Software and Systems Modeling (2022) 21:1115–1133
https://doi.org/10.1007/s10270-022-00977-3

THEME SECTION PAPER

Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning

MohammadHadi Dehghani[1] · Shekoufeh Kolahdouz-Rahimi[1] · Massimo Tisi[2] · Dalila Tamzalit[3]

Received: 19 March 2021 / Revised: 8 January 2022 / Accepted: 12 January 2022 / Published online: 16 February 2022
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

Abstract

The microservice architecture has gained remarkable attention in recent years. Microservices allow developers to implement and deploy independent services, so they are a naturally effective architecture for continuously deployed systems. Because of this, several organizations are undertaking the costly process of manually migrating their traditional software architectures to microservices. The research in this paper aims at facilitating the migration from monolithic software architectures to microservices. We propose a framework which enables software developers/architects to migrate their software systems more efficiently by helping them remodularize the source code of their systems. The framework leverages model-driven reverse engineering to obtain a model of the legacy system and reinforcement learning to propose a mapping of this model toward a set of microservices.

Keywords Microservice architecture · Reinforcement learning · Model-driven reverse engineering · Migration

1 Introduction

Microservices (MSs) are becoming one of the prevalent architectural styles associated with cloud computing [14]. In a MS architecture, applications are developed as a suite of small autonomous and single responsible services, each operating according to its own process and communicating with standard and lightweight communication mechanisms and protocols. The power of MSs lies in their ability to be deployed independently and automatically, and to be combined to create more complex services. MSs can be easily scaled up and tested independently. Because of their fine granularity, MSs allow for applications that make more efficient use of available resources and systematically reuse basic functionalities. These features significantly increase the responsiveness of software systems, as new functionalities can be added in a more agile and affordable way.

Remodularizing monolithic software to many fine-grained MSs is one of the greatest challenges in migrating an existing system to a MS architecture [15,16,22]. Identifying the appropriate parts of code for each MS and then adapting them to the target MS are costly, error-prone and time-consuming [3,22]. Traditional software systems may contain very large modules; therefore, identifying and extracting the individual instructions that compose a fine-grained module from these code-bases are not a trivial task [22,29]. Although significant research has been conducted about decomposing software into MSs, the extraction of MSs by decoupling and reusing pieces of a bigger system, through refactoring and remodularization, is still an open problem [10]. We argue that a high-level model of the system can aid the decomposition of a large code base because it enables focusing on the important aspects of the system while removing the unessential information. Instead of considering the whole code base, at first, developers can focus on a much smaller representation. After that, developers can advance the decomposition by passing

Communicated by L. Burgueño, J. Cabot, M. Wimmer & S. Zschaler.

✉ Shekoufeh Kolahdouz-Rahimi
sh.rahimi@eng.ui.ac.ir

MohammadHadi Dehghani
mhdehghani@eng.ui.ac.ir

Massimo Tisi
massimo.tisi@imt-atlantique.fr

Dalila Tamzalit
Dalila.Tamzalit@univ-nantes.fr

[1] MDSE Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran

[2] IMT Atlantique, Nantes, France

[3] Université de Nantes, Nantes, France

to lower levels of abstraction and finally remodularize at the source code level.

Each MS in a system has a precise responsibility and claims ownership of some system resources. We refer to these resources with the term *nanoentities*, introduced in [17]. Nanoentities are elements used by a MS to provide business capabilities. They can be either data fields, operations, or artifacts. In [17], the authors propose a manual computer-aided decomposition process into MSs that requires defining a certain number of MSs and assigning all nanoentities to exactly one service. After this assignment, developers perform the decomposition by finding the methods that are related to each nanoentity and grouping them in a MS module (possibly adapting them). Developers move the methods that deal with nanoentities to the right MS module iteratively. The remaining monolithic software gets smaller, simplifying the identification of other MSs. The whole refactoring process is very long; therefore, a tool that can reduce this time by facilitating the mapping of the methods of the monolithic system to the most suitable MS is of high importance.

In this paper, we propose a framework that leverages model-driven reverse engineering (MDRE) and reinforcement learning (RL) for aiding the migration of a monolithic code base to a MS-based system. The framework takes three inputs: the existing source code, the entity–relationship (ER) model of the original system and its use case model.

Our processing contains three key steps:

1. A *model of the source code* of the system is automatically extracted. For this step, we use *MoDisco*, a well-known model-driven reverse engineering tool [6].
2. A *model of the microservices*, i.e., the desired result of the migration, is derived by analyzing the system design models. For this step, we leverage *Service-Cutter*, the tool from [17], to identify MSs from ER models and use case models.
3. We automatically propose to the user an assignment of existing methods in the model from step (1) to each MS identified in step (2).

The main technical contribution of the paper is step (3). This step is challenging because it needs to fill the gap between the high level of abstraction of the MS decomposition returned by step (2) and the low level of abstraction of the methods identified in step (1). Note that our contribution in this paper does not aim at improving the conceptual decomposition (i.e., the set of MSs), which is manually determined in step (2) with the help of Service-Cutter. We aim only at facilitating its application to the source code, which in previous work is completely manual.

For each MS, we produce the list of existing methods that should be considered in its implementation. This improves w.r.t. previous work, where microservices are typically associated with classes or packages of the initial code base. Fine-grained method-based migration is sometimes mentioned as part of future work in these proposals [10].

Mapping methods to MSs is a complex problem that depends on factors that are not easy to model, so we choose an end-to-end approach based on AI. The main reason for choosing RL is its ability to generate training episodes (by full plays of the game, e.g., in [27]), which allows us to avoid the problem of unavailability of public real-world systems in monolithic and decomposed forms. In this paper, we show that Deep Q-Learning, a reinforcement learning (RL) technique, can learn an efficient strategy for the distribution of methods to MSs in the user scenario. RL does not require a large training set and has been very successful in areas where specific goals need to be achieved in a given environment [27].

The developed framework is evaluated by applying it to 5 software systems of different sizes and structures. We provide both a ready-to-run (general) approach and a customizable (individual) approach. The general approach has good accuracy and performance for all the case studies, with a decrease in accuracy when the system size increases. The Individual approach increases accuracy for larger systems but requires additional training for the specific system to decompose.

The rest of this paper is organized as follows. Section 2 presents a sample system that we will use to illustrate our proposal. Section 3 introduces the main technologies used throughout the paper. Section 4 presents our framework and Sect. 5 discusses its AI in detail. Section 6 experimentally evaluates the framework. Section 7 surveys related work, and Sect. 8 concludes the paper.

2 Running example

This section provides a running case to exemplify the research problem. This example is used to illustrate how the proposed framework works in Sect. 4, and evaluate the usability and accuracy of our proposal in Sect. 6. The Cargo Tracking System is a notable software project designed to represent the ideas in the domain-driven design book [13]. This system is chosen as an illustrative example by many researchers because its characteristics are close enough to a real-world enterprise system to consider it a complex system while the domain model of the system is adequately simple and clear to be understood by any software developer as it is deliberately designed this way [4,17]. A sample implementation of this system is publicly available.[1]

In [18], the domain model, use cases, and some characteristics were extracted by analyzing the code of the system.

[1] https://github.com/javaee/cargotracker.

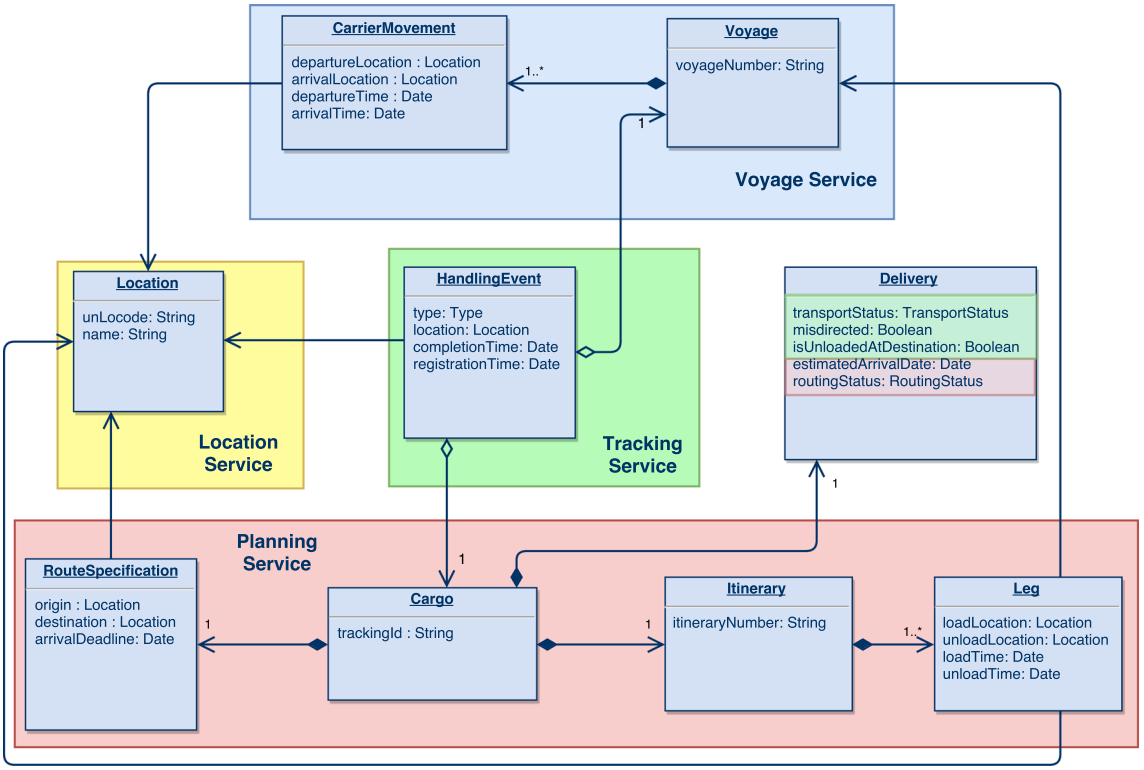

Fig. 1 Cargo tracking system domain model and service cuts with nanoentities [18]

The domain model is shown in Fig. 1. The Cargo Tracking System provides the following functionalities [18]:

- The purpose of the system is to ship a Cargo from the starting point Location to the destination Location. Each Cargo is issued with a TrackingId and a RouteSpecification. After a Cargo object is created, one of the appropriate Itineraries is assigned to it.
- The system calculates which Itineraries are suitable for a Cargo from the list of the existing Voyages. Each Voyage includes some CarrierMovements.
- When a Cargo is routed, HandlingEvents track the advancement of every Cargo's Itinerary. A HandlingEvent contains some information about the event in a particular Location and references a Cargo on a certain Voyage.
- The Delivery of a Cargo includes information about the state and estimated arrival time of the cargo and also indicates if the Cargo is on track.

