	251.845	21.844
Minimum Nodes	171.03	21.855
Manual Deployment	268.073	21.676

When the information obtained from the analysis interface is examined, the total communication costs of the microservice instances are calculated by using the number of instances and object sizes used by the microservices defined in the microservice data exchange model. Besides, the amount of memory required for the deployment of each microservice instance is listed in the .results file by giving information about the memory capacities of the physical resources.

The third interface of the Micro-IDE tool is an evaluator designed to compare the generated deployment models by using algorithmic approaches. Also, the tool allows the comparison of the deployment models created manually and algorithmic techniques. As seen in Figure 13, two different models identified as Original and Optimized deployment models can be compared through the selected microservice runtime execution configuration model and the microservice infrastructure model.

In this study, 1361 microservices are deployed to four nodes consisting of different memory capacities to evaluate the deployment performance of the algorithms. Six algorithms are implemented in the Micro-IDE tool: Hungarian, Genetic,[20] Next Fit, Best Fit, Worst Fit, Sequential Distribution, Uniform Distribution, and Minimum Node Distribution algorithms. Each algorithm generates a different deployment alternative using execution cost (relative unit) and communication cost (MB/s) metrics. Communication Cost defines the total communication cost between microservices in the architecture. Execution cost represents the total execution cost of the microservices running on the nodes in the architecture. The results of these two metrics, which differ according to the selected algorithm, are given in Table 2.

