The Journal of Systems & Software 204 (2023) 111754

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Microservice architecture recovery based on intra-service and inter-service features☆

Wang Lulu [a], Hu Peng [a], Kong Xianglong [a], Ouyang Wenjie [a], Li Bixin [a,*], Xu Haixin [b], Shao Tao [b]

[a] School of Computer Science and Engineering, Southeast University, Nanjing, China
[b] SI-TECH Information Technology Co., Ltd, Beijing, China

ARTICLE INFO

Article history:
Received 15 July 2022
Received in revised form 9 April 2023
Accepted 22 May 2023
Available online 3 June 2023

Keywords:
Microservice architecture
Architecture recovery
System Dependency Graph
Reverse engineering
Software understanding

ABSTRACT

Microservice architecture supports independent development and deployment; it facilitates software system design and co-development. However, it also brings new challenges to a variety of software engineering tasks, especially in reverse engineering. An improper design or maintenance routine may cause complex invocation, obscure code logic, and complicate service layers, which may lead to difficulties in understanding, even further testing, or maintenance. To reduce the severity of this problem, we present a novel microservice architecture recovery technique that parses the source code to build a fine-grained dependency graph. This process recovers six key information components of the microservice architecture, which helps developers understand the system. Experimental results based on 12 projects show that the recovered accuracy is 94% on average. The results benefit any engineer unfamiliar with the project, increases their answering accuracy by 23.81% on average, and reduces their training time by 65.43% on average.

1. Introduction

Microservice architecture is a development approach for a single application in a suite of small services, each running its own process and communicating lightweight mechanisms, often an HTTP resource API (Lewis and Fowler, 2014).

In microservice architecture, each service is independently developed, deployed, upgraded, and scaled. Thus, it is particularly suitable for systems running on cloud infrastructures requiring frequent updating and scaling of their components (Zhou et al., 2018).

Microservice architecture's wide applicability is associated with a series of new problems. Wide-ranging remote calls increase testing and debugging difficulties. Convenient deployment leads to intra-service document neglect. Changes in agile teams, during long-term maintenance, creates a large gap between human understanding and program logic, which is even bigger than that in monolithic projects.

Architecture recovery is usually used to fill that gap on traditional software. It analyzes the source code, models the characteristics or features, extracts the detailed design or overall architecture, and helps developers and maintainers understand the system better. Pitifully, existing research mostly focuses on microservices' design, testing, deployment, verification etc., little about its architecture recovery (Zhou et al., 2018).

Traditional architecture recovery methods are mostly based on retrieval or clustering. They are not suitable for microservice projects: (1) The design and runtime relationships between services are different from those within a service. Therefore the inter-service information, including service interfaces and dependencies are significant to parsing the design information. (2) Traditional clustering process does not distinguish inter-service features from others, which may result in a cluster across services, and leads to confusing. (3) Some dependencies are defined as configurations, which cannot be mitigated using retrieval or clustering.

In this paper, we present a novel microservice architecture feature model that classifies recovery information and performs an automated recovery process using system source code and verifies if the recovered result is helpful for reverse engineering. Our work mainly contributes the following:

1. We present a Microservice Architecture Feature Model (MAFM) that satisfies reverse engineering requirements and consists of key design and deployment information for intra-service and inter-service levels.
2. We present the process of microservice architecture recovery, which uses the MAFM model, based on SSLDG, to

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail address: bx.li@seu.edu.cn (B. Li).

https://doi.org/10.1016/j.jss.2023.111754
0164-1212/

gradually recover the modules, components, services and corresponding dependencies and interfaces.
3. We implement the microservice architecture recovery technique, and carry out experiments on recovery accuracies, effectiveness and efficiency.

The rest of this paper is arranged as follows. Section 2 offers the related background. Section 3 describes the microservice architecture feature model MAFM. Section 4 presents how to recover architecture by SSLDG. Section 5 makes experimental evaluation. Section 6 summaries related works. Section 7 concludes this paper.

2. Background

2.1. Microservice and analysis

In 2014, Martin Fowler and James Lewis formally proposed the microservice concept (Lewis and Fowler, 2014). Generally, the more complex and flexible the business scenario, the more the advantageous microservice architecture becomes. Microservice architecture was quickly adopted by many enterprises and companies. In just a few years, more and more software products developed by internet enterprises are transforming from single architecture to microservice architecture. Neal Ford's "microservice maturity status" report released in 2018 showed that more than half of new projects of all respondents implemented microservice architecture (Ford, 2018). Microservices consist of many small services, each of which can be implemented by various technologies (Wolff, 2018). The services can also be called through a service registry (Wu, 2021). Compared with traditional single applications, each subservice is light, flexible and has strong fault tolerance. When the demand changes, only the relevant subservices need to be changed, and the whole system need not be greatly affected. The microservice architecture divides a complex problem into several small problems. Therefore, microservice architecture has better advantages than traditional architectures in terms of availability (Franklin, 1995) and adaptability (Xia and Wang, 2005).