In [18], an abstract decomposition of this system in MSs is described:

1. 4 MSs are identified, as the objective of the refactoring. We list them in Table 1.
2. The responsibility for each data field is assigned to one MS, as we show by coloring in Fig. 1. As depicted in the figure, the Delivery entity contains nanoentities belonging to both the Planning Service and the Tracking Service.

For bringing this abstract decomposition to the source code level, it is necessary to assign methods to MSs by analyzing their instructions, understanding their behavior, and choosing the MS (among the 4 identified in step 1) that should detain the responsibility for that functionality. The last step is especially time-consuming because it requires in general: (1) to inspect the full source code of the application, understanding the intent and side effects of each method, and (2) to choose a policy for assigning methods to MSs and use it consistently through the refactoring. In this paper, we aim especially at providing automated assistance to this last step.

3 Background

This section introduces the existing technologies leveraged in this research.

3.1 MoDisco

It is often necessary to have a high-level view of a previously developed system, for extracting information about its overall structure, without having to deal with low-level code [6].

Table 1 MSs of the cargo tracking system [18]

MS	Functional responsibility
Voyage service	Covers the part of the domain related to voyages and their movements, regardless of any cargo
Location service	Only contains the Location domain class. Locations are used and referred from almost all entities, but very rarely written. This service could be categorized as a master data service
Planning service	Handles the part of the domain regarding cargos and their itinerary
Tracking service	Responsible to track the actual events of a cargo

Reverse Engineering (RE) techniques are applied to existing systems for obtaining their abstract representation. Model-driven reverse engineering (MDRE) facilitates the extraction of a higher-level presentation of the system by leveraging the concepts of model-driven engineering and RE [6].

MoDisco is an open-source MDRE framework. It provides a set of tools for the extraction of a model view of the system. MoDisco has addressed different challenges of RE such as migration, refactoring, retro-documentation, and quality assurance. MoDisco considers three primary stages for reverse engineering a software system, including Model Discovery, Model Understanding, and Model (Re)Generation [6]. In the Model Discovery stage, models are derived from software artifacts. Then in the Model Understanding stage, these models are analyzed and sometimes modified. Finally, in the model (re)generation stage, the discovered models in the past stages can be utilized to produce different models or another variant of the software system [6]. In this work, we make use of the Model Discovery capability.

3.2 Service-Cutter

The MS architecture is one of the latest Cloud-native architectural styles. It is an architectural approach and not a technological one, and aims at being more sustainable than previous types of architecture since it fosters the development as a set of small independent services [21]. A MS is a service that has a single responsibility (the “micro" attribute). It is built so that it can be deployed, scaled, and also tested easily and independently. Thus, it can be requested without encountering dependencies, and it can be easily replaced and maintained [31]. Each MS is a business capability that can utilize various programming languages and data stores and is developed by a small team [11,14,22]. Systems with a MS architecture are composed of self-contained and independent MSs that interact only through standard and lightweight communication protocols [3]. In addition, the centralized management of these services is a completely separate service too, that may be written in a different programming language, use its own data model, etc. [32].

Service-Cutter [17] is a tool for assisting the migration of monolithic systems to MSs. It derives candidate service cuts from the ER model and use case model of the monolithic system, through graph clustering algorithms. It provides 16 coupling criteria from 4 categories of cohesiveness, compatibility, constraints, and communication, which are distilled from the literature and industry experience to be considered when decomposing a system into MSs. It searches for set of MSs, by considering their *bounded contexts* and independence in terms of *business scope*, which are core concepts in the MS architecture and domain-driven design [13,20,22]. This differentiates Service-Cutter from most migration tools, that aim at optimizing metrics related to class coupling and cohesion [10], and in some cases to the scalability and availability of MSs in the cloud [1].

Service-Cutter is based on a concept called *nanoentity* which represents a unit of resource of the system. The Service-Cutter approach argues that in order to provide capabilities, a service requires resources. Three types of resources are considered as the building blocks of services in this approach [17]:

- Data nanoentities: These nanoentities represent ownership over a field in the system’s data. The data can only be altered by the owner MS. Other MSs can be notified of the alterations in the data.
- Operation nanoentities: The business rules and processing logic are also considered nanoentities.
- Artifact nanoentities: These nanoentities are snapshots of data or operation results that are saved in a specific format.

According to [17], service decomposition can be defined as the process of identifying a set of services and assigning all nanoentities to one (and only one) of these services (Fig. 2).

3.3 Deep Q-learning

Reinforcement learning (RL) is an area of machine learning, used for teaching a computer program (an agent) how to achieve a certain goal inside environments that require sequential decision-making [25,28,33]. RL can solve problems in environments with certain or uncertain rules.

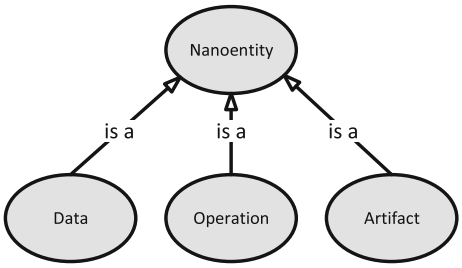

Fig. 2 Data, operations and artifacts generalized into the nanoentity concept [17]

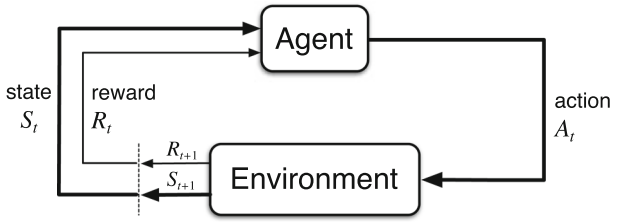

Fig. 3 Agent–environment interaction in a Deep Q-Learning problem [28]

Recently, there have been many advances in the field of RL in various domains, such as robotics and game playing. There are many policies to use inside the RL program [25,28,33]. Q-Learning is a commonly used method in RL that maintains a table containing the reward for each of the agent actions in each possible state, and it is hence suitable for a small number of states and actions. Deep Q-Learning can address larger state spaces by replacing the reward table with a neural network (NNET) that learns the reward for each action in each possible state. The state space for MS decomposition is generally very large because of the number of possible mappings of methods to MSs. In order to handle such a large state space, we apply Deep Q-learning.

We briefly list the common terms in Deep Q-Learning [28]:

- Agent: It is the part of the reinforcement learning program that takes actions in each state of the problem based on some policy.
- Game: The problem that is being solved by Q-Learning. This problem can have different states and throughout the game, the agent can take some actions.
- Action: By taking an action, the game will enter another state. The action taken in each step of the game is determined by the output of the NNET.
- Episode: Each time the game is played from beginning to end.
- State: Each different configuration of the game that occurs throughout the game. The state of the current step of the game is the input of the NNET.
- Reward Function: Function that computes the immediate value given by a certain action in a certain state.
- Q-value: It is an estimation for the cumulative expected reward from a certain state to the end of the game. The reward function only considers the next action, while Q-Value considers the whole path from the current state to the final state of the game. This ideal function is what the NNET tries to learn internally during training.

The agent is the learner and decision-maker. Any external process that the agent interacts with is called the environment. The agent and the environment interact continuously. The agent chooses actions and the environment responds to these actions and calculates the next state, based on the current state and the taken action. The environment also gives rewards to the agent and the agent tries to maximize the overall cumulative reward through its choice of actions. Figure 3 shows the interaction between the agent and the environment in Deep Q-Learning problems [28]. The agent receives the current State (S_t) and the current reward (R_t) from the environment, then it chooses an action (A_t) and sends it to the environment. Finally the environment starts the same process for the next state (S_{t+1}) and the next (updated) reward (R_{t+1}).

4 Proposed framework

In this section, we describe the global structure of the framework. The first steps exploit the tools from related work, i.e., Service-Cutter and MoDisco. The following steps, which are the main contribution of this research, take the result of the existing tools as input and produce the final output which is the mapping of methods to MSs. Finally, the user will have to manually refactor the code, using the generated mapping as a reference.

Figure 4 shows the structure of this framework, and Fig. 5 depicts the proposed workflow. The workflow consists of six steps. In the following, we provide a detailed description of each step. The detailed specification of the RL steps will be provided in Sect. 5.

4.1 Step 1: Artifact preparation

In the first step, the architect prepares the input artifacts required by the framework: an ER model for the system data, a use case model, and the source code of the project. The models are expected in JSON format. These input artifacts are the prerequisites for our framework.

A part of the expected ER model of the running example system is shown in Listing 1. The listing shows that the *Cargo Tracking* system has a *Voyage* entity that contains the *voyageNumber* attribute.