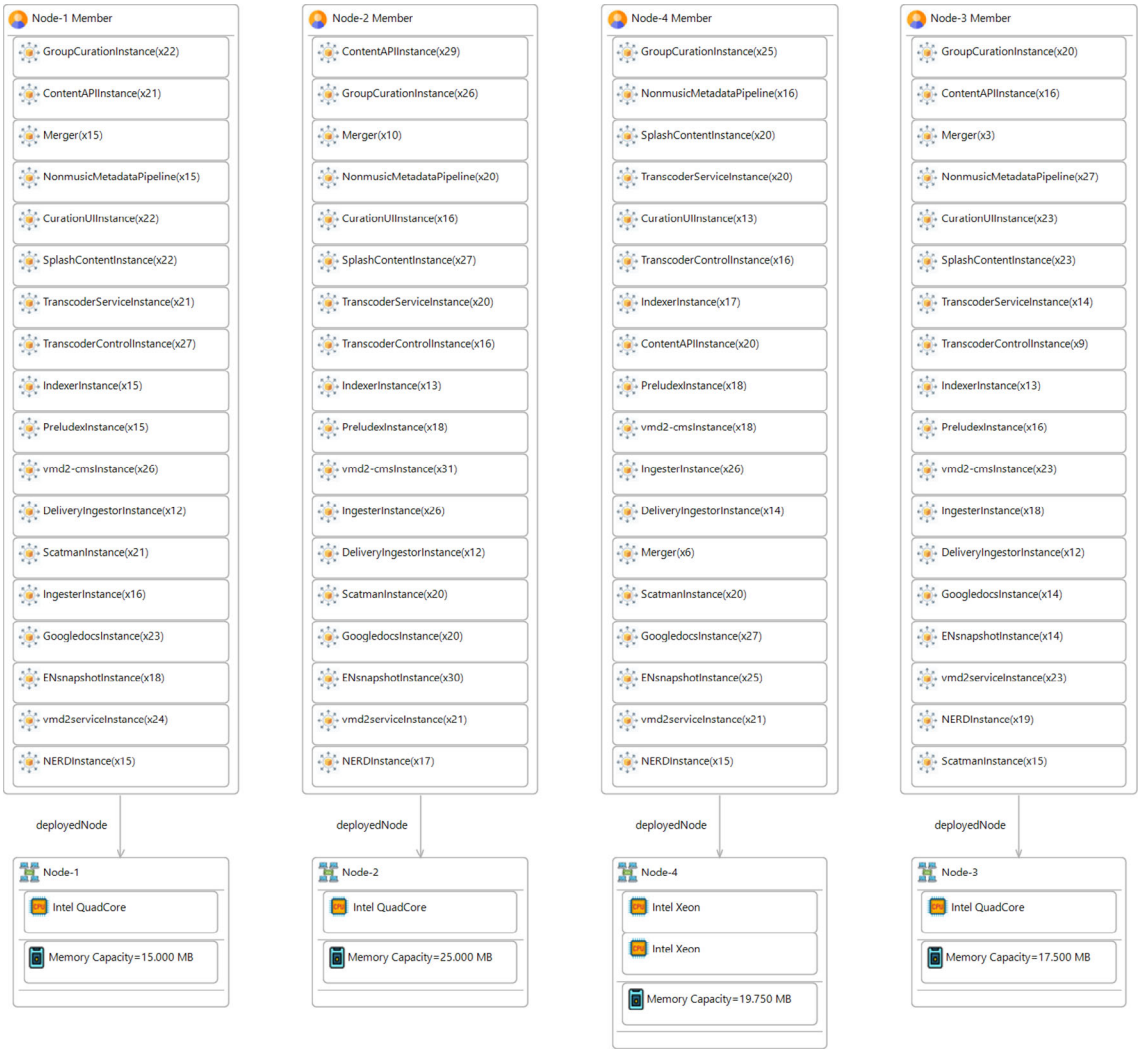


FIGURE 14 A sample deployment model generated manually

In addition to the automated deployment alternatives performed by the algorithms, the proposed tool also allows creating a manual deployment alternative by an expert. Thus, the deployment alternative generated by an algorithm and the deployment model generated by the expert can be compared in terms of performance. Figure 14 shows the manual deployment of 1361 microservices to four nodes. In this figure, the number of service instances is indicated in parenthesis next to the name of microservice instances. Table 2 presents the communication and execution costs for all algorithms implemented in the Micro-IDE tool.

According to the values seen in Table 2, the lowest total communication cost is seen in the Minimum Nodes algorithm. In contrast, this algorithm gives the second-highest value after the Genetic algorithm in total execution cost. The Next Fit algorithm gives the lowest total execution cost among all algorithms. When both metrics are evaluated, the Next Fit algorithm performs relatively better than the other algorithms. On the other hand, Genetic and Minimum Nodes algorithms have the lowest performance in total communication costs. A more detailed analysis of all these values (communication costs for each microservices and their execution costs on each node, the total memory capacity of the services deployed on the nodes) is reported back to the designer through the Micro-IDE tool. Thus, it provides valuable information to the designer for deciding on efficient deployment alternatives.

When the communication and execution costs obtained from the generated deployment models using algorithmic approaches and the manual approach are compared, it is seen that the total communication cost increases in the manual deployment. In terms of execution costs, the manual deployment is more successful than the other algorithms except for Next Fit, Best Fit, and Hungarian. When both metrics are analyzed, Next Fit algorithms highly outperform the manual deployment.

TABLE 3 Performance comparison of algorithms according to manual deployment

Algorithm	Impr. rate of total communication cost (%)	Impr. rate of total execution cost (%)
Hungarian	1.01	1.15
Genetic	0.56	0.33
Next Fit	**27.25**	**12.7**
Best Fit	19.09	− 3.29
Worst Fit	2.75	− 0.94
Sequential Distribution	0.42	− 0.44
Uniform Distribution	6.05	− 0.77
Minimum Nodes	**36.1**	− 0.82

The bold values indicate highest improvement rates according to algorithms.

Two approaches are used when analyzing the validity of the deployment models generated on the Micro-IDE tool. The first approach is to heuristically evaluate deployment alternatives by an expert who knows the architecture very well. In this approach, an expert decides the efficient deployment models through logical reasoning even as the deployment alternatives are generated automatically. The second approach is to compare a deployment alternative with another deployment alternative in terms of communication cost, execution costs, and the total memory capacity of the assigned services in the nodes. As seen in the Deployment Model Evaluator interface (Figure 14) of the Micro-IDE tool, two generated deployment models can be compared, and the comparison results are reported as a .evalresult file

The motivation for calculating the lowest communication cost is deploying two microservices that frequently exchange data on the same node. For instance, ContentAPIInstance subscribes to the data published by DeliveryIngestorInstance in the Spotify example. Similarly, ScatmanInstance and CurationUIInstance services are deployed on the same node. There are many examples in this architecture where data are published by a microservice and subscribed by another microservice. If the connected services are assigned to the same node, the communication cost between them is assumed zero. In this respect, the Minimum Nodes algorithm has the lowest communication cost as it aims to occupy the minimum number of nodes. The performance comparison of the algorithmic approach with manual deployment in terms of improvement rate of communication and execution costs is shown in Table 3.

The algorithmic approaches generally give better results than the manual deployment, as shown in Table 3. The Next Fit algorithm performs the highest improvement rate when both metrics are considered. When only communication cost is analyzed, the Minimum Nodes algorithm shows the best performance with a 36.1% improvement rate. These generators' results may differ according to different microservice runtime execution configuration models and microservice infrastructure models. The proposed tool offers the designer the opportunity to choose the most appropriate alternative. Besides, new CTAP algorithms can be easily added to the Micro-IDE tool for the deployment generation step.

