Hindawi
Mobile Information Systems
Volume 2020, Article ID 4295937, 23 pages
https://doi.org/10.1155/2020/4295937

Research Article

An Approach for Reconstructing Applications to Develop Container-Based Microservices

Joonseok Park,[1] **Daeho Kim**,[2] **and Keunhyuk Yeom**[3]

[1]*Research Institute of Intelligent Logistics Big Data, Pusan National University, Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 46241, Republic of Korea*
[2]*Korea Aerospace Industries. LTD, 78 Gongdan 1-ro Sannam-myeon, Sacheon-si, Gyeongsangnam-do 52529, Republic of Korea*
[3]*Department of Electrical and Computer Engineering, Pusan National University, Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 46241, Republic of Korea*

Correspondence should be addressed to Keunhyuk Yeom; yeom@pusan.ac.kr

Received 21 October 2019; Accepted 31 December 2019; Published 29 January 2020

Academic Editor: Aniello Minutolo

Microservices are small-scale services that can operate independently. An application consisting of microservice units can be developed independently as a service unit, and it can handle individual logic without being affected by other services. In addition, it is possible to rapidly distribute the configured microservices by a container, and a container orchestration technology that manages the distributed multiple containers can be realized; thus, it is possible to update and distribute the microservices separately. Therefore, many companies are moving away from existing monolithic structures and attempting to switch to microservices. In this paper, we present a method for reconstructing a monolithic application into a container-based microservice unit. The microservices of data units are derived through the collection and analysis of monolithic design data. Furthermore, we propose a method to generate a template script based on deployment design data so that the derived microservice and support distribution can be implemented in a container environment. The results of a case study conducted verified that the container-based microservices deployed in this study work properly. In addition, for the development of monolithic applications and the development of container-based microservices presented in this paper, we confirmed that developing on the basis of microservices is efficient by conducting execution time performance evaluation for API calls at various iterations. Finally, we show that microservices constructed using the proposed method have higher reusability than those constructed using existing methods.

1. Introduction

Serverless computing is an execution model in which applications are developed and distributed based on microservice units without the need to build a separate infrastructure in a cloud environment [1]. Currently, major cloud companies are providing serverless computing technologies, such as Amazon AWS Lambda [2], Microsoft Azure Functions [3], and Google Cloud Functions [4], to their services. By 2018, serverless computing became the fastest-growing cloud service, with a reported growth of approximately 75% between 2017 and 2018 according to RightScale's "2018 State of the Cloud Report" [5]. Forbes magazine has suggested that serverless architecture will be among the top ten technologies of note over the next five years until 2022, and it is emerging as a next-generation cloud computing paradigm [6].

Microservices, which constitute the core concept of serverless computing, are small-scale services that can operate independently. Applications consisting of microservices are loose coupling structures with different microservices; hence, individual updates are possible without knowing the internal structure of applications. Therefore, this structure has the advantage of being rapidly developed and distributed compared with the existing monolithic structure and it is highly reusable [7].

Recently, container orchestration technologies, such as Kubernetes [8], Docker Swarm [9], and Mesosphere [10], have emerged for supporting container distribution management. They are suitable for flexible management of

large-scale microservice-based applications because they enable batch and automatic container creation as well as distribution management of numerous containers.

Because of these advantages of microservices and container orchestration technologies, many companies are attempting to adopt them as alternatives to the existing monolithic application structure in order to develop and distribute microservice-based applications as well as to deploy them on a container basis. However, research on how to reconfigure existing monolithic applications into microservice units remains limited [11].

There are many considerations, such as microservice size, API configuration, database processing, and security, depending on the size of the service when it is intended to reconstruct a complicated monolithic structure into a small and independent microservice [12]. In particular, one of the problems is that it is difficult to change the microservice unit because the microservice size is not fixed. Moreover, there are parameters that need to be manually defined for container-based distribution and management, such as initial container management number and network configuration, even in the case of configuration as a microservice [13]. In addition, converting existing monolithic applications to microservices requires in-depth understanding of the structure of existing monolithic applications. Therefore, methods for deriving microservices based on software architecture [14] or data flows [15] have been proposed. Evolving into an object-oriented development environment, the UML- (Unified Modeling Language-) based application design and development approach is being adopted. Thus, there is also a need to convert existing UML design data into microservices based on the application.

To overcome these problems, this paper presents a method that analyzes the design data of monolithic applications and reconstructs them into container-based microservices. We classify the UML design data of a monolithic application by hierarchy, construct a graph that can be converted into microservices, and derive microservices of an entity unit.

Based on the derived microservices and deployment design data, a method to automatically generate a template script that can be distributed and managed in a container orchestration environment is proposed. In addition, a case study is conducted to verify the validity of the proposed method and an actual reorganized microservice is shown to be deployed and executed in a container orchestration environment. Finally, through comparison and evaluation, the proposed method is shown to be superior to existing methods in terms of reusability.

2. Related Work

2.1. Microservices. Rademacher et al. [16] investigated the distinguishing features of microservices by comparing existing service-oriented architecture (SOA) concepts. They explained that microservices are more independent than SOA and that the size of microservices should be set to units that can be developed by each development team. They also showed that the coupling between microservices is extremely low and that the interface is highly abstracted. In our paper, the concept of interface is used in the construction method, and it is defined to support communication between the user and the microservices.

Dragoni et al. [17] suggested that the microservice configuration should consist of units that carry a single business capability. If the business capability unit is large, it needs to be divided into two or more smaller units, and in the microservices, the database is defined as a distributed-type database rather than being shared in a centralized manner. In our paper, we defined a business capability as an entity that is a database unit.

Yu et al. [18] proposed a reference architecture model for constructing a microservice environment in an enterprise environment. They classified three types of microservices in the proposed architecture model. The interface consists of user input, business logic that processes the input value, and a persistence layer, which contains the data area. We classify microservices into presentation, business logic, and persistence layers using the three-layer concept presented in this study.

2.2. Microservices Construction Methods. Mustafa and Marx Gómez [19] proposed a method for constructing microservices in the runtime environment. In their proposed method, after configuring a similar time zone as one session, sessions are separated when there are numerous accesses in a specific session. However, it was not confirmed whether the services that are accessed frequently clearly perform one business capability.

Mazlami et al. [14] defined a microservice as a class group unit in which changes occur for similar purposes. They constructed a graph in which a vertex corresponds to a class and an edge has a weight according to the degree of similarity of change between classes; then, they applied a clustering algorithm that removes low-weighted edges. However, it was not possible to verify the type of implemented microservices constructed in the related research. Moreover, it was not possible to verify whether the group unit of classes actually performed one business logic.

Chen et al. [15] analyzed data flow diagram (DFD) design data for monolithic applications and defined a group of functions and output data as a single microservice unit. In the existing DFD, they defined a rule for each step and connected the data related to the function that performs one business capability. Furthermore, they constructed the refined DFD data and extracted them into microservices. However, when we applied this method, completely independent microservices were not configured because the data can be associated with other functions.

2.3. Container Orchestration. Docker [20] is a container software platform for rapidly building and deploying applications in container form. It runs on the host OS without being allocated a separate OS or resource, unlike a typical hypervisor-based virtual machine. In addition, independent applications such as virtual machines can run a wide variety of applications. Docker can be deployed independently for each microservice application, and it can drive highly efficient microservice applications.

The main problem with Docker is that it does not have the ability to manage its lifecycle when dealing with a large number of containers. To overcome this problem, a container orchestration concept that supports container management and Docker Swarm [9] was developed. Docker Swarm is an open-source software that supports container orchestration in Docker. The client can manage Docker through Swarm Manager, and Swarm Manager sends a command to Docker daemon to find and assign the appropriate node to create the container.

Kubernetes is an open-source container orchestration platform originally developed by Google. Currently, major cloud vendors, such as AWS, IBM, and Microsoft, are providing cloud services in conjunction with Kubernetes, which is becoming a de facto standard. The structure of Kubernetes can be divided into master and node. The master manages several nodes, and a node creates and executes actual containers. Kubernetes's container management functionality is accomplished through template scripts specified in YAML or JSON format. In Kubernetes, the container unit is a Pod that can be created and executed in the form of a template script. ReplicaSet can control these Pods. The scheduler can manage and maintain multiple Pods using the template script. Finally, there is a service template script that supports network functions, while external network client services, such as LoadBalancing and ServiceDiscovery, can be supported by connecting external client containers.

Kubernetes utilizes a YAML file that defines the desired state for deploying an application—e.g., how many ports it wants to service—by attaching labels to various objects, such as Pods, ReplicaSets, Services, and Volume, and passes it to the API server. In Kubernetes' internal mechanism for creating a new Pod, the ReplicaSet controller included with the Kube controller monitors the ReplicaSet and checks if there is a Pod that satisfies the Label Selector condition defined in the ReplicaSet. A new Pod is created by configuring the Pod template and passing it to the API server.

To distribute constructed microservices in a Kubernetes container environment, we propose a method that automatically generates a Pod, ReplicaSet, and service template script files from monolithic design data.

3. Container-Based Microservice Reconstructing Method

The process of reconstructing microservices by analyzing monolithic design data is outlined in Figure 1. To reconstruct the design data as microservices, we perform the detailed activities in four steps. The explanation for each step is as follows.

3.1. Analyze Monolithic Design Data. In this step, the monolithic design data of the microservice reconstruction target are collected and analyzed. It comprises the following activities:

(i) Collect monolithic design data: the type of design data to be collected is the class diagram of UML design data defined by the ECB pattern [21]. Table 1 summarizes the types and examples of stereotypes in UML design data. The three stereotypes defined in Table 1 are collected.

(ii) Classify 3-tier layer: classes defined as stereotypes are collected from UML class design data and classified in a 3-tier hierarchy. Here, 3-tier implies presentation, business logic, and persistence.

(iii) Map layer by class type: a 3-tier hierarchy is defined, and hierarchical mapping is performed. The stereotypes are as follows: <<Boundary>> type is the presentation layer, <<Control>> type is the business logic layer, and <<Entity>> type is mapping to the persistence layer, as shown in Table 1.

3.2. Extract Microservice. In the microservice extraction step, a graph is constructed on the basis of class information collected and classified in the previous step; then, the following activities are performed to derive the microservices of the entity unit:

(i) Analyze class relationship and construct graph: a graph is constructed consisting of vertex corresponding to a class and edge representing a calling relation.

The vertices classified in the presentation layer according to the mapping relationship shown in Table 1 are defined as boundary vertices (expressed as "bv"). Verices classified in the business logic layer are defined as control vertices (expressed as "cv"). Vertices classified in the persistence hierarchy are defined as entity vertices (expressed as "ev").

In addition, the calling relationship between classes in the UML design data is represented as an edge, as shown in Table 2.

Generalization and realization: in relation to generalization and realization, Class A is a parent class, and it is inherited or implemented. Therefore, Class A influences Class B and calls it.

Dependency and association: in relation to dependency and association, Class A calls and uses Class B's object and method; thus, Class A is affected when Class B is changed.

Composition and aggregation: in relation to composition and aggregation, Class A is included as a part of Class B; hence, if Class B is changed, Class A will be affected.

(ii) Reconstruct business logic-centric graph: the microservice conversion graph is reconstructed so that the configured graph can be derived by the microservice unit. The presentation and persistence layer vertex and edge that are not directly connected to the vertex configured in the business logic layer are removed, and the graph reconfigured.