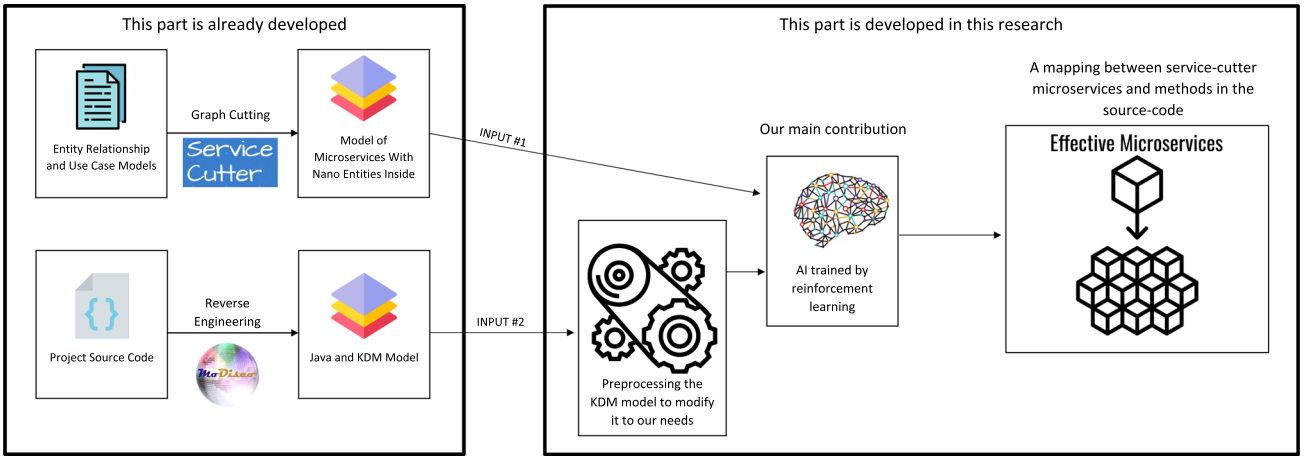

Fig. 4 Structure of the developed framework to facilitate migration to the MS architecture

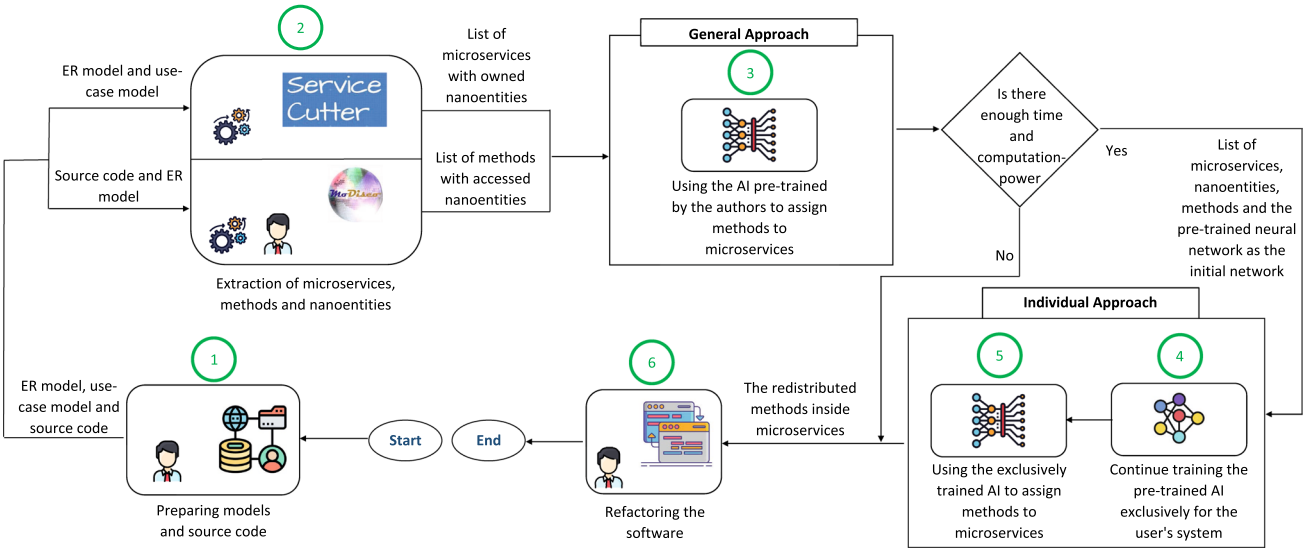

Fig. 5 Workflow of using the framework by the end-user consisting of six steps

```
{
 " name" : " Cargo  Tracking",
 " entities" : [
  {
    " name" : " Voyage",
    " attributes" : [
     " voyageNumber"
    ]
  },
  ...
 ]
}
```

Listing 1 A part of the ER model of the cargo tracking system expected as the output of Step 1

4.2 Step 2: Service-Cutter and MoDisco

In this step, the ER and use case models are given as an input to Service-Cutter, and the source code is analyzed by MoDisco.

Service-Cutter produces a model that indicates which MSs will be included in the target software system, and which nanoentities are under the responsibility of each MS. When a MS is responsible for a nanoentity, every instruction that reads or writes that nanoentity should be included in that MS. We call this model the *MS-Nanoentity model*.

For instance, a part of the MS-Nanoentity model of the running example system is depicted in Listing 2. The listing shows that Service-Cutter recommends the definition of a *Voyage Service*, that should be responsible for the *voyageNumber* nanoentity and of several nanoentities of the *CarrierMovement* entity.

```
{
 " services" : [
  {
   " id" : " 1",
   " name" : " Voyage  Service",
   " nanoentities" : [
     " Voyage.voyageNumber",
     " CarrierMovement.departureLocation",
     " CarrierMovement.arrivalLocation",
     " CarrierMovement.departureTime",
     " CarrierMovement.arrivalTime"
   ]
  },
  ...
 ]
}
```

Listing 2 A piece of the MS-Nanoentity model of the cargo tracking system expected as the output of Step 2

```
{
 " methods" : [
  {
    " name" : " viewTracking",
    " nanoentities" : [
      " Cargo.trackingId",
      " HandlingEvent.type",
      " HandlingEvent.location",
      " HandlingEvent.completionTime",
      " Delivery.transportStatus",
      " Delivery.estimatedArrivalTime",
      " Delivery.misdirected",
      " Voyage.voyageNumber",
      " RouteSpecification.destination"
    ]
  },
  ...
 ]
}
```

Listing 3 A part of the Method-Nanoentity model of the cargo tracking system expected as the output of Step 2

MoDisco generates a model from the source code in Knowledge Discovery Metamodel (KDM) [24]. From this model, we automatically extract a list of the methods that are inside the project. Each method needs to be annotated by the list of attributes and artifacts that are read or written by one of its potential executions. While this task can be automatized by static analysis of the source code, we currently require the user to perform it manually (automation is left for future work). We call the resulting model, the *Method-Nanoentity model*. Note that methods could access nanoentities that are not included in the MS-Nanoentities model, e.g., because they refer to technical aspects at the source code level of abstraction. Similarly, the system may contain technical methods that will not be mapped to a specific MS.

For the cargo tracking system, an excerpt of its Method-Nanoentity model is shown in Listing 3. The listing shows that the code base contains a *viewTracking* method that accesses several nanoentities, including *voyageNumber*.

4.3 Step 3: Application of the general NNET

The output models of Step 2 (MS-Nanoentity and Method-Nanoentity models) are given to a pre-trained NNET that is provided together with the framework. The result of this step is a mapping of the methods to the MSs.

We call this step the *general approach* because all users will use the same NNET to solve their problems. As this NNET is already trained, the users can get a result instantaneously.

The NNET is used in a Deep Q-Learning system, as an agent simulating a user that is mapping methods to MSs iteratively. We define an environment, and a reward function to evaluate the actions taken by the agent. The agent alters the environment randomly to find the best possible action for each given state of the environment. The general training is performed by feeding the NNET with a training set produced by an automatic model generator. The training set contains a large number of pairs of MS-Nanoentity and Method-Nanoentity models. In each episode of training, a different pair of models are used. This will enable the general approach's NNET to produce a solution to the refactoring problem for any given system, however, it may not give the best possible solution due to the variety of the training set (see Sect. 6.3.2).

```
Voyage   Service:
  [ createVoyage,  addCarrierMovement]
Location   Service:
  [ createLocation]
Planning   Service:
  [ viewCargos,  bookCargo,
    changeCargoDestination,
routeCargo]
Tracking   Service:
  [ viewTracking,  handleCargoEvent]
```

Listing 4 Result of Step 3 for the cargo tracking system

The result of this step for the running example is shown in Listing 4. Note that in this example the *viewTracking* method has not been mapped to the *Voyage Service*. While *viewTracking* accesses the *voyageNumber* nanoentity, that is under the responsibility of the *Voyage Service*, the AI did not consider this reason sufficient to include the logic of *viewTracking* in the *Voyage Service*.

If users are satisfied with this result which is produced immediately and requires no training nor development effort, they can skip to Step 6, otherwise, they can proceed to the next step to improve the quality of the result. The accumulated

reward of the mapping is a quality measurement criterion for the output.

4.4 Step 4: Individual training

In all cases, the user can further train the AI for the specific modularization problem, in order to search for better solutions. In Step 4, the user runs a training program that accepts the output models of Step 2 (MS-Nanoentity and Method-Nanoentity models) and the NNET of Step 3 as inputs and starts training the NNET further exclusively for the models of the system to refactor. This way the user can reach a higher quality result provided that they have enough time and computation power available. This approach is called the individual approach because the trained NNET will be specialized in a single system.