8.2 | Evaluation of deployment model generation time based on the algorithms

In this section, the model generation times to create efficient deployment alternatives with the Micro-IDE tool are analyzed according to different scenarios. The performance of the selected CTAP algorithm significantly affects the deployment model generation time. The genetic algorithm proposed by Mehrabi et al.[22] has the highest complexity among the generators applied to test the tool. The algorithms are implemented in Java, and the program runs on a 64-bit computer with Intel Core I-7-6700HQ CPU 2.60 GHz 16 GB RAM. Table 4 shows the generation times of deployment models for the case study in different services and nodes using the genetic algorithm. The number of services and nodes is determined from an industrial perspective to exhibit a realistic approach. The analysis results show that the genetic algorithm generates deployment alternatives in a shorter time as the number of nodes increases and microservices instances decreases.

When the deployment times of the algorithms are analyzed, it is seen that a small number of services can be deployed in less than one second. As the number of services increases, the time increases in direct proportion. However, the generation times achieved even in deploying many services using the genetic algorithm are acceptable, and the deployments

TABLE 4 Deployment model generation times according to the number of services and nodes using the genetic algorithm

No.	Case study	Total microservice instances	Number of nodes	Generation time (s)
1.	Spotify	18	4	2
2.	Spotify	53	4	2
3.	Spotify	206	4	17
4.	Spotify	274	5	15
5.	Spotify	1361	4	1014
6.	Spotify	1361	10	124

TABLE 5 The deployment times of algorithms according to the same number of microservices and nodes (total number of microservice instances: 1361, the total number of nodes: 4)

Algorithm	Generation time (ms)
Genetic	**1,014,548**
Hungarian	3447
Next Fit	450
Best Fit	414
Worst Fit	415
Sequential Distribution	25
Uniform Distribution	32
Minimum Nodes	20

The bold values indicate highest improvement rates according to algorithms.

yield much faster than a manual deployment. Different CTAP algorithms perform at different generation times, and also, generation time differs according to the implementation of the case study. As the complexity of the communication patterns among microservices increased, the deployment generation times have increased proportionally. In our previous study,[6] generation times for generating sample deployment models for the other case study (Taxi Hailing System) are evaluated according to the different number of microservice instances and nodes using the Genetic algorithm. For the Spotify scenario, the generation times of all algorithms used in the tool have been compared in Table 5 through the case study described in Section 4.2.

Table 5 illustrates that most algorithms can deploy thousands of microservices in less than one second. On the other hand, the generation time of a manual deployment model is unpredictable, and it depends on the expert's system knowledge. But, it is hard to obtain a deployment model manually less than 1 s time.

8.3 | Micro-IDE tool evaluation

The Micro-IDE tool is proposed to generate effective deployment alternatives for microservices at the design phase. The core of the tool is the correct implementation of the algorithms. Hence, we first focused on the evaluation of the algorithms implemented in this tool. On the other hand, the usability of the tool is also an important evaluation metric for practitioners. The tool is build on the Eclipse platform and does not require any additional software. The presented tool is created by using EMF[23] and GMF[24] frameworks as a plugin in Eclipse. The usability of the tool is thus highly related to the usability of these frameworks. Earlier studies[25-27] have discussed the practical usability of this platform. The interfaces presented to the end user in other modeling tools such as Acceleo,[28] JetBrains MPS,[29] Simulink,[30] Sirius[31] are more complicated than the modeling interface created with EMF and GMF frameworks.

The initial conceptual part relates more to the approach than the tool itself. For using the tool, it is required that the UML classes and the relationships between the classes are defined by an expert who is responsible for the design of the created architecture. After this process, the related models can be generated with the same difficulty or simplicity as in the Eclipse platform. Among popular graphical modeling frameworks, Sirius offers a more complex graphical modeling

environment. Although this framework offers a rich graphical design interface, it requires the end user to create all necessary components on the tool. When a designer uses a Sirius platform, he/she needs to dominate five parts (viewpoint, representation, mapping, style, and tool) of the view specification model (VSM) to create the required models for the architecture. Besides, the designer should validate the VSMs by checking correctness of specified representations, mappings, tools, and finding missing or incorrect elements.