(iii) Extract microservice based on main entity: in this final activity of the microservice derivation step, a microservice unit having one entity is constructed. Here, entity is a vertex class element classified in the

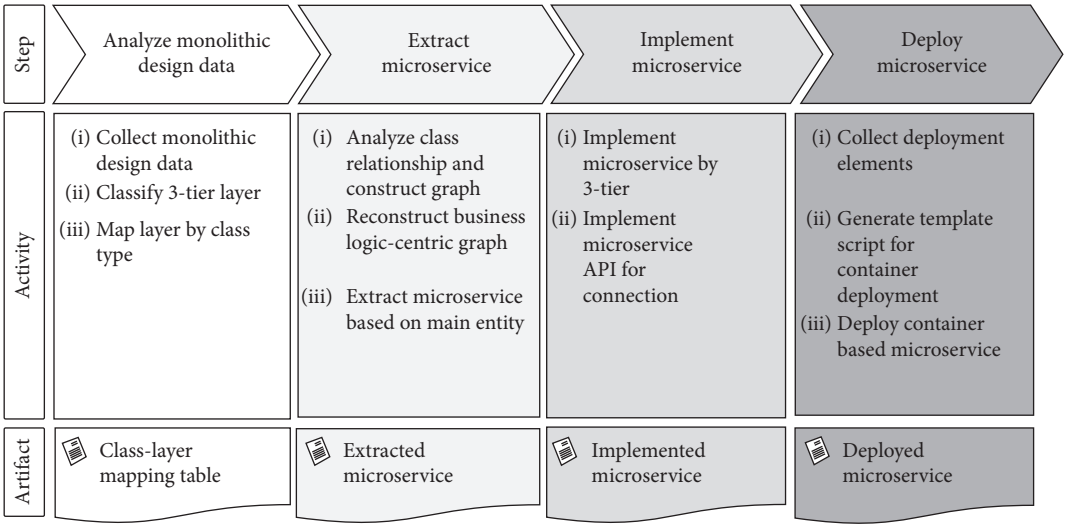
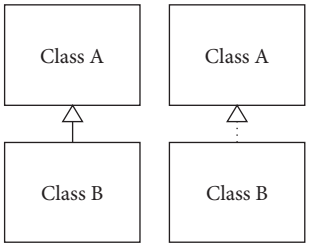
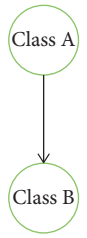
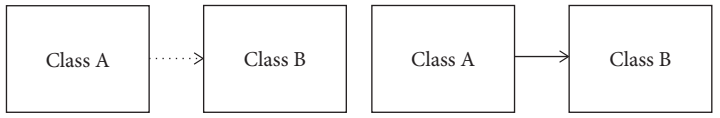
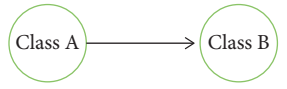
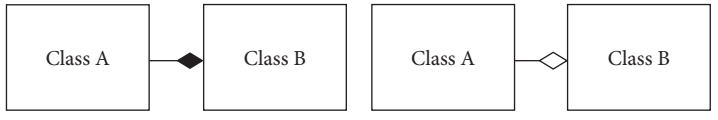
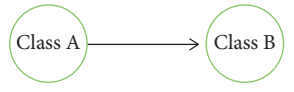


Figure 1: Process of container-based microservice reconstruction.

Table 1: UML design data collection and hierarchy mapping.

Type	Description	Mapping layer by
<<Boundary>>	Class stereotype related to input/output (e.g., main screen UI and billing popup window)	Presentation layer
<<Control>>	Class stereotypes (e.g., payment, reservation, and membership process controller) that realize main function of application	Business logic layer
<<Entity>>	Class stereotypes for accessing and controlling the database (e.g., save member information and register reservation information)	Persistence layer

Table 2: Edge representation according to class relations.

Relationship	UML representation	Edge representation
Generalization and realization		
Dependency and association		
Composition and aggregation		

persistence layer. It traces the vertices of the business logic and presentation layer that are called in the entity and constructs it. However, as a specific control class, which is a vertex of the business logic layer, may be called by many entity classes, the configuration area may overlap. To overcome this problem, a single main entity is determined for the control to resolve the redundancy in the configuration area. Here, one main entity for control is determined by calculating the ratio of the method that calls the entity class and the average value of the method name similarity.

If the main entity calls the other entity directly from the control point of view after constructing the microservice of the entity unit in the control of the determined entity, the design data are changed to be called to the entity through the boundary class of the presentation layer. This is because communication between microservices must be performed by API [22]. This creates a new boundary in the control-entity relationship that is called directly and changes the design data in the control-boundary-entity relationship.

Figure 2 shows the pseudocode of extract microservice based on main entity to derive a microservice in the main entity unit. The input element is graph G, and the output element is the constructed microservice graph. It repeats to determine the main entity of all control classes in the graph. If the control class calls more than one entity, calculate the ratio of the method that calls the entity and the method name similarity of the entity and determine the entity with the higher ratio as the main entity. Otherwise, the class associated with one entity will automatically determine the main entity to be associated with the entity. When the main entity is determined, iterate as many as the number of entities and proceed with the microservice configuration in entity units corresponding to the main entity. Entities for which the main entity is not determined are excluded from the microservice configuration. Finally, when the control is connected to the entity or control class of another microservice, a new boundary is created and the connection is changed to connect the microservice through the created boundary.

3.3. Implement Microservice. In this step, the API connection between the identified microservices is implemented according to the hierarchy:

(i) Implement microservice by 3-tier: implement vertex classes classified in each layer as shown in Table 3. Developers can build on the basis of design data using development tools or languages for each tier.

For example, when the microservices are configured with the following three boundaries, one control, one entity, three APIs, one controller, and one DB are implemented, as shown in Figure 3, for each layer.

(ii) Implement microservice API for connection: the generated boundary is implemented to support API connection between microservices in the microservice derivation step. For example, as shown in Figure 4, when two microservices are connected through a boundary (API CALL 1 and API CALL 2), an implementation for API communication is performed between them. In the case study presented in Section 4, we discuss the implementation process for each layer.

3.4. Deploy Microservice. The final stage of the reconstructing microservice process is microservice deployment, in which a template script is generated that supports deployment and management in a container orchestration environment based on the implemented microservices and UML deployment design data. Here, the design data collect the class and deployment design data, and the microservice refers to the microservice implemented in the Implement Microservice step. The main template scripts are Pod, ReplicaSet, and Service, which are the three main types of template scripts that run on Kubernetes, the container orchestration platform. The process of generating a template script based on the design data and microservice information is shown in Figure 5.

(i) Collect deployment elements: a key template script specification model for deployment and management in the Kubernetes environment, which is a container orchestration environment, is defined as shown in Figure 6. Common info refers to elements that need to be described in all types of template scripts, and Pod, ReplicaSet, and Service are the types of template scripts required to manage container distribution. A description of each type is given in Table 4.

(ii) Generate template script for container deployment: we derive the collection elements from monolithic design data to generate template script specification model elements.

  (1) Microservice information: the microservice information includes information about the name and number of the extracted microservice.
  (2) Class design data: the class information collects the class design name.
  (3) Deployment design data: it is classified into node information and connection information. The node information collects the node name, resource, OS, and application to create the container. The connection information collects the external and internal port, protocol, and external connection IP for communication with the outside of the container.

Table 5 is a mapping table that maps the template script elements to be generated and the collection elements defined in the monolithic design data. In the case of a Pod, a script is generated by mapping elements collected from class information and node information. ReplicaSet is generated by mapping the class design name, the number of microservices configured, and the node information for generating template information. Finally, the service generates a script by

```
Precondition : Need Graph that compose Vertices and Edge with direction
Input : Graph G(E, V)
Output : Graph Microservices

BEGIN
    Loop each Control in Graph G
        IF Control.RelatedCount() >= 2 THEN
            FOR i= 0 to Control.RelatedCount() DO
                CallRate ⟵ Control.CallMethodRate(Entity)
                NameRate ⟵ Control.CompareMethodRate(Entity)
                Avg ⟵ Average(CallRate, NameRate)
                Entity ⟵ SetMaxRateMainEntity(Control, Avg)
            ENDFOR
        ELSE THEN
            Entity ⟵ SetMainEntity(Control)
        ENDIF
    END Loop
    Loop each Entity in Graph G
        IF Entity.hasMainEntity() THEN
            Microservices ⟵ ConstructMicroservice(Control, Entity)
        ENDIF
    END Loop
    Loop each M in Microservices
        IF M.Control.RelateCount() >= 1 THEN
            Microservices ⟵ CreateNewBoundary(M.Source, M.Target)
            Microservices ⟵ CreateNewRelation(M.Source, M.Target, M.Method)
        ENDIF
    END Loop
    RETURN Microservices
End
```

Figure 2: Microservice extraction pseudocode.

Table 3: Implement microservice by layer.

Layer	Implementation method
Presentation layer	API implementation to communicate with the user (or microservice)
Business logic layer	Controller implementation to process information input from boundary
Persistence layer	Implementation of a database that is connected to the controller to process data

mapping the connection information collected from the deployment design data.

A method for collecting UML design data based on a mapping table and generation of a template script is shown in Figure 7. First, the UML design data are converted into a data format (.XMI) such that they can be parsed. Next, elements defined in the mapping table of the second transformed XMI data that are parsed are collected. Finally, based on the extracted collection elements, the template script type is generated.

4. Case Study

In this section, we present a case study based on the process of reconstructing an "online ticket transaction system,"

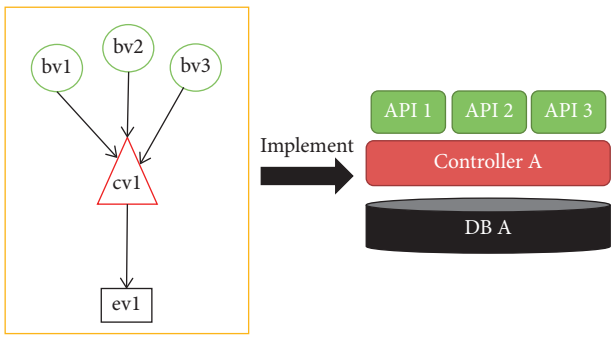


Figure 3: Example of microservice implementation by layer.

which utilizes the actual UML design data, by using the method presented in Section 3. We also verify that the reconfigured microservices operate and run correctly in the container orchestration environment.

The case study "online ticket transaction system" is a system in which sellers and users can register, make inquiries, and sell events (or tickets) online. Figure 8 shows a monolithic application's UML design data of "online ticket transaction system"; we explain the extraction of

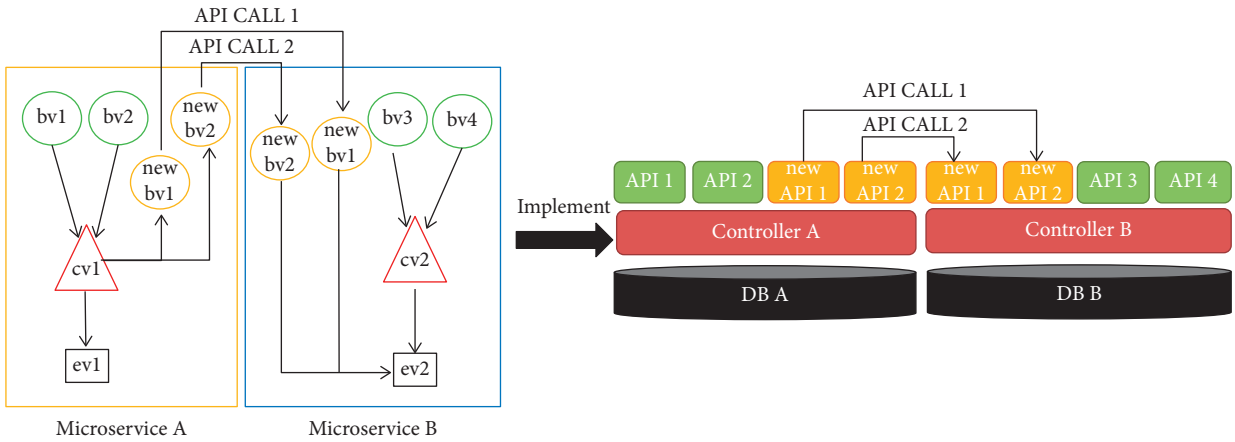


Figure 4: Example of connection API implementation between microservices.

microservices and deployment of container-based microservices in Sections 4.1 and 4.2, respectively.

4.1. Case Study of Monolithic Design Data Analysis and Microservice Extraction. In the first stage, i.e., the monolithic design data analysis phase, UML class design data of the "online ticket transaction system" are collected. As a result of classifying the collected data by hierarchy, 10 boundary classes, eight control classes, and five entity classes were derived. Table 6 summarizes the hierarchical classifications for each stereotype.