In this approach, the AI is trained for a specific problem. In other words, the AI knows from the beginning exactly what models will be given to it to be solved. The environment contains different states of a single pair of MS-Nanoentity and Method-Nanoentity models, therefore the training set concentrates only on a pair of models from one system. This concentration speeds up the training. The AI will be trained with one MS-Nanoentity model and one Method-Nanoentity model. Since the correct solution is not known a priori, the AI will play the given game, looking for strategies that maximize the cumulative reward. The solution that maximizes this reward during the training will be taken as the final output. The resulting NNET will be different for each system, and it can only solve the problem for that particular system. In our experimentation, with a sufficient number of training episodes, the individual approach gives the same or a better answer of the general approach (see Sect. 6.3.2).

While the training in the individual approach could start from a new NNET, we opt for transferring the general training to the individual approach. We have used the same NNET structure in both general and individual approaches so that the general NNET can be trained by the individual approach[2]. The training of the individual approach is completely automatic and no development effort is needed. The users only need to provide the individual approach training program, with the Service-Cutter and MoDisco models and the neural network will be trained automatically. In our experimentation, this technique causes a reduction of the training time without a loss in accuracy.

[2] As we will see in the next section, the general approach depends on the number of MSs of the system. The closest general NNET in terms of the number of MSs can be used as a starting point for the individual approach training.

4.5 Step 5: Application of the individual NNET

Step 5 is similar to step 3, as the simple application of a trained NNET. This time the trained NNET comes from Step 4. The user can also repeat steps 4 and 5 if they are not satisfied with the result. The accumulated reward of the mapping, returned by the RL algorithm after each full execution and the normalized accuracy (see Sect. 6), are quality criteria for the output.

Note that in our small example the general approach is already capable of finding the optimal solution, shown in Listing 4. The individually trained NNET will produce the same mapping as the general approach.

4.6 Step 6: Guided refactoring

After the user has a satisfactory result which may come from Steps 3 or 5, they can eventually use this result as a guide to manually refactor their code base in MSs.

For instance, given the solution in Listing 4, the user could start designing the *Voyage Service* MS by manually including and refactoring the code of the *createVoyage* and *addCarrierMovement* methods.

5 Application of Deep Q-learning

In both the individual and the general approaches, Deep Q-Learning is applied to find a mapping between MSs and methods of a given system. The instantiation of Deep Q-Learning terms in the context of this research problem is as follows:

Let M be the set of the methods of the monolithic system, each member of this set is a method and is represented with m. Also, MS represents the set of MSs which is provided by Service-Cutter and is produced based on the models of the monolithic system. Each member of this set is a MS and will be depicted as ms.

- Agent: The program that assigns methods to MSs.
- Game: Assigning all methods to their suitable MSs, one by one, until there are no methods left unassigned. It is possible to play the game several times. Each game iterates over the method set (M) and assigns each member (m) to a MS (ms). Each step of this sequential game corresponds to a certain method m and finding the best suitable MS for it.
- Action: Assigning the current step's method to a MS. So the number of possible actions in each and every step is the number of MSs ($|MS|$). Each step of the game ends with taking one action.
- Episode: Playing the game once from the beginning to end, i.e., each time all the methods are assigned to MSs.

Each episode includes taking a series of actions (steps) to reach the end. The number of these steps is the same as the number of the methods of the system ($|M|$). Each episode is an iteration over the method set M.

- State: The configuration of the game in each step of an episode. It contains information about the chosen MS for each method we have considered so far in the game.
- Reward Function: A number indicating how good or bad is assigning a certain method *mt* to a certain MS *ms* in a certain state. Listing 5 shows the pseudo-code of the reward function used in both approaches. First, an integer r is initialized to 0. Then, we compare the nanoentities of the method with the nanoentities of all the MSs of the system. For each identical nanoentity we find, if it was found in the chosen MS, 20 points are added to the reward, otherwise, 10 points are reduced from the reward. Then, the function rewards cases where the set of nanoentities of the method is a subset of the nanoentities of the MS. This will cause minimum coupling between MSs as Service-Cutter classifies nanoentities into separate groups. The fewer the methods of a MS that access the nanoentities of other MSs, the less coupled the MSs are. So, for each nanoentity that is present in the method, but not present in the MS, 10 points are reduced, otherwise, 20 points are added. Finally, if the chosen MS has already exceeded the average number of methods, 5 points are reduced from the reward. This small reduction is useful to decide assignments between two MSs that would give the same reward based on the previous calculation. The global reward function prioritizes mappings between methods and MSs with highly similar nanoentities, regardless of the ordering. Note that the exact reward values we use have been determined by several executions of the algorithm over the given use cases.
- Q-value: An estimation of how much reward we will get from all the remaining steps of the refactoring if a particular method is assigned to a particular MS in the current step.

The MS-Nanoentity and Method-Nanoentity models are the inputs of the RL program. A game is defined inside the RL program to be played and mastered by the AI. The game consists of steps and in each step, an action is taken by the AI and then a reward is given to the AI indicating how good its action was. This is how the AI will learn to take better actions in each step of the game. For our RL, in each step of the game, a certain method is assigned to an MS. The reward function will indicate how good was the assignment of that method to that MS, in that particular state. For instance, an MS may already have several methods assigned to it due to the previous steps. Consequently, overfilling an MS will not be desirable.

```
int reward( MS ms, Method mt){
 int r = 0
 for each MS m except ms {
   for each nanoentity m and mt
share{
     r -= 10
   }
 }
 for each nanoentity n inside ms {
   if n is shared with mt{
     r += 20
   }
 }
 for each nanoentity n inside mt {
   if n is shared with ms {
     r += 20
   } else {
     r -= 10
   }
 }
 if ms has more than the average
   number of methods assigned {
     r -= 5
 }
 return r;
}
```

Listing 5 Pseudo-code of the reward function for assigning a method called `mt` to a MS named `ms`

While in our tool we use NNETs of various sizes, in Fig. 6 we show a small sample with 4 output nodes and 12 input nodes, to illustrate the structure of the NNET used in both approaches. We apply the NNET to decompose a system model with 4 MSs (the right side of the figure): MS1, MS2, MS3, and MS4. The NNET includes three main layers. The input layer, the hidden layers, and the output layer. Each layer consists of some nodes and each node is connected to all the nodes of the next layer, except the nodes of the output layer. The nodes of the input layer get their values from the agent. There are 3 hidden layers between the input layer and the output layer, each containing 100 nodes. The nodes of the first hidden layer get their values from the input layer nodes' values multiplied by the weights of the edges connecting the two layers. In the same manner, the second hidden layer gets values from the first hidden layer, the third hidden layer gets value from the second and the output layer nodes get their values from the third hidden layer nodes. Each of the nodes of the output layer represents an action which in the context of this research corresponds to an MS. The input layer has three nodes for each MS of the system model. The first node holds the number of matching nanoentities between the method of the current step and the first MS. The second node holds the number of non-matching nanoentities between the method and the first MS. The third node holds the number of already assigned methods to the first MS. The rest of the input layer is filled in the same manner for the second, third, and

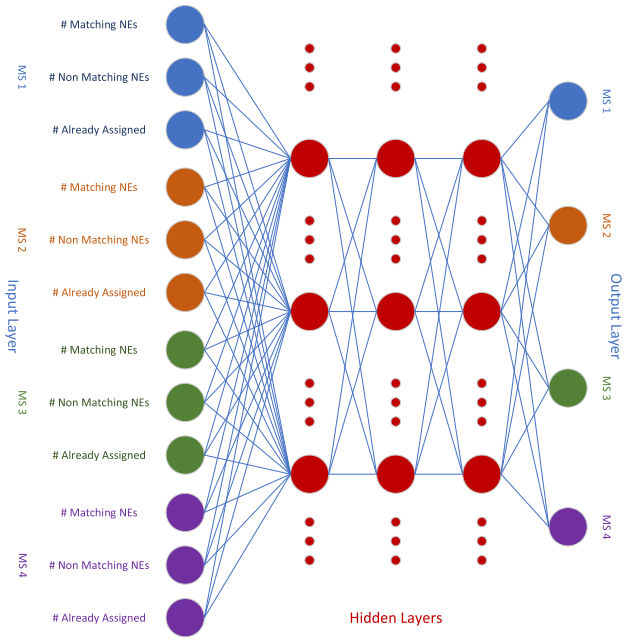

Fig. 6 Structure of the neural network used in the individual and the general approach

fourth MS of the system. Therefore, in each step of the game, we are considering one particular method of the system and all of the MSs of the system. This causes the NNET to be more reliant on the number of the MSs of the system, rather than the number of the methods or the nanoentities. In the training phase, in each step of the game, the weights of the edges of this NNET evolve based on the feedback reward. This alteration is done so that the NNET would produce a better output the next time. Therefore, the NNET will choose better actions that cause better feedback rewards in the future. When the training phase is over, the NNET will be used to solve real problems. This time, for each step of the game, the agent asks the NNET for the best action in the current state and the game proceeds to the end.

6 Evaluation

To evaluate the accuracy of the framework in solving a real problem, both the individual and general approaches have been applied to five case studies including the Cargo Tracking System presented in Sect. 2. First, we present the complete solution to the running example in Sect. 6.1, and then we discuss the accuracy and the speed of the proposed framework in solving four other case studies from literature in Sect. 6.2. We discuss how the approach scales in Sect. 6.3, and summarize the evaluation key results in Sect. 7.