We use EMF and GMF as model-driven architecture in this study since GMF offers a simple usage framework to the end user. In addition, we use Emfatic[32] plugin to provide convenience to the designer in model-driven development. Thanks to Emfatic, we can create our model and generate code with a single text-based definition file. The most challenging part of this tool is enhanced code generation during development and working on a more complex infrastructure. However, the effort required by the developer offers the designer a user-friendly and practical environment. In this context, the GMF framework provides an advantage compared to other alternative modeling tools.

9 | DISCUSSION

In this study, we develop a simulation environment using model-driven engineering (MDE) to generate efficient deployment alternatives for microservices at the early design phase. Thanks to the MDE approach, an executable model can be generated using case studies in the design phase. Thus, this approach reduces maintenance costs since it facilitates detecting errors before the coding phase.[33,34,35] MDE provides the designer with an abstract design environment and allows the creation of many models and metamodels using model transformations. The proposed Micro-IDE tool based on MDE generates efficient deployment models at the design phase of the project life cycle. It allows designers to decide efficient deployment models for the architecture by analyzing and evaluating many generated deployment models.

Finding an efficient deployment alternative is a critical problem for MSAs, often consisting of thousands of microservices with different system functionalities. In the industrial area, the management tools such as container technologies used to deploy microservices need a user-created configuration file. Therefore, the configuration file must be reorganized to create a new deployment model. To overcome this problem, we propose a systematic approach[6] to deploy MSAs by creating an automated deployment tool that fulfills functional and quality concerns according to the available resources. The approach is implemented using the EMF and does not require extra hardware and software except the Java platform. In this respect, it differs from the other automated microservice deployment tools. Efficient deployment alternatives can be generated automatically using physical resources (microservice infrastructure model) and runtime execution configuration parameters of microservices (microservice runtime execution configuration model). Different algorithms are applied to generate several alternatives in the toolset, and these algorithms obtain various performances. Besides, the proposed Micro-IDE tool allows the addition of new algorithms, and the new deployment alternatives specific to the algorithm can be offered as long as the input format for the implemented algorithm is correct.

The MSA of the Spotify application[15] is discussed to support the proposed approach with a real-world case study. In the Spotify scenario, 4–10 nodes and 1361 microservices are used for tool testing. In addition to generating an automatic deployment alternative, the tool also enables designing the deployment model manually. In Section 8, the experimental results show that the algorithmic approaches outperform manual deployment in terms of the model generation times and the improvement rates of communication and execution costs. However, this does not mean that there is no need for an expert when deciding and managing the deployment alternatives. The tool is a complementary and supportive alternative for the human expert who can design, manufacture, and evaluate the generated deployment alternatives. After generating automatic deployment alternatives, it is possible to choose closer to the optimum result by the expert's intervention, if necessary.

One of the most important benefits of the Micro-IDE tool is the early analysis of the system, the analysis and evaluation of the generated deployment models during the design phase, and the generation of many deployment alternatives for deciding the most appropriate model for the system. If the deployment process is performed in the development phase, any change in this phase causes returning to the design, implementation, documentation, and test phases in the project life cycle, which leads to increases in management cost and time losses.

To clarify the limitations of the proposed Micro-IDE tool, four possible validity threats named internal validity, construct validity, conclusion validity, and external validity are discussed based on a standard checklist offered by Wohlin.[36]

Internal validity: The most critical validity threat in this study is the inability to reach any expert who knows well the case studies handled in the study. For this reason, the experiments for manual deployment are performed by the first author intuitively.

Construct validity: This type of validation is concerned with assessing the challenges encountered in the data extraction process.[36] To evaluate our tool, all of the microservice types used in real scenarios for the Spotify application case study could not be reached. Another critical point is that it could not be benefited from a real execution scenario while determining microservice instance counts included in the case studies. In this case, the microservice instance counts are intuitively identified by the first author. Besides, all metamodels designed for case studies are created by the author by examining the components on their web platforms. Therefore, we could not scrutinize the usability of the tool support in the industry and the other different types of parameters according to the requirements of industrial companies.

Conclusion validity: When the evaluation results are examined, the deployment model generation times of the algorithms are calculated using the proposed tool, but the model generation time of the manual deployment varies according to the system information of the expert. Our experiments show that the algorithmic approaches used within the scope of the study can generally deploy thousands of microservices in less than a minute. Therefore, we inferred from the experimental analyses that the algorithmic approaches' model generation times are lower than the manual deployment approach.