Analyze the relationship between classes and construct an edge that points in the direction of the calling method. Figure 9(a) shows G(E, V) consisting of a vertex classified by class type and an edge as a calling relation. In this case, the reconstruct business logic-centric graph method is adopted to reconstruct the microservice conversion graph centered on business logic. As shown in Figure 9(b), as a reconstruction result, we can construct a conversion graph G′(E, V) capable of microservice derivation.

Based on the given microservice conversion graph, the microservice derivation of the main entity unit is performed as shown in Figure 10. The "online ticket transaction system" currently consists of five entities (Member, Report, Event, Ticket, and Payment). To construct the microservice of the entity unit, the configuration is executed by tracing the direction of the call from each entity. As a result, as shown in Figure 10, the microservice configuration areas overlap in ev3, ev4, and ev5.

To address the case in which the microservice configuration area is overlapped, one main entity is determined per control vertex. Figure 11 shows the process of determining the main entity of cv6 among cv6, cv7, and cv8 classes in which microservice overlap occurs. cv6 (TicketViewController) calls ev3 (EventInfo) and ev4 (TicketInfo) methods. Therefore, cv6 must determine the main entity of either ev3 or ev4; it is calculated by applying the method calling and the method name similarity as decision criteria. The rate of the invoking method is calculated as of 25% because ev3 calls one of the four methods that ev3 has. In the case of ev4, one of four methods is called; hence, again, the rate is calculated as 25%. The name similarity is calculated at a rate of 100% because it includes a method called "Ticket" in all the methods of ev4 called "TicketViewController," which is the class name of cv6. In the case of ev3, the rate is calculated as 0% because it does not include "ticket" and "transaction" methods. Finally, the invocation method and name similarity calculation values are averaged, respectively, and hence, ev3 is calculated as 12.5%, ev4 is calculated as 62.5%, and ev4 is determined as the main entity.

As with cv6, the main entity is determined in the same way for cv7 and cv8, which invoke various entities. As a result, ev4 is determined as the main entity of cv7 and ev8 as well as of cv5, and five microservices of entity units are configured without overlapping the configuration area, as shown in Figure 12.

If the microservice configuration area is not duplicated, but the existing control accesses other microservice entities (cv6 ⟶ ev3, cv7 ⟶ ev3, and cv8 ⟶ ev4), it changes the design data to create a new boundary and invokes it in the case of a relationship that directly calls the entity. As shown in Figure 13, in the case of cv6, cv7, and cv8, which are microservice controls that directly call other microservice entities, a new boundary (new bv1, bv2, bv3) is created to change the design data to perform boundary communication between microservices.

4.2. Case Study of Microservice Implementation and Deployment. The reconfigured microservices that are extracted in the previous step are implemented by hierarchy and implement interconnection APIs between microservices. Figure 14 shows the process of implementing member microservices among the microservices configured in the "online ticket transaction system" according to the hierarchy presented in Table 3. In this case study, ExpressJs, which is a package module of Node.js API, is used for API implementation of the presentation layer. The controller of the business logic layer uses node.js on the basis of JavaScript and MySQL as the database of the persistence layer.

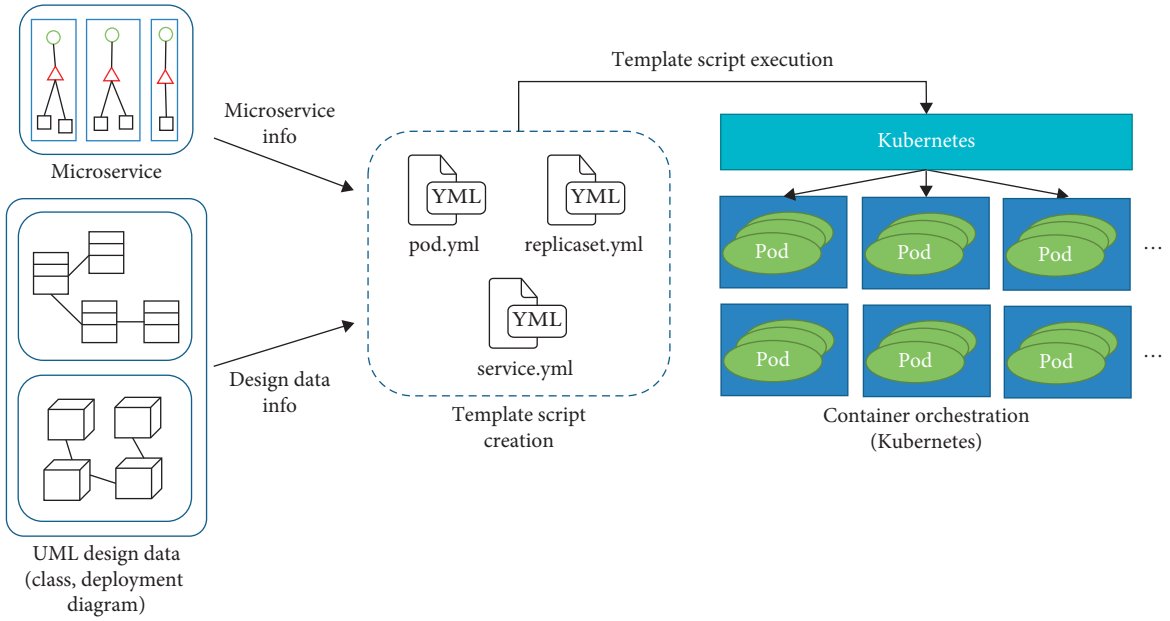


Figure 5: Outline of template script generation.

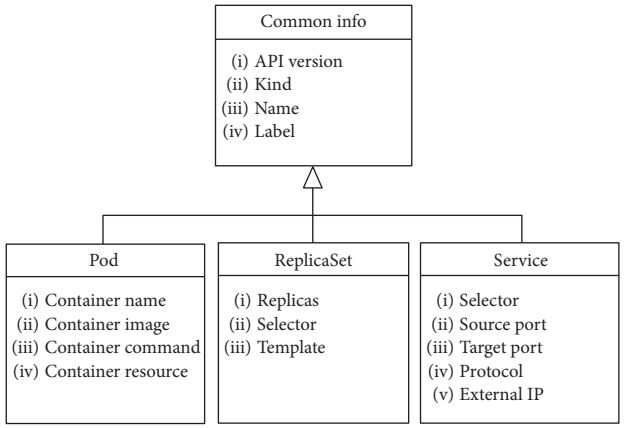


Figure 6: Kubernetes main template script model.

Table 4: Description of Kubernetes main template script model.

Element	Description
Common info	Common info is an element that describes all types of template scripts, and it specifies the script version (apiVersion), the type of template script, the name, and the tag information (label).
Pod	It is a container unit defined by Kubernetes. It should specify at least one container name, container image, container command, and container resources.
ReplicaSet	In Kubernetes, it can control and manage multiple Pods; an administrator can replicate a desired number of Pods, and it can control specific Pods with a label selector. It is possible to create a template script by specifying the template of the Pod container in advance and creating a template script for the same type of Pod through ReplicaSet.
Service	The created Pod service is responsible for network-related services to communicate with external clients. Like ReplicaSet, it can control external communication with a specific Pod through a label selector and specify the source port, target port, protocol information, and external IP.

Figure 15 shows the actual source code of member microservice implemented according to the classified layer. In the case of the API, it is implemented on the basis of the internal input method parameter information of the boundary class. In the case of membership of bv1, the API path is implemented as "/user/signup," and the input parameters are implemented to receive app id, app pass, and name. In the case of the controller, it is implemented on the

Table 5: Template script generation mapping table.

Design information	Collection element	Pod	ReplicaSet	Service
Microservice information	Microservice name Microservice number	name –	 Replicas	
Class information	Class name	Label	Name, selector, label	Name, selector, label
Node information	Node name Node OS Node resource Node application	Container name Container image Container resource Container command	Template (container name) Template (container image) Template (container resource) Template (container command)	
Connection information	External port Internal port Connection protocol External IP			Source port Target port Protocol External

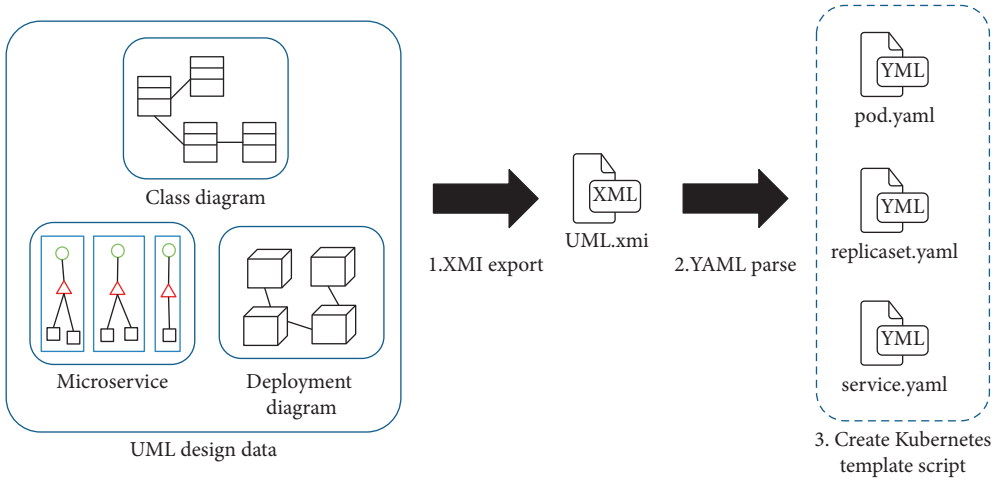


Figure 7: Method to generate template script.

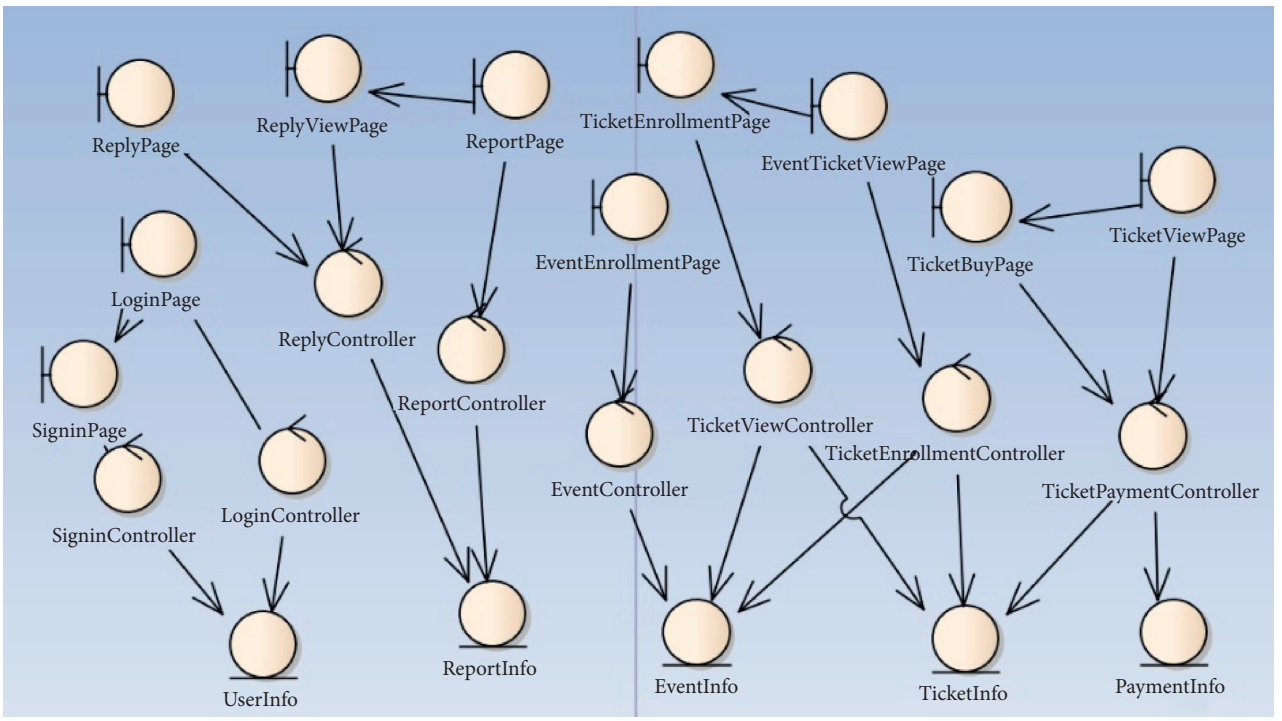


Figure 8: Online ticket transaction system UML class design material.