6.1 Solution of the running example

The implementation of the cargo tracking system has many methods. The most challenging methods to assign to MSs are those which provide the main functionalities, namely, the business functionalities represented in use cases of the system. We consider these extracted use cases of the system as the main methods that need to be assigned to MSs. The list of these methods, with the nanoentities each of them reads or writes, is as follows [18]:

1. **viewTracking**:
   - Nanoentities written: -
   - Nanoentities read: *Cargo.trackingId*, *HandlingEvent.type*,*HandlingEvent.location*, *HandlingEvent.completionTime*, *Delivery.transportStatus*, *Delivery.estimatedArrivalTime*, *Delivery.misdirected*, *Voyage.voyageNumber*, *RouteSpecification.destination*, *Stock.stockName*
2. **viewCargos**:
   - Nanoentities written: -
   - Nanoentities read: *Cargo.trackingId*, *RouteSpecification.destination*, *RouteSpecification.arrivalDeadline*, *Delivery.routingStatus*, *Itinerary.itineraryNumber*
3. **bookCargo**:
   - Nanoentities written: *Cargo.trackingId*, *RouteSpecification.origin*, *RouteSpecification.arrivalDeadline*, *RouteSpecification.destination*
   - Nanoentities read: *Location.unLocode*
4. **changeCargoDestination**:
   - Nanoentities written: *RouteSpecification.destination*
   - Nanoentities read: *Cargo.trackingId*, *RouteSpecification.destination*
5. **routeCargo**:
   - Nanoentities written: *Itinerary.itineraryNumber*, *Leg.loadLocation*, *Leg.unloadLocation*, *Leg.loadTime*, *Leg.unloadTime*
   - Nanoentities read: *Cargo.trackingId*, *RouteSpecification.destination*, *RouteSpecification.origin*, *RouteSpecification.arrivalDeadline*, *Location.unLocode*, *Voyage.voyageNumber*, *CarrierMovement.departureLocation*, *CarrierMovement.arrivalLocation*, *CarrierMovement.departureTime*, *CarrierMovement.arrivalTime*
6. **createLocation**:
   - Nanoentities written: *Location.unLocode*, *Location.name*

- Nanoentities read: -

7. **createVoyage**:
   - Nanoentities written: *Voyage.voyageNumber*
   - Nanoentities read: -
8. **addCarrierMovement**
   - Nanoentities written: *CarrierMovement.departure Location*, *CarrierMovement.arrivalLocation*, *CarrierMovement.departureTime*, *CarrierMovement. arrivalTime*
   - Nanoentities read: *Voyage.voyageNumber*
9. **handleCargoEvent**
   - Nanoentities written: *HandlingEvent.type*, *Handling Event.completionTime*, *HandlingEvent.registration Time*, *HandlingEvent.location*, *Delivery. transportStatus*, *Delivery.misdirected*, *Delivery. estimatedArrivalTime*, *Delivery.isUnloadedAt Destination*, *Delivery.routingStatus*
   - Nanoentities read: *Voyage.voyageNumber*, *Cargo.trackingId*

Given the principles of loose coupling and bounded context [22], the correct mapping of the methods to MSs is the one shown in Listing 4. We train our AI and confirm that it reaches this mapping, with the given models of the Cargo Tracking System as inputs. The performance of this process is discussed in the next subsection.

6.2 Accuracy and execution time

6.2.1 Experimentation setup

We first evaluate our proposal in terms of accuracy and execution time. For the experimentation, we use five case studies that were already used for experimental evaluation in existing work on MS migration [18,19].

As these case studies consist of 3–9 MSs and 5–19 methods, we were able to manually define by inspection an ideal mapping for all of them. Both the general and the individual approaches can reach the optimal mapping with less than two thousand episodes of training. To be able to quantify how far we are from the optimal mapping in the experimentation, we define a value function with respect to our goal (having bounded contexts and microgranularity). The value function is maximized by the optimal manual mappings of our five case studies. We use this value function in this section to evaluate the accuracy of our approach. In the next section, we will use it to discuss how this accuracy changes as the learning progresses.

The value function we use is the weighted average of two parameters: one concerning the context bounds the methods that are mapped to a MS should as much as possible access only the nanoentities of that MS) and one concerning the granularity of MSs (a MS should have small size). We give to the context-bound parameter double weight w.r.t. granularity, to make it a priority.

Figure 7 shows the formula of the context-bound parameter in Equation (1) and the formula of the granularity parameter in Equation (2). Both parameters range from 0 to 1, where a higher value indicates a more valuable solution. The notations of *MS*, *M*, and *NE* represent the set of all microservices, methods, and nanoentities respectively. The *contextNE(ms)* and *NE(m)* functions, return the nanoentities of a microservice (ms) and of a method (m) respectively. The *mappedMS(m)* and *mappedM(ms)* functions, return the microservice that a method m is mapped to, and the method that a microservice ms is mapped to, respectively.

To quantify how much this solution respects the bounded context, the formula counts, for each method m, how many of its nanoentities are under the responsibility of the MS to which m is mapped. This formula is maximized if all the nanoentities that are under the responsibility of the MS are included in at least a method mapped to that MS.

To evaluate the granularity of MSs, the function compares two ratios for each MS: (1) the number of the nanoentities (responsibilities) of a MS divided by the number of all nanoentities of the system and (2) the number of methods mapped to a MS divided by the number of all the methods of the system. If these ratios are close, the formula approaches close to 1, indicating a good granularity.

By using this value function, we also compare our accuracy and performance to two baseline approaches, Random Search and Brute Force. These approaches search through the possible solutions the ones that maximize the value function. Random Search explores a random set of solutions, while brute force proceeds in a deterministic order and will explore all solutions if given enough time.

In our experimentation, we first identify for each case study the maximum value function for the system. For the three smaller systems (Booking, Cargo, and Trading) we compute automatically this value by applying the brute force approach. Finally, we compute a normalized accuracy as a percentage representing the ratio of the value function for the solution divided by the maximal value function for the system. For the Insurance and Lakeside systems that are bigger and the optimal solution cannot be found by brute force, we consider the maximum ideal accuracy to be 100 percent and ignore the fact that a solution with 100 percent accuracy might not exist at all.

$$contextBoundNormalized = \frac{\sum_{m \in M} |contextNE\,(mappedMS\,(m)) \ \cap \ NE\,(m)|}{\sum_{m \in M} \ |NE\,(m)|} \tag{1}$$

$$granularityNormalized = 1 - \frac{\sum_{ms \in MS} \left| \frac{|contextNE(ms)|}{|NE|} - \frac{|mappedM(ms)|}{|M|} \right|}{|MS|^2} \tag{2}$$

Fig. 7 Components of the value function related to context-bound and granularity

Table 2 Required time to reach the best solution for each case study

Case study	# of MSs	# of Methods	Problem space size	Time to reach the best solution (ms)		
				Random search	Brute force	RL (general approach)
Booking	3	5	243	1611	1962	93
Cargo	4	9	262144	4323	10520	103
Trading	4	10	1048576	17847	32797	107
Insurance	9	10	$3/5 \times 10^9$	15 h (estimate)	1 day (estimate)	119
Lakeside	5	19	$1/9 \times 10^{13}$	10 years (estimate)	17 years (estimate)	166

6.2.2 General approach evaluation

Table 2 shows the required time by the random search, brute force, and general approach to reach the best solution for each case study.

From partial runs of the brute force approach, we can estimate that a brute force search on the two largest case studies would take days or years, respectively, to complete. The random search approach is applied until we reach the best solution which we previously found out by brute force. For the larger case studies, the completion time of the random search is estimated after completing increasingly complex problems.

Finally, we execute the Deep Q-Learning agent using the pre-trained general NNET to check if we can reach the best solutions. As shown by the table, the general approach reaches the best solutions in the first three case studies. For the bigger case studies, where we do not formally know the maximum accuracy possible, a 98% normalized accuracy is gained.

We can observe that the execution time for the random search and brute force approaches grows steeply with the size of the case studies. Instead, as the NNET in the general approach is already trained, it solves all the case studies in about 100 milliseconds and there is no need to apply the individual NNET to any of the use cases. The individual training is confirmed to be useful for decomposing the randomly generated bigger systems (see Table 4).

In Table 3 and Fig. 8, we can see how the accuracy of the general approach evolves during training. For all case studies, the accuracy of the general approach tends to increase with the number of training episodes. Large systems require very long training to stabilize. The graph shows that interrupting the training at 2000 episodes would not be enough to have a stable accuracy for problems of the size of the Insurance use case. That is why we have trained the general NNETs for 100,000 episodes. During training, the NNET performs abstractions from the training episodes, intuitively to derive a general rule. The fluctuations in accuracy are sometimes due to abstractions from a first set of episodes, that are proved wrong by a following set, and need to be replaced by different abstractions. This is common in training NNETs [28].