External validity: This type of validity is concerned with the applicability of the results from the study in a more general context. In the study, specific parameters are used for the deployment of microservices to resources with limited capacity. These parameters are:

- Memory capacities of microservices
- Memory capacities of nodes
- The processing power of nodes
- Estimated executions costs of microservices on each node
- Communication costs between microservices

In this study, the microservice infrastructure metamodel, which includes the characteristics of the nodes, is created by considering cloud computing resources. The bandwidth of the network is assumed as unlimited. Additionally, apart from the parameters used in popular container technologies such as Kubernetes and Docker Swarm, communication costs between microservices, which affect the total cost significantly, are used as a constraint. In future work, the other types of parameters that need to be used as constraints will be investigated, and the adaptation of these parameters to algorithmic approaches will be evaluated by interviewing companies working on MSAs.

10 | RELATED WORK

In the cloud environment, efficient deployment of microservices to limited capacitated resources is a critical issue in terms of efficient use of the cloud resources and high performance of the microservice-based system. Nowadays, the deployment process in microservice-based systems is usually performed by an expert who knows the system very well. However, as the number of microservices in the system increases, it becomes difficult for the expert to deploy hundreds or even thousands of microservices manually. Many automated deployment tools such as Kubernetes, Docker Swarm, Apache Mesos, and other alternative technologies are used for microservices. However, these technologies need a user-created configuration file. However, if a feasible solution cannot be found according to the specified configuration file, the designer must rebuild the configuration file. Besides, the heavy traffic generated in services is difficult to handle by the designer dynamically, and deployment can take considerably longer than an algorithmic approach. This situation prevents the efficient deployment of microservices to limited resources. In this section, literature studies dealing with the efficient deployment of microservices to limited resources and tool-supported system approaches are investigated. Besides, tool-supported studies using a model-driven development approach are presented to examine MSAs during the design phase.

When the tool-supported studies for microservices are examined, it is seen that few studies propose model-driven development for microservices. Among these studies, Granchelli et al.[37] present an MSA recovery tool named MicroART that depends on MDE principles. This tool recovers the system in three stages: Physical Architecture Recovery, Service Discovery Identification, and Logical Architecture Recovery. Since the model-based representation of the MSA is allowed

in this study, the models generated by this tool can be graphically created. It is also stated that the MicroART tool is used for many purposes such as documentation, architectural analysis, architectural reasoning, or verification between deployed architecture and designed architecture. According to the literature studies, MDE is rarely used for microservices. This study uses model-based engineering for MSAs. Still, the proposed MicroART tool only adopts the recovery of a microservices-based system, and it uses a container technology (Docker) for deployment operations.

Similarly, in another study[38] that uses model-driven development, a productive platform is developed for a MSA, including a basic metamodel, a generation platform, and supporting services for workload generation. In the developed tool, Kubernetes container technology has been used for the deployment of microservices. The primary purpose is to measure and evaluate the performance and resilience of MSAs with desired features. While the problems are injected into the microservice environment generated on the tool, the behavior of the services can be evaluated. Thus, the tool enables anomaly detection. In this respect, this model differs from the model-driven automatic deployment tool we proposed.

There are also related studies in the literature that develop tools or platforms by dealing with the deployment problem of microservices. Profeta et al.[39] propose a microservice-based platform for deploying, executing, and composing big data analytics (BDA) applications in different scenarios and areas. The ALIDA platform is designed as a unified platform where both BDA application developers and data analysts interact. Using this platform, developers can register new BDA applications via the exposed API and web user interface. Besides, data analysts can use these BDA applications to create workflows through a dashboard user interface to manipulate and visualize results from one or more sources. It is also stated that the platform provides an optimal deployment configuration by synthesizing a machine learning model based on performance metrics of BDA application workflows. One of the disadvantages of this platform is to be designed for only BDA applications. In this case, different types of microservice-based applications cannot be created using this platform. In another study, Guo et al.[40] propose a new Cloudware PaaS platform based on MSAs and container technologies for deploying services to users through a browser. This platform can deploy traditional desktop applications directly to the cloud and provides the end-user with the opportunity to use the services only through browsers. It is mentioned that this platform has some advantages such as complexity control, individual deployment, flexibility, fault tolerance, and extensibility since it uses MSA. However, it is seen that basic container technologies are used for the deployment of the services, and the platform is designed for using the services directly from the cloud.

In Reference 41, the authors develop a tool called pipekit equipped with instructions determining service dependency by providing a container orchestration language. This tool provides a communication layer to exchange data between microservices using a storage unit for each service. Since the data exchange between different services should be determined in the first configuration to deploy MSAs, the deployment process is performed by notifying the system when the services are ready. The study states that Docker orchestration tools such as Docker Compose do not support complex scenarios sufficiently. Thus the pipekit communication layer tool is designed as an extended version of Docker Compose.