Table 6: Online ticket transaction system class-hierarchy mapping table.

Presentation layer		Business logic layer		Persistence layer	
SignupPage	bv1	SignupController	cv1	UserInfo	ev1
LoginPage	bv2	LoginController	cv2	ReportInfo	ev2
ReplyPage	bv3	ReplyController	cv3	EventInfo	ev3
ReplyViewPage	bv4	ReportController	cv4	TicketInfo	ev4
ReportPage	bv5	EventEnrollmentPage	cv5	PaymentInfo	ev5
EventEnrollmentPage	bv6	TicketViewController	cv6		
TicketEnrollmentPage	bv7	TicketEnrollmentController	cv7		
EventTicketViewPage	bv8	TicketPaymentController	cv8		
TicketBuyPage	bv9				
TicketViewPage	bv10				

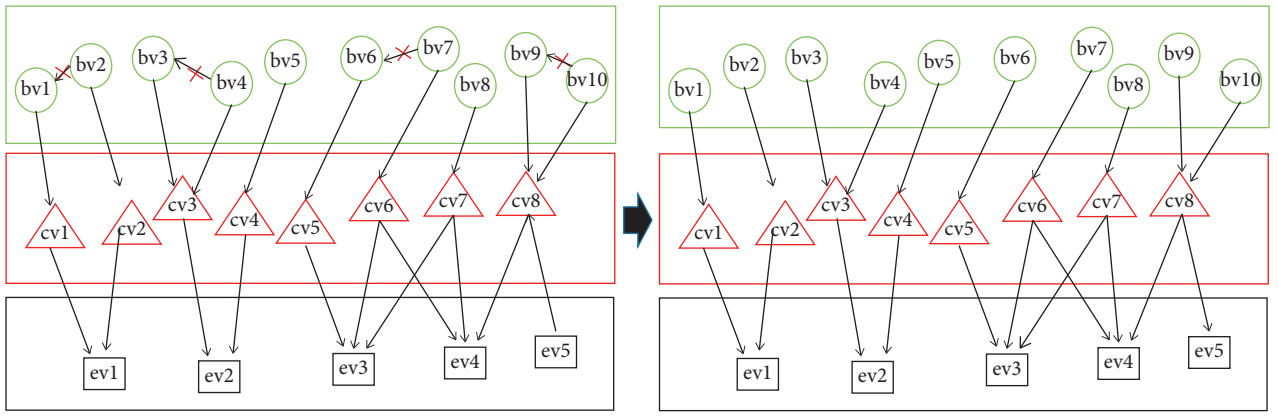


Figure 9: Microservice transition graph of online ticket transaction system. (a) Remove edges that do not connect directly by control class. (b) Converted microservice G′(E, V) reconstruction.

basis of the name and method of the control class. It is created and invoked in conjunction with the API path. Finally, in the case of DB, we implement the table name on the basis of the entity class name and field name as SQL statements based on the method parameter.

After implementation by layers, connection of API is implemented between microservices. The process of implementing a new bv1 among the three new boundaries generated in the microservice extraction step is the same as that shown in Figure 16. The new bv1 is implemented so that cv6 is a boundary containing information to call the EventSelect () method of cv3, and the new bv1 is implemented to invoke EventSelect () with API Path "/event/EventSelect."

After implementing the microservices, a template script is generated to support distribution management in Kubernetes, which is a container orchestration environment. The design data on the left-hand side of Figure 17 are the UML deployment design data of the "online ticket transaction system," consisting of two nodes and one connection information. We export the UML deployment design data to an XMI file for creating the Pod, ReplicaSet, and Service template script according to the method shown in Figure 7.

The right-hand side of the output screen of Figure 17 is the result of exporting design data as XMI data. The nodes and connection information are defined as <element> and <connector> tag value, respectively. Tag internal attributes are defined as name, stereotype, type, and value, where name is the name of the node or connection information, stereotype is the type of the design element (OS, Device, etc.), type is the property of the type (RAM, CPU, etc., for the device), and value is the value for type. For example, DBserver, which is a node name, has a device stereotype, the type of a device is CPU, and the value of RAM is expressed as XMI data with one core and 256 MB specified as the value. The mapping table for the collection element of the transformed XMI data is defined in Table 7 and the XMI data are parsed as shown in Figure 18.

Based on the mapping table of the extracted values, a template script for each type is generated as shown in Figure 19. The area marked with ① denotes the process of creating the name of the Pod template script per microservice name. The area indicated by ② denotes the process of generating the replica value of the ReplicaSet template script by the number of configured microservices. The area marked ③ denotes the process of extracting class diagram information and creating Pod, ReplicaSet, and label of the Service template script. The area marked with ④ denotes the process of extracting the node information of the deployment diagram and creating the container and template information of the Pod and ReplicaSet template script. The area marked with ⑤ denotes the process of extracting connection information and creating the Service template script.

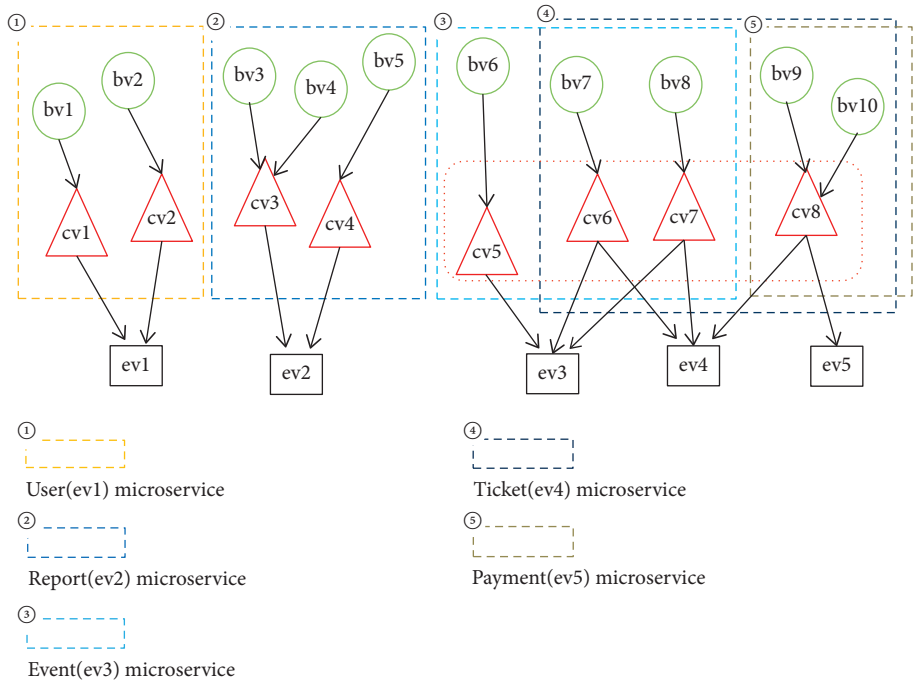


Figure 10: Entity duplication microservice configuration area in entity units.

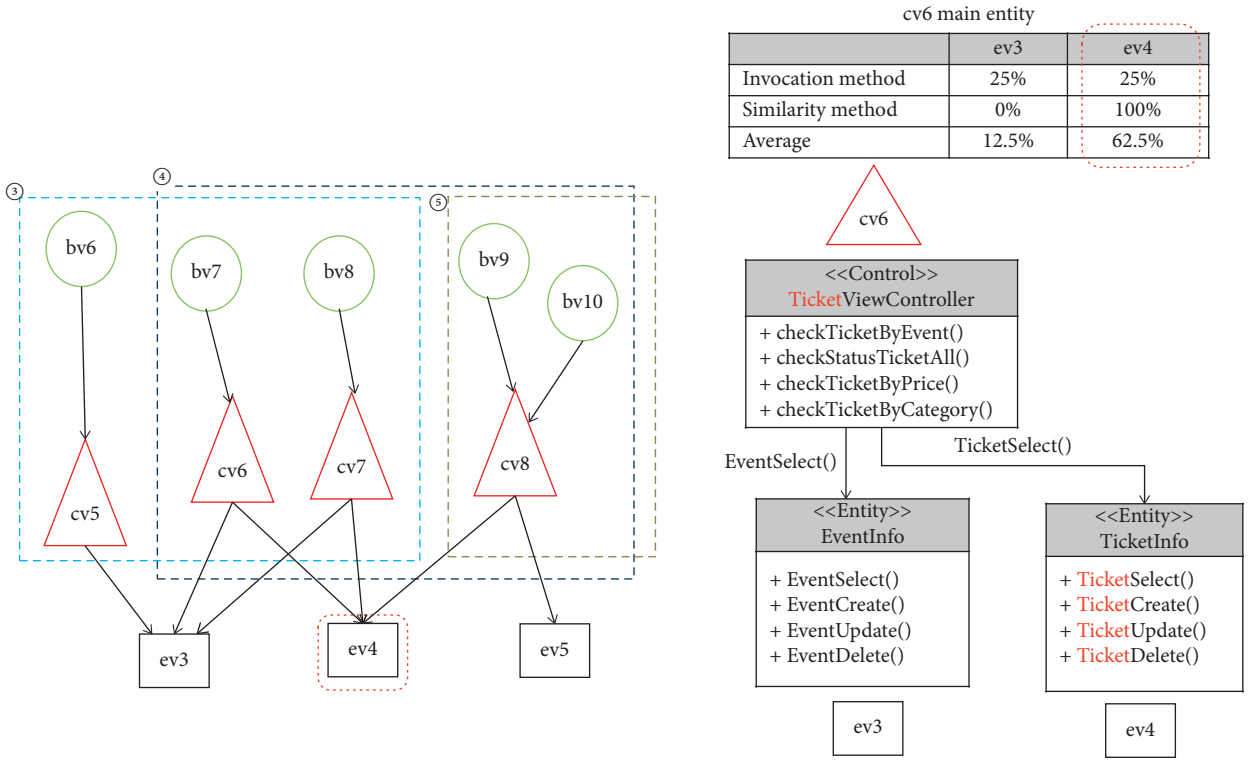


Figure 11: Determining the main entity of cv6.

4.3. Microservice Verification. We verified that the template script file generated in Section 4.2 works correctly. For the verification, Kubernetes, which is a container orchestration environment, was constructed and one master and two nodes were driven into a virtual machine environment. In the case of the master, 2 vCPUs and 4 GB of RAM were allocated, and for

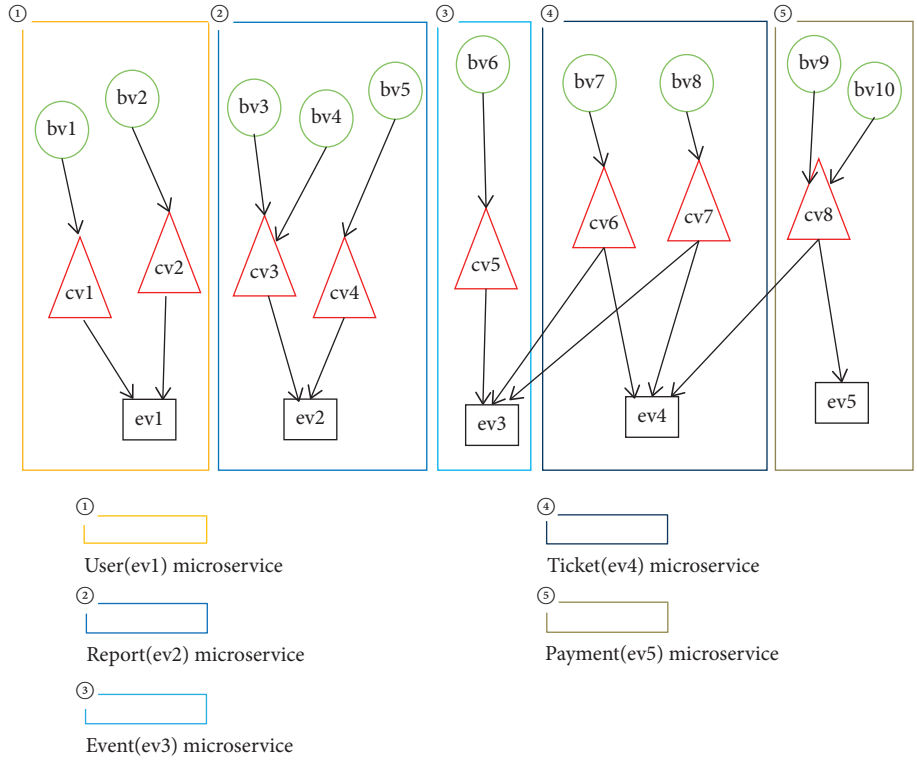


Figure 12: Microservice configuration result in the main entity unit.