Table 3 Normalized accuracy of the general approach during training (tabular)

Episodes	Normalized accuracy in case studies				
	Booking	Trading	Cargo	Insurance	Lakeside
250	100	64	94	27	35
500	100	100	99	76	53
750	100	100	100	65	91
1000	100	100	100	46	97
1250	100	100	100	98	98
1500	100	100	100	81	96
1750	100	100	100	98	98
2000	100	100	100	98	98

6.2.3 Individual approach evaluation

As the individual approach is run on the user's computer, we show how its accuracy evolves w.r.t. its execution time (i.e., as number of episodes). We compare this accuracy with the random search and brute force approaches. Figures 9, 10, 11, 12 and 13 show for each case study (in the order of size of the case studies) the normalized accuracy of the approaches w.r.t. the number of episodes each approach has

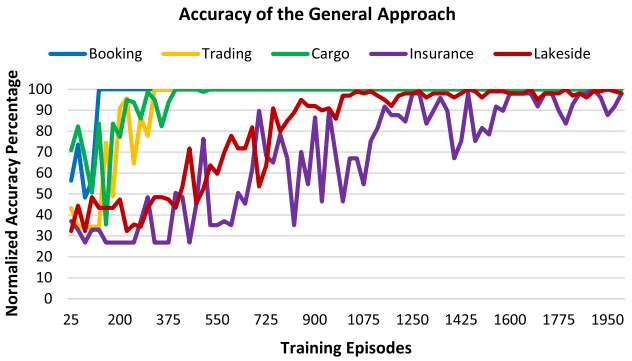

Fig. 8 Normalized accuracy of the general approach during training (graphical)

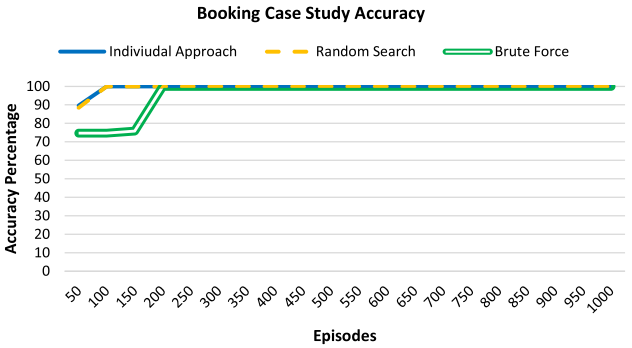

Fig. 9 Normalized accuracy of the individual approach in solving the Booking case study

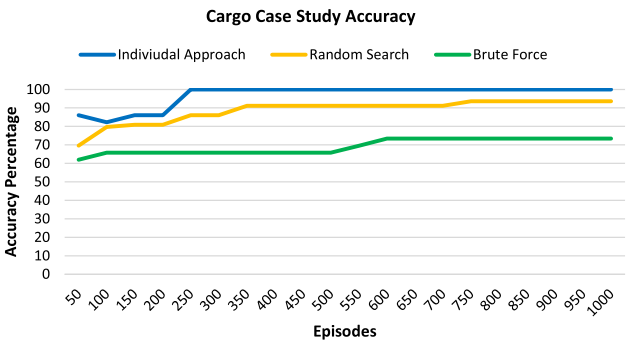

Fig. 10 Normalized accuracy of the individual approach in solving the Cargo Tracking case study

passed. Each episode corresponds to arriving at a complete solution, regardless of its accuracy.

The results show that in the smallest case study (Booking), all three approaches are able to reach the maximum accuracy before 1000 episodes. As the case studies get bigger, the individual approach takes a clear lead in reaching higher levels of accuracy.

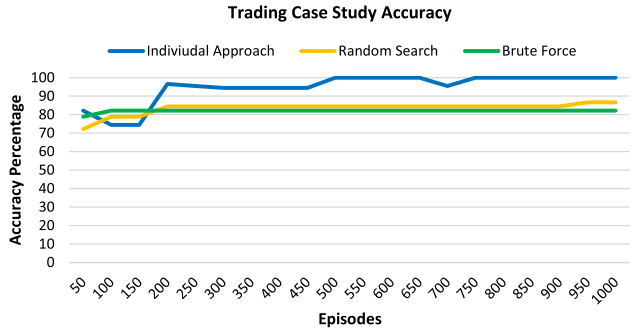

Fig. 11 Normalized accuracy of the individual approach in solving the Trading case study

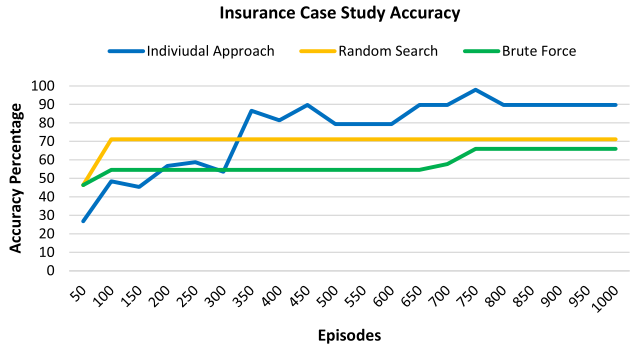

Fig. 12 Normalized accuracy of the individual approach in solving the Insurance case study

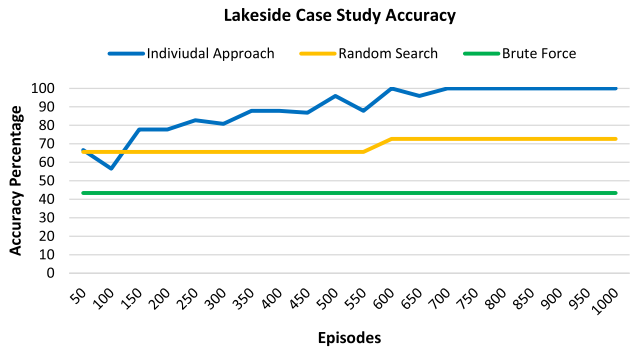

Fig. 13 Normalized accuracy of the individual approach in solving the Lakeside case study

6.3 Scalability

The most time-consuming part of the developed framework is the AI training. To find out how much time is needed for the training phase, we need to know how much time a single episode takes to complete, and how many episodes are needed for the AI to reach a certain accuracy. These parameters are discussed in Sects. 6.3.1 and 6.3.2, respectively. The computer on which these benchmarks were run, has 12 GB of RAM and 4 processor cores of 2.5 GHz.

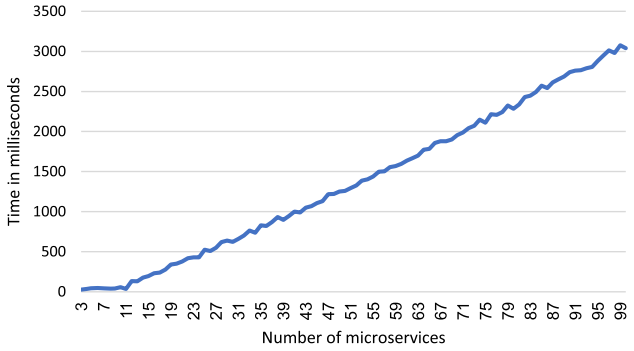

Fig. 14 Time needed for each episode of the training program to complete based on the number of MSs

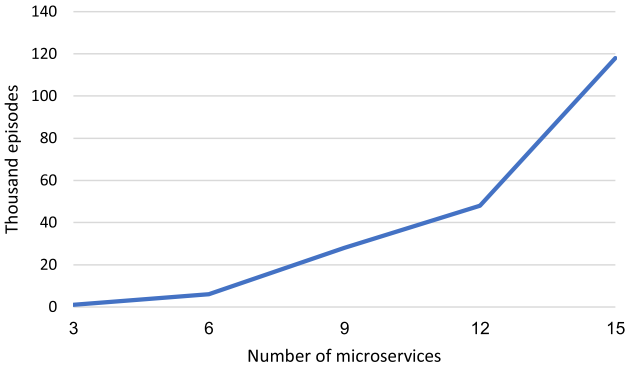

Fig. 15 Thousand episodes needed to reach 80% accuracy based on the maximum number of MSs in general approach

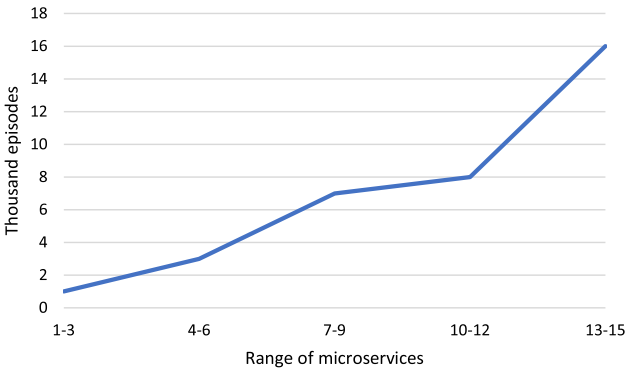

Fig. 16 Thousand episodes needed to reach 80% accuracy based on the range of MSs in the general approach

6.3.1 Time charts

As previously mentioned, an episode of a Deep RL system requires playing the training game once from beginning to end and reaching a solution. The duration of an episode depends on the size of the problem. To estimate this dependency for large systems we automatically generate 100 system models, each containing 1–100 MSs, respectively. Figure 14 shows the time needed for one episode to be completed (in both general and individual approaches) for each one of these systems. The chart shows linear behavior. It means that the time needed for an episode to complete increases as the number of MSs increases. This is mainly because the size of the input layer and the output layer of the NNET depend on the number of MSs. Note that the result obtained with randomly generated systems can be generalized to any system, since the completion time of an episode does not depend on the structure of the system, but only on its size.

6.3.2 Episode charts

In this section, as we want to evaluate only the training speed of approaches (and not their real-world accuracy), we consider the number of episodes needed to reach 80% accuracy for synthetic system models of different sizes. The system models we use are generated with the following configuration: `n` MSs, `n` nanoentities, and `n` methods, where each nanoentity is only present in one MS and one method, but randomly indexed. Since in this simplified scenario the methods that should be mapped to each MS are easily identified (by their exclusive nanoentity), then it is trivial to calculate the accuracy of the solution.

The NNET trained by the general approach computes a solution for any system, while the NNET trained by the individual approach is specialized for a particular system. Therefore, they are discussed separately.

General Approach Episode Charts. As the training set of the general approach includes several different system models, its training requires a high number of episodes. At first, we tried to train a single NNET to solve problems with any number of MSs. So it was needed to set a limit to the maximum number of MSs the AI can solve. As Fig. 15 shows, we set the maximum number of MSs to 3 and gradually increased it to see how many episodes are needed for the AI to reach 80% accuracy based on the maximum number of MSs.

Then, instead of training only one NNET to solve problems with 1 to n MSs, we trained multiple NNETs, each of them able to solve the problems in the range of 3n+1 to 3n+3 MSs ($n \geq 0$). Figure 16 shows the number of episodes needed to reach 80% accuracy for each one of the limited-range general NNETs. This approach takes much less training than the former to reach the same accuracy. This is because the problem it needs to solve is more constrained.