Carvalho et al.[42] propose multiple provider selection approaches for cloud service selection in microservice-based applications. This approach can select several services from a single provider for a microservice and separate providers for each microservice, unlike other techniques. Multiple criteria decision-making method is used to rank the cloud services, and the selection process is mapped with the multi-choice knapsack problem using the greedy algorithm approach. The authors develop a Python-based tool to evaluate the performance of the proposed approach. This tool offers the services to be used and the cost of each provider as a JSON file and the providers to host the microservices. There are no predetermined servers for generating efficient deployment models in the study, and the user is informed about the cost by selecting the multi-cloud provider selection mechanism. In this context, it is foreseen that this information can be used as preliminary information for modeling physical resources in our tool.

Sampaio et al.[1] emphasized that a microservice-based application's performance and resource usage depend on the deployment of the microservices, and the existing deployment tools such as Kubernetes, Docker Compose developed for this purpose have minimal capabilities. Therefore, an adaptation mechanism called REMaP is proposed to generate automatic deployment of microservices. The proposed mechanism can automatically change the service placement at runtime using the relationships (communication costs) and resource usage histories of the services. An MAPE-K-based adaptation manager[43] is used to make this displacement. This proposed mechanism has dealt with the deployment of microservices during a running time and does not evaluate the system during the design phase. For this reason, it differs from the Micro-IDE tool. In another study[2] that deals with optimal and automatic deployment of microservices, an algorithm executed for three stages is presented to minimize the total cost on nodes. In the first stage, the algorithm generates the constraint sequence that determines the deployment of microservices and these services on the nodes. Then,

in the second stage, it generates a set of constraints containing the connections to be established, and finally, it synthesizes the related deployment plan based on these constraints. The optimization metrics are determined that minimize the total cost of the deployment using these constraints. The authors compute a series of deployment plans by modeling a real-world MSA in ABS[44] to test the proposed method. This proposed tool is very close to our approach. Our solution differs from this tool since The Micro-IDE does not need any additional hardware. Besides, many deployment alternatives are presented to the designer by selecting different algorithms at the system's design phase. Therefore, the proposed tool differs from our study since the deployment plan is created using a single algorithmic approach, and it requires the ability to use ABS language for modeling.

11 | CONCLUSION AND FUTURE DIRECTIONS

Adopting MSAs provides many advantages such as scalability, maintenance, flexibility, smaller, and faster deployment, but existing management tools do not consider the runtime information of services. The proposed Micro-IDE tool efficiently performs an automated deployment approach for microservices by using the communication costs among microservices, resource usage of microservices, and their execution costs on the resources. Microservice data exchange, microservice definition, microservice communication, microservice infrastructure, microservice runtime execution configuration, and microservice deployment models were proposed in our previous study[6] are utilized in integration to realize the deployment. The proposed tool allows generated deployment alternatives using different algorithms and enables detailed analysis of the deployment alternatives in terms of execution and communication cost for each service. To evaluate the generated deployment alternatives, the Micro-IDE tool provides interfaces for the designer or end-user. Besides, the tool gives feedback to the designer to further improve the created architecture. We adopted the music and podcasts application called Spotify as a case study to validate the proposed tool. The generation times of the deployment alternatives are less than one second in most algorithms, which is acceptable for the design phase. Besides, it is seen that most of the tested algorithmic approaches generate a deployment alternative with lower communication and execution costs compared to manual deployment.

In future studies, a plug-in will be integrated for one of the container environments such as Kubernetes, Docker Swarm, Apache Mesos, and the deployment configuration generated by an optimization algorithm will be automatically deployed to the server and virtual machines. Since this plug-in will be able to deploy without a user configuration, a significant and unique contribution will be provided to the industrial applications of the microservice concept. This design tool plug-in that automatically deploys microservices to existing management tools will be the first in the literature to the best of our knowledge. Besides, the success of the plug-in and the algorithmic approaches will be studied in an experimental cloud environment, and the validity of the results of the Micro-IDE simulation tool will be analyzed.

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

Data openly available in a public repository that issues datasets with DOIs.

AUTHOR CONTRIBUTIONS

Işıl Karabey Aksakalli implemented the proposed Micro-IDE tool and evaluated the tool in terms of different parameters used for the Capacitated Task Assignment Problem (CTAP). Turgay Çelik pioneered the proposed tool and revised the terminological writing of the article. Ahmet Burak Can contributed to the article by determining the constraints of the tool and the properties of the limited capacitated resources. Bedir Tekinerdoğan proofread and revised this article and greatly improved the article to make the proposed method understandable.