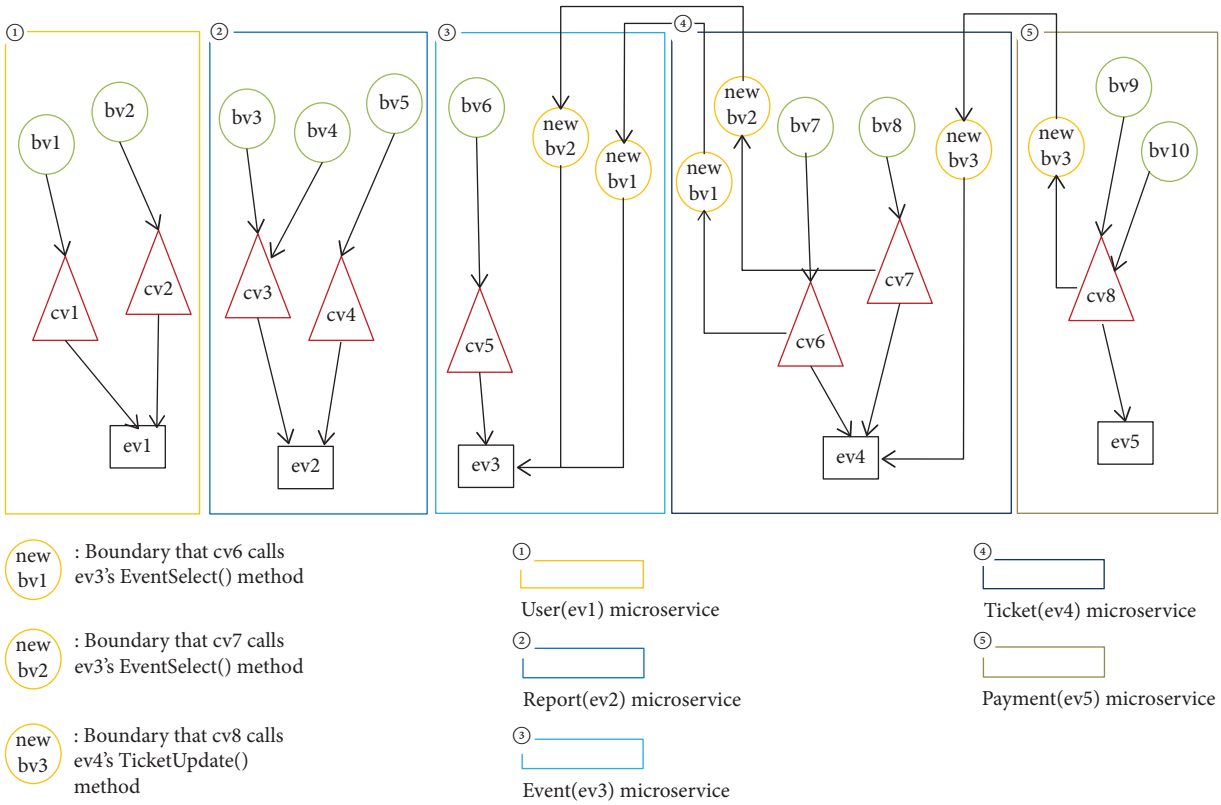


Figure 13: Result of microservice extraction.

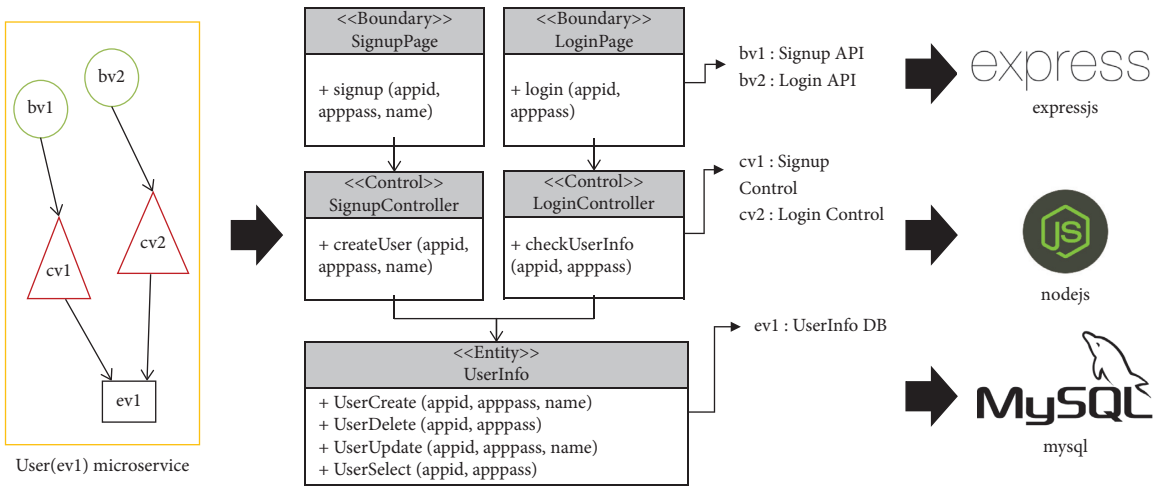


Figure 14: Microservice implementation process by tier.

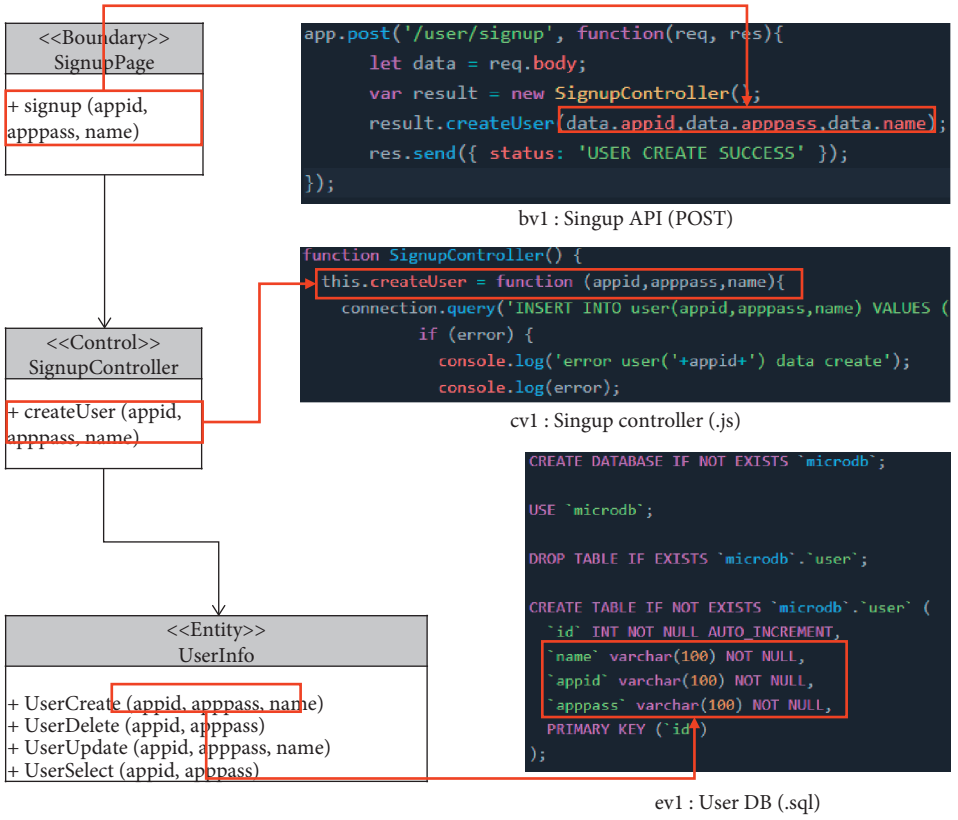


Figure 15: Example of member microservice implementation by layer.

a node, 2 vCPUs and 2 GB of RAM were allocated. The result of executing the three generated types of template scripts in the Kubernetes environment is shown in Figure 20. Here, no. 1 is a screen for inquiring about all the Pod states in the Kubernetes control environment. As a result of executing the generated script, five Pods (userpod, reportpod, ticketpod, eventpod, and paymentpod) were created. Furthermore, it was confirmed that all the Pod states were normal operation. No. 2 is the result of executing the template script of ReplicaSet. Currently, five Pods are managed through labeling and we can confirm that all the Pods are running through the Pod status. No. 3 is the result of executing the service template script. It was confirmed that five Pods by labeling are connected to each endpoint by configuring the protocols, port information, and external IP.

When the implemented microservices were deployed in the Kubernetes environment, we verified that they work correctly. As shown in Figure 21, after implementing the user microservice of the "online ticket transaction system," it was

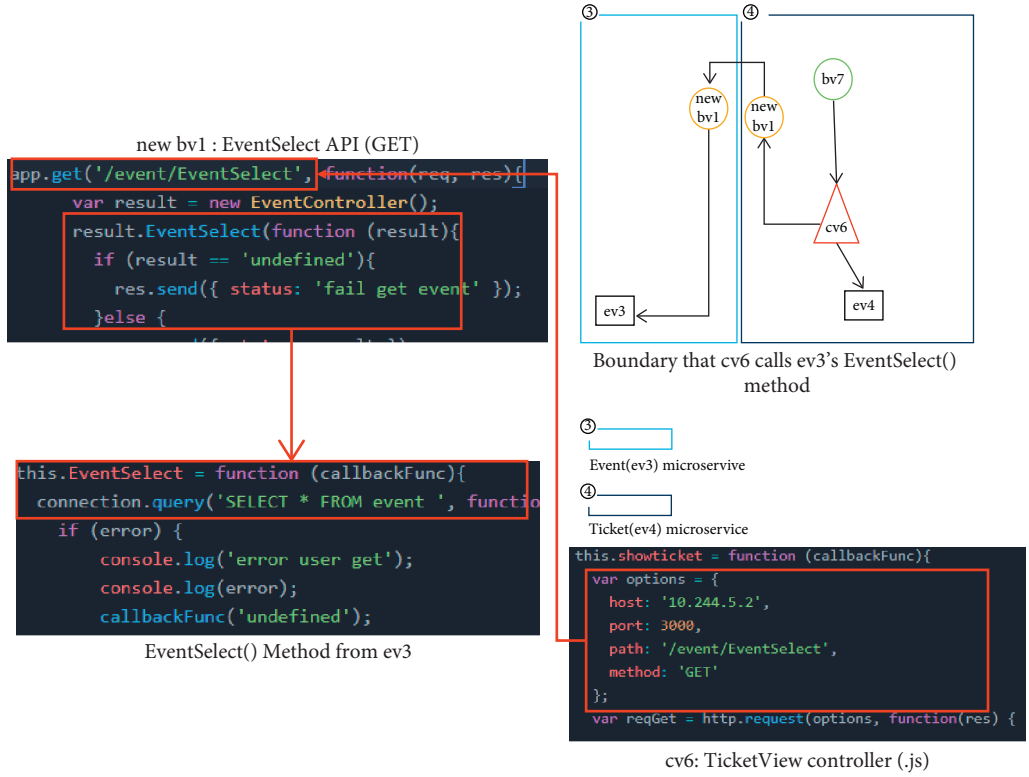


Figure 16: New bv1 implementation and API connection.

distributed to the container environment (userpod). To check whether the distributed member microservice executed, we confirmed the response using Postman [23], an API test tool.