Individual Approach Episode Charts. The individual approach only trains on one set of models and solves the same problem over and over again, each time taking random actions and searching for solutions that maximize the global reward. Focusing on one system makes the training much faster than the general approach which has to deal with a huge number

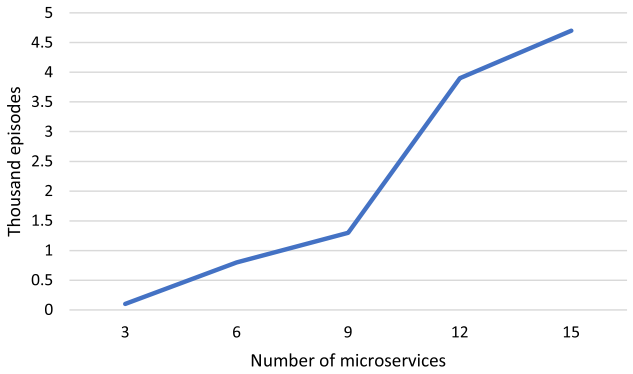

Fig. 17 Thousand episodes needed to reach 80% accuracy based on the number of MSs in individual approach

Table 4 Thousand episodes needed to reach 80% accuracy based on the number of MSs in different approaches

# of MSs	General approach (unlimited range)	General approach (limited range)	Individual approach
3	1	1	0.1
6	6	3	0.8
9	28	7	1.3
12	48	8	3.9
15	118	16	4.7

of variant problems. Figure 17 shows again the number of episodes required to reach 80% accuracy for the same sample models used for the general approach.

Comparison. Table 4 summarizes the number of episodes needed to reach 80% accuracy in different approaches. Even though the training for the general approach can be accelerated by limiting the range of the number of MSs, the individual approach still is much faster in training. The table also shows that the training of limited-range general NNETs is more effective than training a single general NNET. In our experimentation, 35,000 total training episodes for the set of limited-range NNETs produce an agent that has the same accuracy as a single general NNET trained by 118,000 episodes. As each training episode only takes about a few hundred milliseconds, the individual approach can migrate systems with up to 15 MSs with minutes of training. The general approach required hours of training to obtain the same result on systems with up to 15 MSs. The curves show that longer training would make our approach, able to decompose bigger systems. However, we experimentally evaluated the accuracy of these approaches in migrating systems with more than 9 MSs only for randomly generated systems, as the available real-world case studies had 9 MSs maximum.

6.4 Discussion

Two approaches have been introduced in this research. The general approach produces immediate solutions and generates high-quality solutions for systems with up to 15 MSs. The individual approach is able to improve the quality of the results of the general approach for the cases where the general approach may not reach 100% of accuracy, but it requires additional training by the user. However, it does not require any programming effort for training as the training program is already developed and is ready to be compiled and executed. The source code of both approaches is available online in a GitHub repository.[3] The process of using the NNET trained during the general approach as a starting point for training in the individual approach is in the spirit of transfer learning, i.e., it reuses the knowledge stored in an NNET, to solve a similar problem [33].

The case study of the Cargo Tracking System is used in this research to present the correctness and applicability of the developed framework. Additionally, the scalability of the framework is illustrated by the time charts and the episode charts. The results indicate that for systems with up to 15 MSs, the general approach will produce high-quality results (98% to 100% accuracy for all the five use cases). This is due to the limitation of the computation power of the system that we have used throughout this research. Additionally, it is possible for the users of the framework to improve the results for bigger systems by applying the individual approach. For example, it would only take a few hours to train the individual NNET adequately for a system with 30 MSs. The presented time chart proves the linearity of the time needed for an episode of training, with respect to the number of MSs of the system. The Episode charts start off linear at first, but as the number of MSs grows, they shift from linear to exponential. This shifting occurs slowly in the individual approach and, therefore, it is able to solve the problems with higher numbers of MSs more efficiently. Additionally, this shifting gets more pace in the limited version of the general approach. Furthermore, the unlimited version of the general approach shifts from linear to exponential in an even faster manner. This means that the individual approach will produce a better result than the general approach for systems with a higher number of MSs.

We did not observe any phenomenon of overfitting with training up to 120,000 episodes in the case studies and scalability evaluations.

RL is commonly used in sequential problems, whereas in our approach, it is being used in an unordered problem space. However, there is no evidence that the sequential nature of RL worsens performance in unordered problems.

[3] https://github.com/hadiDHD/MS-MDE-RL.

Our good experimentation results confirm the applicability of this choice.

6.5 Threats to validity

The use cases we have considered have different problem space sizes and structures, although for evaluating the performance of our framework when dealing with bigger systems, we developed a random model generator that can generate both the services model and the methods model of an artificial randomly generated software system. Decomposing randomly generated systems has generally different performance than decomposing real systems, posing a threat to the possibility to generalize those experimentation results. However, our experimentation shows that solving randomly generated models takes more time than solving real-world case studies of similar size. This can be seen by comparing Table 4, which shows the number of training episodes to reach 80% accuracy based on the number of MSs of the system, and Table 3, which consists of the number of training episodes for the five use cases. Therefore, we expect that our approach will have better performance for large real-world systems w.r.t. the generated ones we used here.

In this research, the authors have leveraged two tools developed by other researchers. This fact naturally causes dependency and also makes our framework vulnerable to potential validity issues caused by these tools, even before our approaches are used. It should be noted that the authors of this paper, did not encounter any specific validity issues when leveraging these two preliminary tools when evaluating the five uses cases.

7 Related work

Our work intersects the research areas of migration to MSs, MDE, and AI. Hence, we divide the related work into two groups. The first one includes works related to the application of AI in MDE. The second one includes research that investigates migration to MSs by leveraging MDE or AI techniques.

7.1 MDE and AI

The effectiveness of applying DL to solve and automate various problems in the field of MDE is presented in different researches. In [8] the ML-based framework is developed to drive transformation automatically from sets of input/output models. Therefore, by applying this framework, there is no need for developers to write transformation specifications. In this research, the models are preprocessed and represented as an independent tree. These models are then fed to the artificial neural network (ANN) to provide an automatic representation of transformation. The evaluation presents a positive result by applying this framework to models of different sizes. Nguyen et al. [23] presented a tool for automatic supervised classification of metamodels by applying ML techniques. In this research, the NN learns labeled metamodels and then classifies unlabeled ones. The result of the evaluation of different metamodels reflects the effectiveness of this tool in categorizing the unlabeled data. For the purpose of repairing models, the concepts of reinforcement learning (RL) are applied in [5]. The learning process is performed through the interaction between the learning agent and its environment. The EMF-based tool is presented in this research to identify and repair the errors. This approach is tested on the number of broken models and provided personalized solutions for all of them. An automated process is introduced in [2] to analyze and compare a large number of (meta)models by applying model clustering techniques, as an unsupervised ML technique and considering the structural context in the form of n-grams. The result of the evaluation in this research shows the high accuracy of n-grams. In order to cluster software modeling artifacts, graph kernels are applied in [9]. Clustering enables the user to handle a large collection of models for different tasks such as validation, verification, and testing. Additionally, it is possible to generate model diversity in this research to enforce model collection to include models with distinct sizes. In this research, positive evaluation results are presented in terms of efficiency and usability after applying this approach.

7.2 Migration to microservices

In [12], an approach is presented to modernize legacy applications into MSs with the help of a model-centered process that analyzes and visualizes the current structure and dependencies between the business layer and the data layer of Java Enterprise applications. Different diagrams are generated to help remodularize the system to MSs. These diagrams are generated based on the models extracted from the software system via MoDisco. The main difference between their research and ours is that they only facilitate the migration to MS architecture by generating some visual artifacts from Java annotations to help the developers understand the structure of the monolithic system, whereas our approach produces actual source code decomposition. In [7] the challenges of migration from monolithic applications to MS-based applications are tackled. In this paper, a model-driven approach is implemented utilizing JetBrains MPS (a text-based metamodeling framework) for the automatic migration to MSs. The solution is composed of two major components, an MS Miner and an MS Generator, leveraging two Domain-Specific Languages (DSLs), one for the MS specifications, and another for their Deployment. The input monolithic application is written in Java and the generated MS-based

application is represented with the "Jolie" language (a programming language for defining MSs). The responsibility of MS Miner is to search in the abstract syntax tree of the imported Java code for specific patterns and suggests to the developer the set of MSs for the migration. MS Generator uses model-to-text transformations to generate MS specification and deployment files from the models extracted by MS Miner. Differently from them, we start from a conceptual decomposition in a set of MSs manually derived with the help of Service-Cutter, and then we look for a mapping of the code to those MSs. In [1] an automatic decomposition method targeting application scalability and performance is proposed. The presented decomposition method leverages a black-box approach that uses the application access logs and unsupervised machine learning to auto-decompose the application into MSs. The application access logs are mined using a clustering method to discover URL partitions having similar performance and resource requirements. Such partitions are mapped to MSs. The proposed method supports auto decomposition to MSs, deploying the MSs using appropriate resources, and auto-scaling the MSs. The experimental evaluation shows improved performance of the auto-created MSs compared with the monolithic version of the application and the manually created MSs. Differently from us, the authors aim at optimizing the system performance, while we focus on the business domain and bounded context for each MS. In [30], a model-driven tool for a specification of REST MS architectures is presented. This work aims to automate and ease the process of MS software architecture specification and configuration. A DSL for MS software architecture modeling named MicroDSL is developed for this purpose. The abstract syntax of this language is represented by a meta-model specified in Ecore. The MicroDSL concrete syntax specifies the MS architectures. A set of code generators as part of the MicroBuilder tool are developed to generate executable program code, based on MicroDSL specification. The proposed tool is evaluated by applying it to a case study and considering a comprehensive questionnaire. It should be noted that this research facilitates developing MSs from scratch while our research supports the migration from monolithic architectures toward MSs. In [26], an approach is introduced for the identification of MSs from object-oriented source code. They consider the relationships between the elements of the source code, the data layer, and some help from the architect of the system to identify MSs via a hierarchical clustering algorithm. They evaluate their work by comparing the (semi)automatically identified MSs with the manually identified ones. While this work shares a similar objective to ours, our accuracy is higher: we automatically find the optimal solution in all considered use cases, while a perfect match is rare in their experimental evaluation.