ORCID

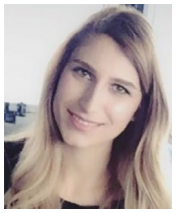
Işıl Karabey Aksakallı https://orcid.org/0000-0002-4156-9098
Turgay Çelik https://orcid.org/0000-0001-9449-2402
Ahmet Burak Can https://orcid.org/0000-0002-0101-6878
Bedir Tekinerdoğan https://orcid.org/0000-0002-8538-7261

REFERENCES

1. Sampaio AR , Rubin J, Beschastnikh I, Rosa NS. Improving microservice-based applications with runtime placement adaptation. *J Internet Serv Appl*. 2019;10(1):1-30.
2. Bravetti M, Giallorenzo S, Mauro J, Talevi I, Zavattaro G. Optimal and automated deployment for microservices. Proceedings of the International Conference on Fundamental Approaches to Software Engineering; 2019:351-368; Springer, Cham.
3. Kubernetes. Kubernetes official website. Accessed July 31, 2020. https://kubernetes.io/
4. Docker. Docker swarm. Accessed July 31, 2020. https://docs.docker.com/engine/swarm/
5. Apache. Docker swarm. Accessed July 31, 2020. http://mesos.apache.org/
6. Aksakalli IK , Celik T, Can AB, Tekinerdogan B. Systematic approach for generation of feasible deployment alternatives for microservices. *IEEE Access*. 2021;9:29505-29529.
7. Kalske M. Transforming monolithic architecture towards microservice architecture; 2018.
8. A. D. Best architecture for an MVP: Monolith, SOA, microservices, or serverless? Accessed July 31, 2020. https://rubygarage.org/blog/monolith-soa-microservices/-serverless
9. Chawla H, Kathuria H. *Building microservices applications on Microsoft azure: designing, Developing, Deploying, and Monitoring*. Apress; 2019.
10. Partanen A. Microservices vs. Service-oriented architecture; 2018.
11. gRPC. Language Guide (proto3). Accessed January 12, 2020. https://developers.google.com//protocol-buffers/docs/proto3
12. Villamizar M, Garcés O, Castro H, et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. Proceedings of the 2015 10th Computing Colombian Conference (10CCC); 2015:583-590; IEEE.
13. SOA vs. Microservices: what's the difference? Accessed January 12, 2020. https://www.ibm.com/cloud/blog/soa-vs-microservices
14. D. L. Ek M. SOA vs. Microservices: what's the difference? Accessed February 17, 2020. https://www.spotify.com/
15. Goldsmith K. Microservices at spotify. Accessed February 17, 2020. https://www.infoq.com/news/2015/12/microservices-spotify/
16. Budinsky F, Ellersick R, Steinberg D, Grose TJ, Merks E. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Professional; 2004.
17. Moore B. Eclipse development: using the graphical editing framework and the eclipse modeling framework; 2004. Books24x7.com.
18. Rubel D, Wren J, Clayberg E. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional; 2011.
19. Voelter M, Kolb B, Efftinge S, Haase A. From front end to code-MDSD in practice; 2006.
20. Daly C. Emfatic language reference. IBM alphaWorks; 2004.
21. Kolovos DS, Rose LM, Abid SB, Paige RF, Polack FA, Botterweck G. Taming EMF and GMF using model transformation. Proceedings of the International Conference on Model Driven Engineering Languages and Systems; 2010:211-225; Springer.
22. Mehrabi A, Mehrabi S, Mehrabi AD. An adaptive genetic algorithm for multiprocessor task assignment problem with limited memory. *Proceedings of the World Congress on Engineering and Computer Science*. 2009;2:115.
23. Eclipse Modeling Framework (EMF). Accessed May 20, 2021. https://www.eclipse.org/modeling/emf/
24. GMF tooling. Accessed May 14, 2021. https://www.eclipse.org/gmf-tooling/
25. Molina F, Toval A. Integrating usability requirements that can be evaluated in design time into model driven engineering of web information systems. *Adv Eng Softw*. 2009;40(12):1306-1317.
26. Saadatmand M, Odontidis V. *Investigating Eclipse Modeling Tools to Improve Usability in SaveIDE*. PhD thesis. Master's thesis. Master of Computer Science Department of Computer Science &Hellip; 2008.
27. Kolovos DS, Paige RF. Towards a modular and flexible human-usable textual syntax for EMF models. Proceedings of the MODELS Workshops; 2018:223-232.
28. Musset J, Juliot É, Lacrampe S, et al. Acceleo user guide; Vol. 2, 2006:157 http://acceleo.org/doc/obeo/en/acceleo-26-user-guide pdf
29. Pech V, Shatalin A, Voelter M. JetBrains MPS as a tool for extending Java. Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools; 2013:165-168.
30. Karris ST. Introduction to Simulink with Engineering Applications. Orchard Publications. 2006.
31. Eclipse Sirius. Accessed May 12, 2021. https://www.eclipse.org/sirius/
32. Eclipse emfatic; Accessed March 20, 2021. https://www.eclipse.org/emfatic/
33. Bézivin J. On the unification power of models. *Softw Syst Model*. 2005;4(2):171-188. Springer.
34. Brown AW, Booch G, Iyengar S, Rumbaugh J, Selic B. An MDA Manifesto The MDA Journal: Model Driven Architecture Straight From The Masters. University of Surrey; 2004.
35. Schmidt DC. Model-driven engineering. *Comput IEEE Comput Soc*. 2006;39(2):25.
36. Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering; 2014:1-10.
37. Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. Microart: a software architecture recovery tool for maintaining microservice-based systems. Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW); 2017:298-302; IEEE.
38. Düllmann TF, vanHoorn A. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion; 2017:171-172; ACM.