Unlike traditional single architecture, which requires data from the whole business to provide services externally, microservice architecture divides the whole business into several business units according to a set of division rules that provide services externally. These independent business units are called subservices. Each subservice is a separate application with its own database and needs to be deployed separately. As the microservice architecture becomes more sophisticated, it can deal increasingly complex services. The emergence of subservices has introduces a set of new issues/problems, such as service registration, service discovery, service invocation (He et al., 2018), and service deployment. To solve these new problems, a complete microservice architecture solution needs to be gradually developed. The service itself provides solutions such as cluster (Parimalam and Sundaram, 2017) management, load balancing, service registration center, cloud configuration center, service fault tolerance (Lei and Dai, 2020; Rasheedh and Saradha, 2021) and other mechanisms to support service operations, log monitoring, monitor services, link tracking, and many other operations that can monitor the operation status of the service in real time. The use of service gateways (Akbulut and Perros, 2019) facilitates the use of services.

2.2. System Dependency Graph

The System Dependency Graph (SDG) of a program is a directed graph where the nodes are entities in source code (such as variables, statements, methods, classes) and the edges are the dependencies (such as data dependencies, control dependencies, invocations).

The directed graph has nodes and edges. The nodes and edges of the dependency graph are classified into different types to represent the context of the dependencies. In this way, as the slice gets smaller, it becomes more accurate.

2.2.1. Intra-procedural dependencies of PDG

A PDG (Program Dependency graph) combines all dependency relationships within each procedure. According to Ferrante (Ferrante et al., 1987), program dependencies can mainly be classified into two types, control dependencies and data dependencies. Control-dependent edges represent control conditions on which the execution of a statement or expression depends; data-dependent edges represent flow of data between statements, variables, or expressions.

Control Dependency (Ball and Horwitz, 1993): Let v and w be vertices in a CFG. Node w is directly L-control dependent on v iff w post-dominates the L-branch of v and w does not post-dominate v.

Control dependencies are computed from the control flow graph, which is irrelevant to other code characteristics.

Data Dependency (Orso et al., 2001): A data dependency is a triple (d; u; v) where d and u are statements and v is a variable; d defines v, u uses v, and there exists a path from d to u along which v is not defined.

Control and data dependencies are combined to build the PDG (Harrold et al., 2000). The definitions give a good description and computational approach for dependencies in the intra-procedural context (aliasing will be discussed in the next section); in the inter-procedural context, method interactions and object types bring other contents in data dependencies.

2.2.2. Inter-procedural dependencies of SDG

SDG additionally presents some types of dependencies based on following code features.

Polymorphism. Polymorphism, or dynamic binding, is a key characteristic of object-oriented languages, which permits an object to have different types in static view, and only one type is used during program execution to determine corresponding target.

In object-oriented programs, the access of member variables (or "domain variables") for an object is strictly determined statically, and only the access of methods are polymorphic.

Parameter. A parameter may be an object or a basic-type variable (such as Integer or Boolean), which means the parameter is a shallow copy or a deep copy. In SDG, there should be "Formal parameter in/out" nodes at a method declaration, and "Actual parameter in/out" at a method invocation statement. "Formal parameter in/out" and "Actual parameter in/out" nodes are connected, directly or indirectly, to realize the data flow for parameter passing on SDG.

Summary edge. It is used to decouple caller & callee methods (Horwitz et al., 1988): A summary edge connects an actual-in node and an actual-out node if the value associated with the actual-in node may affect the value associated with the actual-out node.

2.2.3. Other dependency graphs and SSLDG

Dependency graphs were introduced by Kuck (Kuck et al., 1981) as an intermediate program representation. Types of nodes and edges are invented to cover all dependency representations of programs.

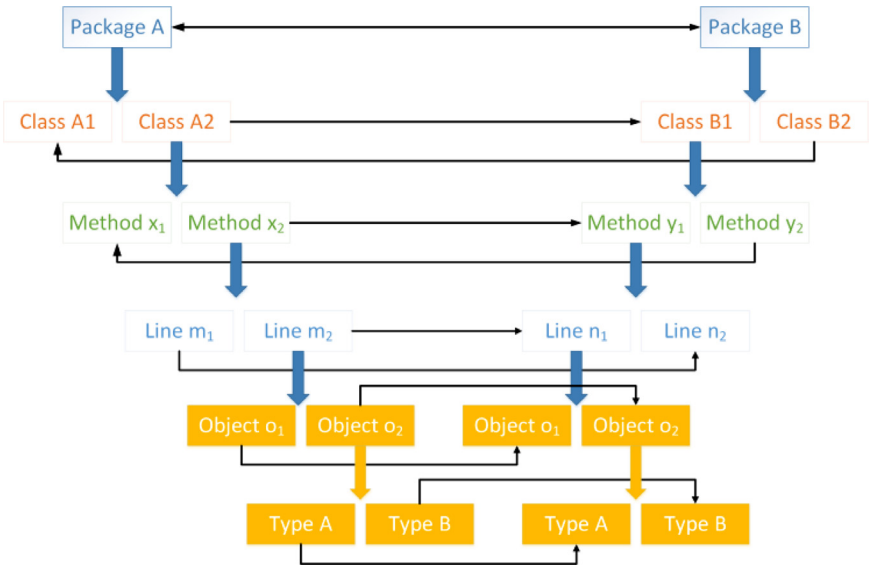

Fig. 1. Structure of SSLDG.