8 Conclusion and future work

In this research, a framework is developed to find the best possible mapping of methods to MSs, in order to remodularize monolithic software systems. The purpose is to facilitate the migration to an MS architecture. The framework exploits existing tools and contains two training approaches for the embedded NNET. The applicability, correctness, and scalability of the developed approaches have been evaluated in this paper by applying them to well-known use cases and running performance benchmarks.

We plan the following future work:

Automatic annotation of methods with nanoentities. As it was mentioned in Sect. 4, the user of the framework annotates each extracted method with the read/written nanoentities inside that method. However, this step can be automatized, by static analysis of the source code. The user would only need to define a relation between the nanoentities and the data access layer of the source code and all the methods that directly or indirectly access these nanoentities would be annotated automatically.

Automatic Refactoring. The final output of our framework is a mapping of methods to MSs. The developers will have to refactor these methods manually to the mapped MSs. Automating this step will provide the opportunity of building a fully automatic framework that is able to migrate the source code of a monolithic software system to the source code of a MS-based software system.

Neural Networks. The NNET-based approach in deep RL can be naturally extended with more complex NNET topologies and sets of input features. In particular, we are interested in integrating method and nanoentity names in the NNET learning process, so that the agent can make decisions based on name similarity.

Method Decomposition. In some cases, the developers may want to decompose some methods so the parts of the method that read/write a certain set of nanoentities, would be separated. We want to add these actions to the RL agent and produce a new reward function accordingly. The reward function would assign a proper negative reward based on the number of lines of code inside that method.

Learning the Bounded Context. In the future, we would like to extend the determination of microservices granularity by taking into account the notion of *bounded contexts* advocated by Evans' domain-driven design approach [13]. This notion of bounded context is fundamental to identifying the right granularity of a microservice. It represents a subsystem aligned with part of the business domain. Thus, a bounded

context specifies a coherent and unified business perimeter for each service. Future work will focus on extending the semi-automated approach to identify bounded contexts of a software system that needs to migrate toward a microservice architecture. One of the main issues to address is when central concepts of the domain are polysemous, generally tied to the fact that there are different stakeholders within the domain. For instance, the concept of "Customer" is present for the sales department and also for the support department. The challenge will be to identify, in our RL approach, the common concepts, before working on defining the mapping between these polysemous concepts for aims of integration.

References

1. Abdullah, M., Iqbal, W., Erradi, A.: Unsupervised learning approach for web application auto-decomposition into microservices. J. Syst. Softw. **151**, 243–257 (2019). https://doi.org/10.1016/j.jss.2019.02.031
2. Babur, Ö., Cleophas, L.: Using n-grams for the automated clustering of structural models. In: International Conference on Current Trends in Theory and Practice of Informatics, pp. 510–524. Springer (2017)
3. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T.: Microservices migration patterns. Softw. Pract. Exp. **48**(11), 2019–2042 (2018). https://doi.org/10.1002/spe.2608
4. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) Service-Oriented and Cloud Computing, pp. 19–33. Springer, Cham (2017)
5. Barriga, A., Rutle, A., Heldal, R.: Personalized and automatic model repairing using reinforcement learning. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 175–181. IEEE (2019)
6. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
7. Bucchiarone, A., Soysal, K., Guidi, C.: A model-driven approach towards automatic migration to microservices. In: Bruel, J.M., Mazzara, M., Meyer, B. (eds.) Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, pp. 15–36. Springer, Cham (2020)
8. Burgueño, L., Cabot, J., Gérard, S.: An lstm-based neural network architecture for model transformations. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 294–299. IEEE (2019)
9. Clarisó, R., Cabot, J.: Applying graph kernels to model-driven engineering problems. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, pp. 1–5 (2018)
10. De Alwis, A.A.C., Barros, A., Fidge, C., Polyvyanyy, A.: Remodularization analysis for microservice discovery using syntactic and semantic clustering. In: Dustdar, S., Yu, E., Salinesi, C., Rieu, D., Pant, V. (eds.) Advanced Information Systems Engineering, pp. 3–19. Springer, Cham (2020)
11. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
12. Escobar, D., Cárdenas, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., Casallas, R.: Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American Computing Conference (CLEI), pp. 1–11. IEEE (2016)
13. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, Boston (2003)
14. Fowler, S.J.: Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. O'Reilly Media Inc., Newton (2016)
15. Gouigoux, J.P., Tamzalit, D.: From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 62–65. IEEE (2017)
16. Gouigoux, J.P., Tamzalit, D.: "Functional-first" recommendations for beneficial microservices migration and integration lessons learned from an industrial experience. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 182–186. IEEE (2019)
17. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: a systematic approach to service decomposition. In: European Conference on Service-Oriented and Cloud Computing, pp. 185–200. Springer (2016)
18. Gysel, M., Kölbener, L.: Service cutter: a structured way to service decomposition. Bachelor's thesis, Department of Computer Science, University of Applied Sciences of Eastern Switzerland (HSR FHO), Rapperswil (2015)
19. Kapferer, S.: A Modeling Framework for Strategic Domain-driven Design and Service Decomposition. Master's thesis, Department of Computer Science, University of Applied Sciences of Eastern Switzerland (HSR FHO), Rapperswil (2020)
20. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html (2014). Accessed 24 Oct 2020
21. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Int. J. Open Inf. Technol. **2**(9), 24–27 (2014)
22. Newman, S.: Building Microservices. O'Reilly Media Inc., Newton (2015)
23. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: a machine learning approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 272–282. IEEE (2019)
24. OMG: Knowledge discovery meta-model specification version 1.3. (2011). http://www.omg.org/spec/KDM/1.3/PDF. Accessed 9 Feb 2022
25. Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., Silver, D.: Mastering atari, go, chess and shogi by planning with a learned model. Nature **588**, 604–609 (2020)
26. Selmadji, A., Seriai, A.D., Bouziane, H.L., Oumarou Mahamane, R., Zaragoza, P., Dony, C.: From monolithic architecture style to microservice one based on a semi-automatic approach. In: 2020 IEEE International Conference on Software Architecture (ICSA), pp. 157–168 (2020). https://doi.org/10.1109/ICSA47634.2020.00023
27. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018). https://doi.org/10.1126/science.aar6404
28. Sutton, R.S., Barto, A.: Reinforcement Learning: An Introduction. The MIT Press, Cambridge (2018)
29. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations and issues for migrating to microservices architectures: an empiri-

cal investigation. IEEE Cloud Comput. (2017). https://doi.org/10.1109/MCC.2017.4250931
30. Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., Luković, I.: Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. Enterp. Inf. Syst. **12**(8–9), 1034–1057 (2018). https://doi.org/10.1080/17517575.2018.1460766
31. Thönes, J.: Microservices. IEEE Softw. **32**(1), 116 (2015)
32. Wolff, E.: Microservices: Flexible Software Architecture. Addison-Wesley Professional, Boston (2016)
33. Zhu, Z., Lin, K., Zhou, J.: Transfer learning in deep reinforcement learning: a survey (2020)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

MohammadHadi Dehghani received his bachelor's and master's degrees in Computer Software Engineering in 2018 and 2021 from the University of Isfahan in Iran where he was a member of the Model Driven Software Engineering Research Group (MDSE). His current research interests include Model-Driven Software Engineering and Reinforcement Learning.

Shekoufeh Kolahdouz-Rahimi is an Assistant Professor in the Software Engineering Department at the University of Isfahan. She is an active member of the Model Driven Software Engineering Research Group (MDSE) at this University. She has completed her PhD in Computer Science at Kings College London in 2013. Her current research interests include model-driven software engineering and domain-specific languages.

Massimo Tisi is an associate professor in the Department of Computer Science of the Institut Mines-Telecom Atlantique (IMT Atlantique, Nantes, France), and deputy leader of the NaoMod team, LS2N (UMR CNRS 6004). Since 2019 he coordinates the Lowcomote Marie Curie European Training Network. He has been visiting researcher at McGill University and the National Institute of Informatics (NII) in Japan, and post-doctoral fellow at Inria. He received his PhD degree in Information Engineering at Politecnico di Milano (Italy), where he was a member of the Database and Web Technologies group. His research interests revolve around software and system modeling, domain-specific languages and applied logic. He contributes to the design of the ATL model-transformation language and investigates the application of deductive verification techniques to model-driven engineering.

Dalila Tamzalit received her Ph.D. in computer science at the university of Nantes in 2000. She is an assistant professor at the University of Nantes in France since 2001. Her main research interest concerns software evolution foundations and methodologies. She published several peer-reviewed articles on this research topic in international journals and conferences. In Tthese last years, she focuses on Software Architecture Evolution. In addition to chairing the series of international workshops on Model-Driven Software Evolution (MoDSE) since 2007, Dalila Tamzalit is the head of the French CNRS research group on evolution, reuse and traceability of Information Systems (ERTSI) and co-founder and co-organiser of its dedicated workshop since 2002. She has also been co-organizer, program committee member and reviewer for French and international symposia, workshops, conferences and journals on software engineering and software evolution. Dalila Tamzalit (dalila.tamzalit@univ-nantes.fr) LS2N, University of Nantes, France.