39. Profeta D, Masi N, Messina D, Dalle Carbonare D, Bonura S, Morreale V. A novel micro-service based platform for composition, deployment and execution of BDA applications. Proceedings of the 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA); 2019:182-185; IEEE.
40. Guo D, Wang W, Zeng G, Wei Z. Microservices architecture based cloudware deployment platform for service computing. Proceedings of the 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE); 2016:358-363; IEEE.
41. de Guzmán PC, Gorostiaga F, Sánchez C. Pipekit: a deployment tool with advanced scheduling and inter-service communication for multi-tier applications. Proceedings of the 2018 IEEE International Conference on Web Services (ICWS); 2018:379-382; IEEE.
42. Carvalho J, Vieira D, Trinta F. Greedy multi-cloud selection approach to deploy an application based on microservices. Proceedings of the 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2019:93-100; IEEE.
43. Computing A. An architectural blueprint for autonomic computing. *IBM White Pap*. 2006;2006(31):1-6.
44. Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M. ABS: a core language for abstract behavioral specification. Proceedings of the International Symposium on Formal Methods for Components and Objects; 2010:142-164; Springer, New York, NY.

AUTHOR BIOGRAPHIES



Işıl Karabey Aksakallı graduated from Gazi University in 2013 and started to work as a research assistant at Atatürk University in 2014. After receiving her MSc degree from Atatürk University in 2015, she was appointed as a research assistant to Erzurum Technical University. She started her PhD in 2016 at Hacettepe University and received the PhD degree in 2021. Her research topics include microservice architectures, optimization methods, distributed systems, machine learning, and deep learning techniques.

Turgay Çelik received his BS (2003), MSc (2005), and PhD (2013) degrees in Computer Engineering from Hacettepe University, Turkey. From 2003 to 2005, he served as a research assistant at Hacettepe University. Currently, he is a lead software engineer in Bifes Defence & Aerospace Inc., Ankara, Turkey. He has 10 years of professional experience in software engineering research and software development. His research topics 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 is currently affiliated with the Department of Computer Engineering at Hacettepe University, Turkey. He received the PhD degree in Computer Science from Purdue University, West Lafayette. He has BS and MS degrees in Computer Science and Engineering from Hacettepe University. He is a member of the IEEE. His main research areas are computer vision, distributed systems, and network security.

Bedir Tekinerdoğan is a full professor and chair of the Information Technology group at Wageningen University & Research in The Netherlands. He received his MSc degree (1994) and a PhD degree (2000) in Computer Science, both from the University of Twente, The Netherlands. He has more than 25 years of experience in software/systems engineering and is the author of more than 350 peer-reviewed scientific papers. 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.

How to cite this article: Karabey Aksakallı I, Çelik T, Can AB, Tekinerdoğan B. Micro-IDE: A tool platform for generating efficient deployment alternatives based on microservices. *Softw Pract Exper*. 2022;1-27. doi: 10.1002/spe.3088

APPENDIX. THE OVERALL METAMODELS AND THEIR RELATIONSHIPS