As shown in Figure 22, after the API test, the request was sent to the userpod and it was confirmed that the response was received. The left panel of Figure 22 shows a situation in which a signup API (user/signup) is called from the host information of the userpod 10.244.5.2. To sign up for membership, we sent information (app id, app pass, and name) directly to the user in JSON form and confirmed that the information input from the userpod is processed. The upper right panel of Figure 22 is a command for querying database information in dbserver of userpod before an API call. The bottom right panel is a screen to query database information of dbserver of userpod after an API call. As a result of the API call, it was confirmed that the information input from the database information of dbserver of userpod was generated.

5. Microservice Construction Method Evaluation

We analyzed the distinguishing features and advantages of the presented microservice construction method in comparison with existing related methods and also evaluated them in terms of reusability.

5.1. Comparative Analysis with Related Methods. Table 8 summarizes the evaluation of existing microservice configuration methods and the proposed microservice configuration method according to the stated criteria. The criteria are defined with reference to the microservice configuration standard [16–18]. A microservice is defined to operate independently (independence), microservice interface communication is included (interface), business capability is performed, and independent data elements are included (data store).

Existing related studies have shown that a composed microservice can have an independent configuration, but it cannot be clearly confirmed whether it has a single business logic or whether it can have a single business logic but not be an independent configuration. However, the proposed method performs independent configuration and a single business logic and is classified as 3-Tier. Thus, it can have both an interface element and a database element related to input/output at the same time.

5.2. Performance Evaluation. For performance evaluation, we implemented the monolithic code of the "online ticket transaction system" and the microservices-based code according to the method proposed in this paper, respectively, on the same host, as illustrated in Figure 23.

As shown in Figure 24, the API execution time was measured after application deployment to each environment. We measured 10, 20, 50, 100, 250, 500, and 1000 iterations of each of the implemented monolithic and microservices at 0.1 second intervals to measure the maximum, minimum, and average values of the response times for API calls.

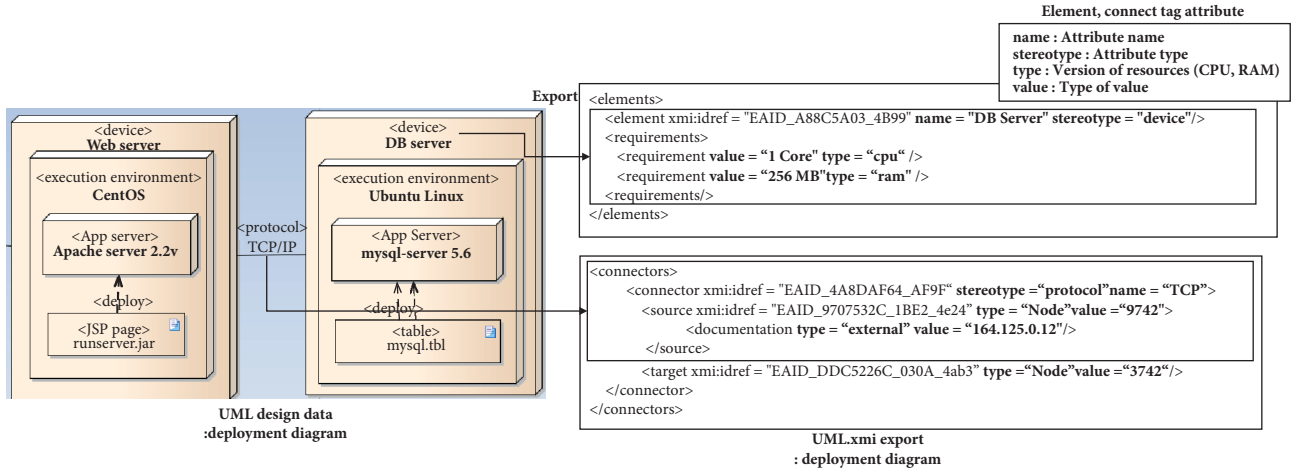


Figure 17: UML design resources export to XMI.

Table 7: Extraction method by extraction target element.

Extraction target element	Extraction method
Node name	Extract the name whose stereotype is "device"
Node resource	Extract name and value whose stereotype is "device"
Node OS	The stereotype extracts the name of the "execution environment" or "OS"
Node application	Extract the name and value whose stereotype is "AppServer" or "application"
External port	Extract the value of the target, that is, the internal tag of connector
Internal port	Extract the value of source, the internal tag of the connector
Connection protocol	Extract the name whose stereotype is "protocol"
External IP	Extract the value whose type is "external"

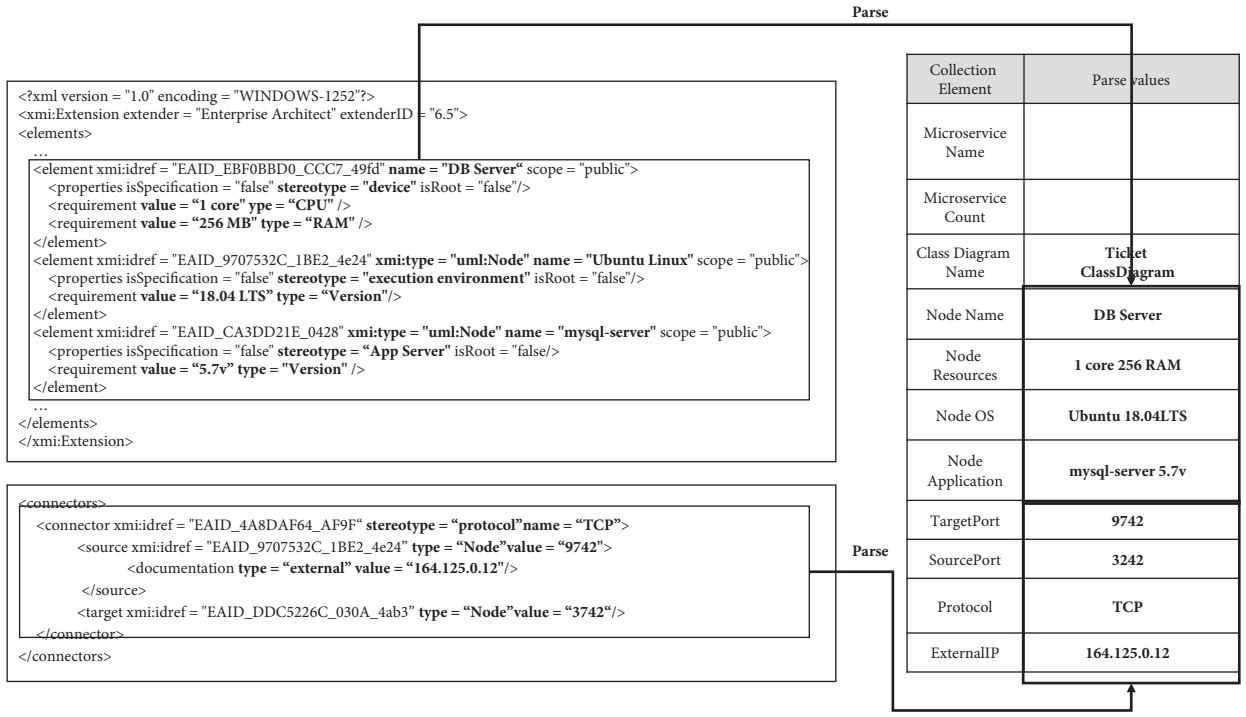


Figure 18: Example of XMI data parsing.

Collection Element	Parse values
Microservice Name	User Microservice Report Microservice Event Microservice Ticket Microservice Payment Microservice
Microservice Count	5
Class Diagram Name	Ticket ClassDiagram
Node Name	WebServer DBServer
Node Resources	WebServer: 1 core / 256 RAM DBServer: 1 core / 256 RAM
Node OS	WebServer: CentOS 6.7v DBServer: Ubuntu 18.04LTS
Node Application	WebServer: nodejsv0.10.48 DBServer: mysql5.7v
TargetPort	3242
SourcePort	9742
Protocol	TCP
ExternalIP	164.125.0.12

```
apiVersion: v1
kind: Pod
metadata:
  name: ticketmicroservice
  labels:
    app: ticketapp
spec:
  containers:
-name: dbserver
  image: ubuntu:18.04
  command: ["/bin/bash", "-c", "--"]
  args: ["while true; do sleep 30; apt-get update; export DEBIAN_FRONTEND=noninteractive; apt-get -q -y install mysql-server; service mysql start; done;"]
  resources:
    requests:
      cpu: "1m"
      memory: "256Mi"
-name: webserver
  image: centos:6.5
  command: ["/bin/bash", "-c", "--"]
  args: ["while true; do sleep 30; yum update; yum install -y curl; curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash; source ~/.bash_profile; nvm install v0.10.48; done;"]
  resources:
    requests:
      cpu: "1m"
      memory: "256Mi"
```

TicketPod.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: ticketreplicaset
  labels:
    app: ticketapp
spec:
  replicas: 5
  selector:
    matchLabels:
      app: ticketapp
    matchExpressions:
      -{key: app, operator: In, values: [ticketapp]}
  template:
    metadata:
      labels:
        app: ticketapp
    spec:
      containers:
-name: dbserver
  image: ubuntu:18.04
  command: ["/bin/bash", "-c", "--"]
  args: ["while true; do sleep 30; apt-get update; export DEBIAN_FRONTEND=noninteractive; apt-get -q -y install mysql-server; service mysql start; done;"]
  resources:
    requests:
      cpu: "1m"
      memory: "256Mi"
-name: webserver
  image: centos:6.7
  command: ["/bin/bash", "-c", "--"]
  args: ["while true; do sleep 30; yum update; yum install -y curl; curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash; source ~/.bash_profile; nvm install v0.10.48; done;"]
  resources:
    requests:
      cpu: "1m"
      memory: "256Mi"
```

TicketReplicaSet.yml

```
apiVersion: v1
kind: Service
metadata:
  name: ticketsvc
  labels:
    app:ticketapp
spec:
  selector:
    app: ticketapp
  ports:
  - protocol: TCP
    port: 3242
    targetPort: 9742
  externalIPs:
  - 164.125.0.12
```

TicketService.yaml

Figure 19: Result of template script generation by type.

```
root@master-VirtualBox:/home/master/Desktop/YAML# kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
eventpod      2/2     Running   0          38s
paymentpod    2/2     Running   0          20s
reportpod     2/2     Running   0          35s
ticketpod     2/2     Running   0          29s
userpod       2/2     Running   0          44s
```

(a)

```
root@master-VirtualBox:/home/master/Desktop/YAML# kubectl describe replicaset ticketreplicaset
Name:         ticketreplicaset
Namespace:    default
Selector:     app=ticketapp,app in (ticketapp)
Labels:       app=ticketapp
Annotations:  <none>
Replicas:     5 current / 5 desired
Pods Status:  5 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

(b)

```
root@master-VirtualBox:/home/master/Desktop/YAML# kubectl describe svc ticketsvc
Name:              ticketsvc
Namespace:         default
Labels:            app=ticketapp
Annotations:       <none>
Selector:          app=ticketapp
Type:              ClusterIP
IP:                10.105.196.91
External IPs:      164.125.0.12
Port:              <unset>  3242/TCP
TargetPort:        9742/TCP
Endpoints:         10.244.1.7:9742,10.244.1.8:9742,10.244.2.7:9742 + 2 more...
```

(c)

Figure 20: Pod (EventPod), ReplicaSet, Service template script generation, and execution result. (a) Confirm that all pods generated by the template script are working normally. (b) Confirm that ReplicaSet created through template script manage 5 pods. (c) Confirm that service generated by the template script contains all of the connection information.

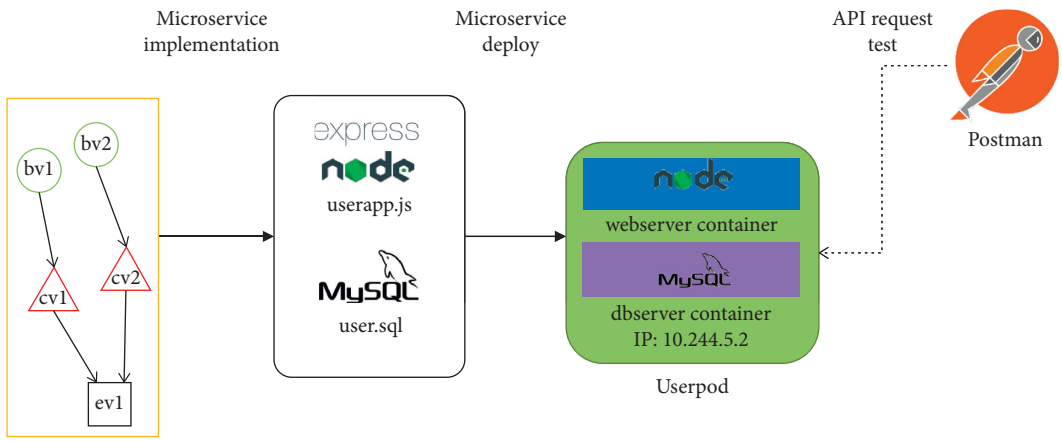


Figure 21: User microservice API call test.

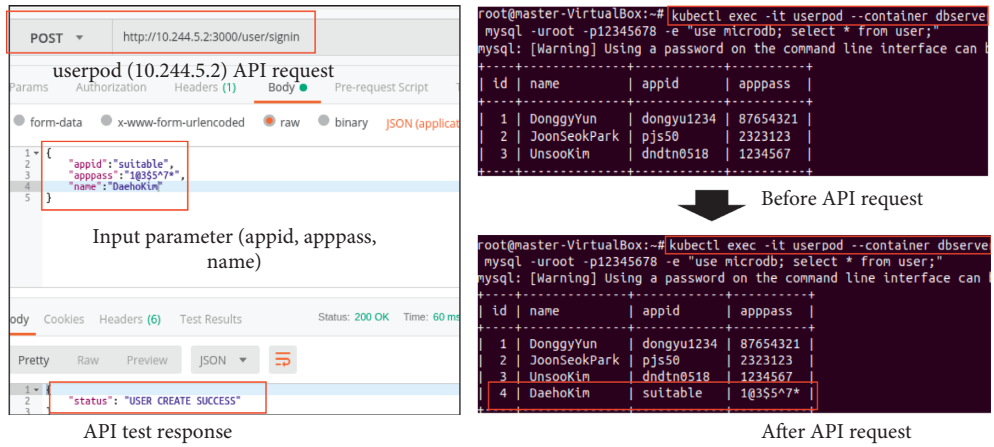


Figure 22: Userpod signup API test result.

Table 9 shows the API call averages for the monolithic and microservice implementations. Less execution time means higher throughput for the application.

As shown in Table 9, when the API call was made, the average API execution of the structure converted to microservices was faster in most iterations than the monolithic structure. Figure 25 compares the total average execution time of the APIs for each iteration.

As shown in Figure 25, in the case of monolithic, it can be seen that the execution time increases rapidly at 500 iterations and beyond. Conversely, in the case of the microservices based on the method presented in this paper, it can be seen that the performance is stable at 500 iterations and 1000 iterations.

5.3. Reusability Evaluation. An application consisting of microservice units can be individually updated without knowing the internal structure of other microservices. This is because microservices have high reusability compared to monolithic structures. We measured the reusability metric of the microservices constructed by applying the proposed method and related methods to the same monolithic application [15]. Cohesion and coupling were measured as the reusability metric; high cohesion and a low coupling value indicate high reusability. Figure 26 shows the result of applying the design data of the "online ticket transaction system" presented in the case study to the DFD-based microservice composition method [15]. The dotted area represents the configured microservices and consists of a total of nine microservice units grouped into one data and process.

The reusability metric measurement uses the method of calculating the degree of cohesion and coupling of design data designed by object-oriented software [24]; the method of calculating the cohesion is the same as that shown in equations (1) and (2). $S(M)$ is the sum of weight for a method M based on M appearances in groups of a subset tree. $R(M)$ is the value obtained by dividing the sample value by the weighted sum of the group of attributes connected with the method; it has a sample value between zero and one. TM is the total number of methods in $R(M)$ Cohesion (X) is calculated by averaging the sampled values of all the $R(M)$ values. In the case of Chen et al. [15], attribute was defined as data, the method was defined as a process that accesses data,

Table 8: Comparative analysis with related research.

Criteria	Mustafa and Marx Gómez [19]	Mazlami et al. [14]	Chen et al. [15]	Proposed microservice construction method
Construction method	Define similar time zones as one session and suggest ways to separate sessions where multiple accesses appear	Clustering of classes with similar purpose of class change	Rule is defined to connect one process with one data and suggests how to configure them as microservices	Suggesting a method for constructing a graph expressing the class relation in the monolithic design data and then constructing it as an entity unit.
Independence	Independently separated by session unit	Classes with similar purpose of change are linked independently	It can be associated with other data and cannot be an independent configuration	Independent connections between classes affecting entity unit
Interface (API)	Separated services include interfaces that support user access.	Not included	Not included	Constructed microservices perform API communication between user input/output and microservice
Business capability	It is not known whether a separate service explicitly performs one business logic because it isolates the frequently occurring services independently	Class groups with similar purpose for change do not verify that they perform one business logic	Define through rule that a group of operation and data should be subdivided into one business logic	The constructed microservice defines the business logic to be executed in one entity unit.
Data store	Not included	Not included	Data associated with operation are presented as a microservice construction element	Constructed microservices include entity elements for processing data

and the degree of cohesion was measured. In our approach, we define attribute as entity and method as boundary. We also define control that accesses entity and measures the cohesion.

$$R(M) = \frac{S(M)}{\text{Maximum of } S(M) \text{ for all methods in a subset tree}},$$

$$\text{Calculation of connection weight values between methods: } R(M), \tag{1}$$

$$\text{Cohesion}(X) = \frac{\sum R(M)}{TM}, \tag{2}$$

$$\text{Class cohesion calculation: cohesion}(X).$$

Table 10 shows the result of cohesion calculation of microservices constructed through each configuration method. In the case of Chen et al. [15], nine groups of microservices were constructed. In the case of the proposed method, five groups were constructed. The relation and group weight values associated with the configured group were calculated.

The individual $R(M)$ values, which are the result of dividing the maximum value of the sum of the group weights in the groups by relation, are shown in Table 11. The average value of $R(M)$ of the microservice constructed through each configuration method was calculated. The degree of cohesion of the microservice constructed in [15] was 0.694, whereas that of the microservice constructed by the proposed method was calculated to be 0.823.

Equation (3) shows the calculation method for measuring the degree of coupling between classes. A, D, and I represent the number of classes of association, dependency, and inheritance relationships, respectively. The higher the relationship, the higher is the degree of coupling between classes. Further, a, b, and c are nondeterministic constants, which are to be considered by the developer or designer. In the case of monolithic design data, the constant a is 1 for direct connection and 0.5 for indirect connection. In the case of Chen et al. [15], the degree of coupling between data processes was measured. In the case of microservices constructed using the proposed method, the degree of coupling between entities and controls was measured.

$$a = \begin{cases} 1, & \text{if calling directly from a class,} \\ 0.5, & \text{if calling indirectly from a class,} \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

$$\text{Coupling(class)} = aA_c + bD_c + cI_c,$$

$$\text{Class coupling calculation: coupling}(X).$$

The sum of the combined measurements of the design Chen et al. [15] was calculated as 13.5, as stated in

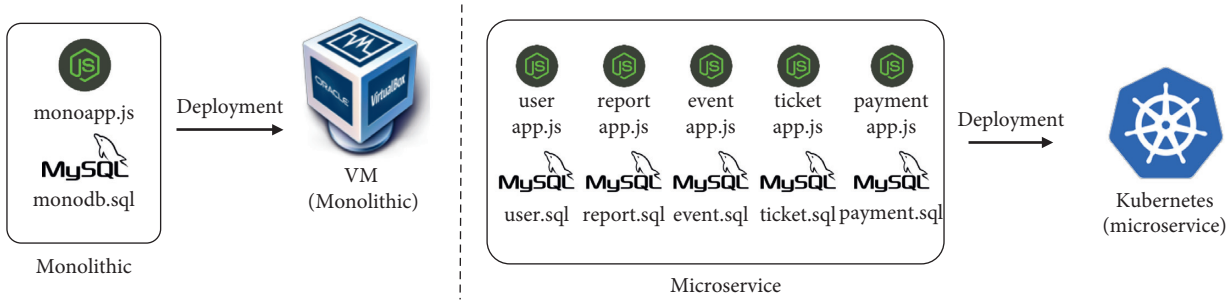


Figure 23: Deployment of implemented application.

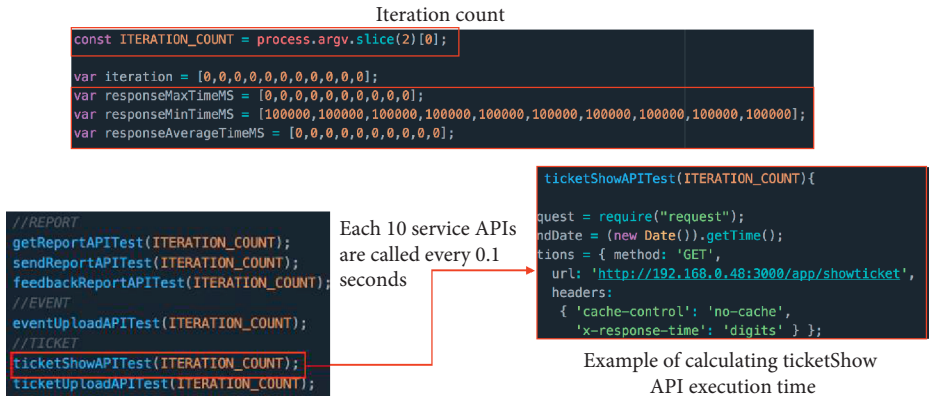


Figure 24: API execution time measurement.

Table 9: Average API execution time comparison (unit: milliseconds).

Type	API	Iteration						
		10	20	50	100	250	500	1000
Monolithic	LOGIN API	32.40	27.90	24.24	24.03	21.72	93.35	353.82
	SIGNUP API	34.30	30.35	27.62	26.56	24.29	96.04	356.36
	SHOW REPORT API	35.20	31.05	28.22	27.79	26.45	93.64	356.36
	SEND REPORT API	23.40	20.05	18.40	17.19	14.88	18.97	16.07
	FEEDBACK API	24.30	20.50	19.28	17.67	15.46	19.40	16.55
	UPLOAD EVENT API	25.50	21.55	19.98	18.28	16.23	19.88	17.00
	GET TICKET EVENT API	44.00	38.40	38.92	41.20	52.79	221.04	794.89
	UPLOAD TICKET API	25.90	21.70	20.56	18.81	16.77	21.34	17.58
	BUY TICKET API	26.20	22.25	21.36	19.52	17.66	21.03	18.33
	GET PAYMENT API	41.40	36.35	28.22	36.32	36.59	109.82	370.17
Microservice	LOGIN API	18.00	13.40	15.20	12.38	11.38	11.16	10.20
	SIGNUP API	18.90	13.35	16.28	13.18	11.97	11.76	10.89
	SHOW REPORT API	20.30	15.30	18.12	15.38	14.82	39.83	14.69
	SEND REPORT API	18.30	12.90	14.92	12.72	11.29	11.45	10.46
	FEEDBACK API	18.80	14.05	16.10	13.43	12.08	12.40	11.22
	UPLOAD EVENT API	18.50	14.25	15.92	12.91	11.34	11.65	10.38
	GET TICKET EVENT API	26.60	21.30	24.42	21.78	29.33	56.63	54.34
	UPLOAD TICKET API	17.60	14.25	15.92	12.91	11.64	12.07	10.91
	BUY TICKET API	18.00	13.20	14.12	12.65	11.26	11.92	10.77
	GET PAYMENT API	18.10	14.90	17.38	14.68	14.28	14.82	14.61

Table 12. On the other hand, in the case of the microservice design data constructed using the proposed method, as the direct connection relation of ev3-ev6, ev3-cv7, and ev4-ev8 is changed to an indirect connection by the new boundary, it was calculated as 9.5, as stated in Table 13.

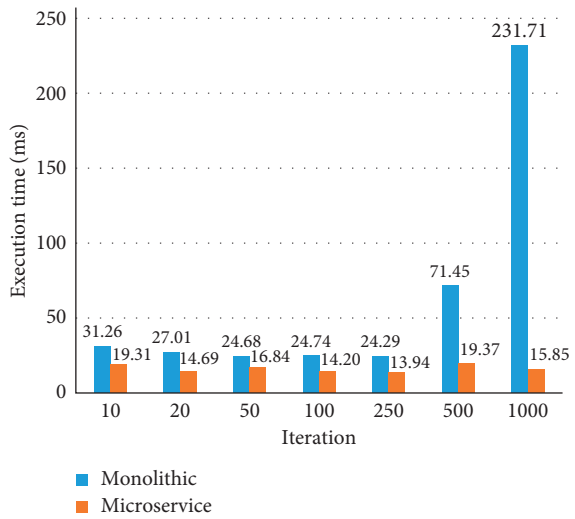


Figure 25: Average API execution time comparison per iteration.

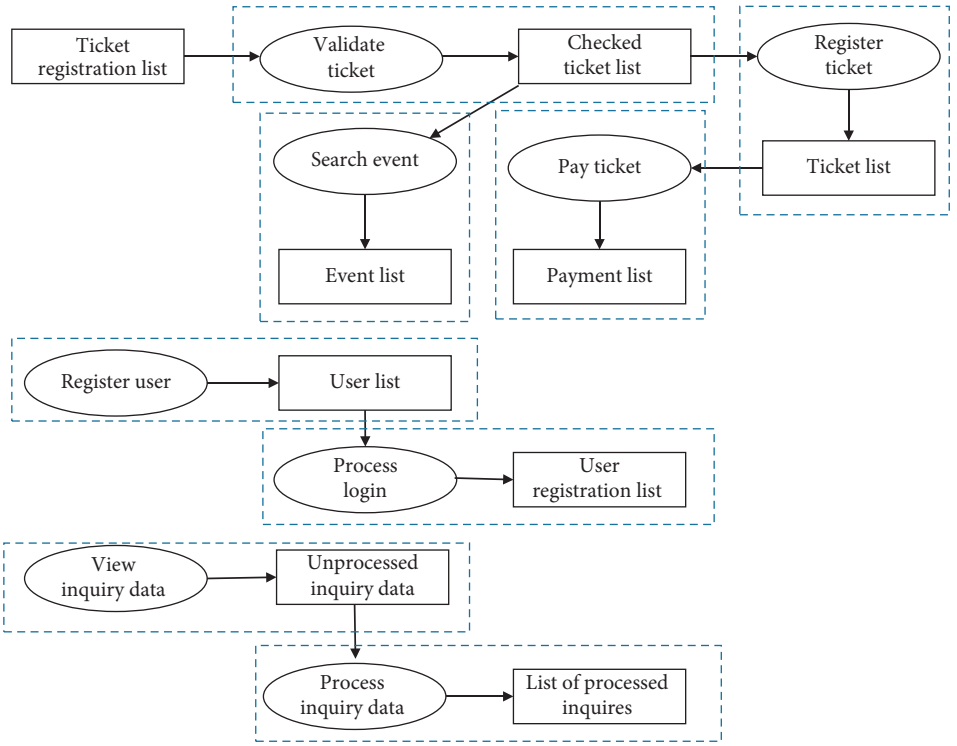


Figure 26: Microservices through the method presented by Chen et al. [15].

Table 10: Group weight calculation.

	Group	Relation	Count	Group weight
Microservice construction method in Chen et al. [15]	Group1 (D1)	P1, P2	2	0.222
	Group2 (D2)	P2, P3, P4	3	0.333
	Group3 (D3)	P2, P4	2	0.222
	Group4 (D4)	P2	1	0.111
	Group5 (D5)	P5	1	0.111
	Group6 (D6)	P5, P6	2	0.222
	Group7 (D7)	P7	1	0.111
	Group8 (D8)	P7, P8	2	0.222
	Group9 (D9)	P7, P8, P9	3	0.333
Proposed microservice construction method	Group1 (ev1)	cv1, cv2, bv1, bv2	4	0.19
	Group2 (ev2)	cv3, cv4, bv3, bv4, bv5	5	0.238
	Group3 (ev3)	cv5, bv6, new1, new2	4	0.19
	Group4 (ev4)	cv6, cv7, bv7, bv8, new1, new2, new3	7	0.333
	Group5 (ev5)	cv8, bv9, bv10, new3	4	0.19

Table 11: $R(M)$ calculation.

Microservice construction method in [15]		Our proposed microservice construction method	
Relation	$R(M)$	Relation	$R(M)$
P1	0.25	cv1	1
P2	1	cv2	1
P3	0.375	cv3	1
P4	0.625	cv4	1
P5	1	cv5	0.364
P6	0.667	cv6	0.364
P7	1	cv7	0.636
P8	0.833	cv8	0.364
P9	0.5	bv1	1
		bv2	1
		bv3	1
		bv4	1
		bv5	0.364
		bv6	0.364
		bv7	0.636
		bv8	0.636
		bv9	0.364
		bv10	0.364
		new1	1
		new2	1
		new3	1

Table 12: Chen et al. [15] design data coupling calculation.

		$P1$	$P2$	$P3$	$P4$	$P5$	$P6$	$P7$	$P8$	$P9$	Coupling (C)
Microservice construction method in Chen et al. [15]	D1	1	0.5	0	0	0	0	0	0	0	1.5
	D2	0	0.5	1	0.5	0	0	0	0	0	2
	D3	0	0.5	1	0.5	0	0	0	0	0	2
	D4	0	1	0	0	0	0	0	0	0	1
	D5	0	0	0	0	1	0	0	0	0	1
	D6	0	0	0	0	0.5	1	0	0	0	1.5
	D7	0	0	0	0	0	0	1	0	0	1
	D8	0	0	0	0	0	0	0.5	1	0	1.5
	D9	0	0	0	0	0	0	0.5	0.5	1	2

Table 13: Coupling calculation for the proposed construction method.

		cv1	cv2	cv3	cv4	cv5	cv6	cv7	cv8	Coupling (C)
Proposed microservice construction method	ev1	1	1	0	0	0	0	0	0	2
	ev2	0	0	1	1	0	0	0	0	2
	ev3	0	0	0	0	1	0.5	0.5	0	2
	ev4	0	0	0	0	0	1	1	0.5	2.5
	ev5	0	0	0	0	0	0	0	1	1

Table 14: Comparison and evaluation of reusability with respect to Chen et al.

Criteria	Chen et al. [15]	Proposed microservice construction method
Microservice construction unit	Single process and data	Class unit that invokes entity
Number of constructed microservices	9	5
Cohesion	0.694	0.823
Coupling	13.5	9.5

Therefore, we measured the cohesion and the coupling, which are the reusability factors of the microservices constructed by the related research method [15] and the proposed method. Table 14 shows that the proposed microservice configuration method has high cohesion and low coupling and is highly reusable.

6. Conclusion

This paper presented a method for analyzing monolithic application design data and constructing them as container-based microservice units. The proposed methodology involves processes of monolithic design data analysis, microservice extraction, microservice implementation, and microservice deployment. In the monolithic design data analysis stage, monolithic design data were collected and classified into different types. In the step of microservice extraction, a graph was constructed by analyzing classes and associations; then, an entity unit microservice was derived. In the microservice implementation stage, the implementation of microservices was conducted by layer. Finally, in the microservice deployment phase, we created a deployment support script that can gather deployment environment elements and deploy microservices in a container environment based on the collection elements.

In addition, we conducted a case study on each step of the proposed microservice construction method. Finally, we confirmed that the "online ticket transaction system," which is a monolithic application, was configured as a microservice unit and distributed and operated on a container basis. Moreover, by comparing the proposed method with related research and evaluating the reusability of the constructed microservices, we confirmed that the microservices developed in this study are superior in terms of reusability.

If the proposed microservice construction method is applied to an existing monolithic application, it can be configured as a container-based microservice unit at low cost. The advantage is that, in the initial container environment, the user need not manually generate a template script; he or she can generate a template script through the mapping table. Therefore, further development of the microservice unit in the serverless computing environment is expected in the future.

In particular, future research will focus on developing a monitoring tool to manage efficient container operation after distributing microservices constructed by the proposed method and on improving the performance of API Gateway to solve the network overhead issue between microservices.

Data Availability

The data used to support the findings of this study are available from the authors upon reasonable request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) and the Ministry of Education (no. NRF-2017R1D1A1B03030243).

References

[1] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.
[2] Amazon AWS Lambda, https://aws.amazon.com/lambda/?nc1=f_ls.
[3] MS Azure Function, https://azure.microsoft.com/services/functions.
[4] Google Cloud Function, https://cloud.google.com/functions.
[5] 2018 State of the Cloud Report, 2018, https://www.rightscale.com/lp/state-of-the-cloud.
[6] Forbes, https://www.forbes.com/sites/louiscolumbus/2017/08/15/gartners-hype-cycle-for-emerging-technologies-2017-adds-5g-and-deep-learning-for-first-time/#272197a05043.
[7] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *Proceedings of the 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, pp. 8–13, Buenos Aires, Argentina, May 2017.
[8] Kubernetes, https://kubernetes.io/.
[9] Docker Swarm, https://docs.docker.com/engine/swarm/swarm-tutorial/.
[10] Mesosphere, https://mesosphere.com/.
[11] I. Baldini, P. Castro, K. Chang et al., "Serverless computing: current trends and open problems," in *Research Advances in Cloud Computing*, pp. 1–20, Springer, Berlin, Germany, 2017.
[12] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.
[13] E. Casalicchio, "Autonomic orchestration of containers: problem definition and research challenges," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, Taormina, Italy, October 2016.
[14] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architecture," *IEEE International Conference on Web Services*, vol. 40, no. 11, pp. 524–531, 2017.
[15] R. Chen, S. Li, and Z. Li, "From monolith to microservices: a dataflow-driven approach," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475, Nanjing, China, December 2017.
[16] F. Rademacher, S. Sachweh, and A. Zündorf, "Differences between model-driven development of service-oriented and microservice architecture," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 38–45, Gothenburg, Sweden, April 2017.
[17] N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, pp. 195–216, Springer, Berlin, Germany, 2017.
[18] Y. Yu, H. Silveira, and M. Sundaram, "A microservice based reference architecture model in the context of enterprise architecture," in *Proceedings of the IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference*, pp. 1856–1860, Xi'an, China, October 2016.
[19] O. Mustafa and J. Marx Gómez, "Optimizing economics of microservices by planning for granularity level," in *Proceedings of the ProWeb 2017 Programming Technology for the Future Web*, Brussels Belgium, April 2017.
[20] Docker, https://www.docker.com/.

[21] Concept: Entity-Control-Boundary Pattern, http://ndpsoftware.com/OpenUpBasic/openup_basic/guidances/concepts/entity_control_boundary_pattern,_uF-QYEAhEdq_UJTvM1DM2Q.html.
[22] Microservice–Architecture e-book: Creating Composite UI Based on Microservices, https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/microservice-based-composite-ui-shape-layout.
[23] Postman, https://www.getpostman.com/.
[24] I. Baig, "*Measuring cohesion and coupling of object-oriented systems–derivation and mutual study of cohesion and coupling*," Dissertation, http://urn.kb.se/resolve?urn=urn:nbn:se:bth-6010, 2004.