- **Hierarchy Dependency graph**: presented by Li (Li et al., 2004) which focus on different slicing grains (package/class/method/statement level), beneficial to software analysis (such as Panda Panda and Mohapatra, 2015).
- **System Dependency Net**: presented by Zhao (Zhao et al., 1996) for dependencies in concurrent object-oriented programs.
- **Process Dependency Net**: presented by Cheng (Cheng, 1993), which represents program dependencies in a concurrent procedural program with single procedure.
- **Multithreaded Dependency graphs**: presented by Zhao (Zhao, 1999) with dependencies for Java threads.

Extra work without a new graph name has been done to complete the dependency graph (Sinha et al., 1999; Larsen and Harrold, 1996; Horwitz et al.; Livadas and Johnson, 2000; Ranganath and Hatcliff, 2004), or to validate the graph grammar (Horwitz et al.).

SSLDG (Wang et al., 2020) focuses on how to accurate model dependencies in object-oriented programs. For the same object in static view, SSLDG separates its related dependencies of different polymorphic situations (so called types), as shown in Fig. 1.

1. Package. Package layers contain the dependencies between packages. Such dependencies have bidirectional influence. So, slicing based on this layer may only work when the dependency graph is not connected.
2. Class. Class layers contain the dependencies between classes and interfaces, including inheritance, association, and composites.
3. Method. Method layers contain the dependencies on method implementation and invocation.
4. Statement/Line. The dependencies of this layer are mainly control and data dependencies.
5. Sub-statement. This level decomposes the dependencies in upper layers. It mainly divides the statements into fragments by objects, and further divides the objects with possible polymorphic types; it establishes the dependencies between such types instead of dependencies between statements or other entities.

In the architecture recovery process, the entity recovery accompanies dependency recover. Dependency traversal is frequently used to update the relationships between newly generated nodes. SSLDG provides fine-grained dependencies and minimizes the inaccuracy caused by traversal, which is very similar to using it in the context of program slicing.

3. Microservice architecture feature model

When extracting microservice architecture information, it is often necessary to prioritize the information related to the microservice architecture evaluation. For example, in business-related code, information from each subservice does not need to be reviewed because the business function depends on the demand itself, and each microservice project will have completely different business implementations, so the code related to functional requirements can be regarded as architecture independent. You can choose to ignore the schema recovery. Therefore, this research starts from the characteristics of the microservice architecture, and compares it with the traditional single architecture, and finally divides the characteristics of the microservice architecture (see Fig. 2).

Intra-service features refer to some common features of each subservice, including *modules and their dependencies, components and their dependencies,* and *services.* When services are developed, it usually depends on many modules or packages, including local dependencies and third-party dependencies. To distinguish them, this paper refers to the third-party dependencies related to microservices as components, so that the information related to components as *component features.* Local public dependencies or other dependencies are named as modules and summarized as *module features*. *Component features* and *module features* belong to structural features within services.

Inter-service features refer to the topological relationship between services, including *service deployment*, *service interfaces* and *service dependencies.* The inter-service invocation relationship refers to the process that each subservice invokes other services or is invoked by other services. By extracting the inter service invocation relationship, the topology of the whole service invocation can be obtained. *Declared dependency* means the invocation information found in service configuration, while the *actual invocation* means those found by parsing the source code.

Interface information refers to the API information provided by each subservice itself. For example, the restful interface can be called by other services as a provider or directly provided to the gateway. It plays an important role in evaluating the scale and interface granularity of subservices. *Deployment information*

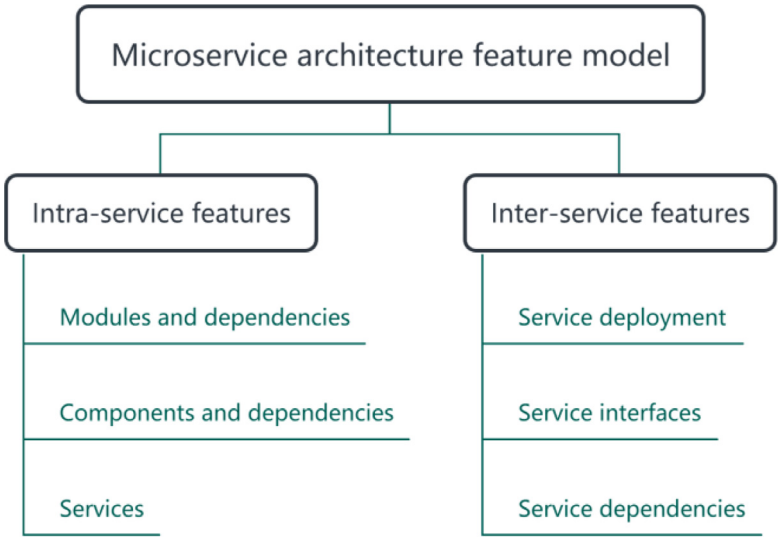

Fig. 2. Microservice architecture feature model.

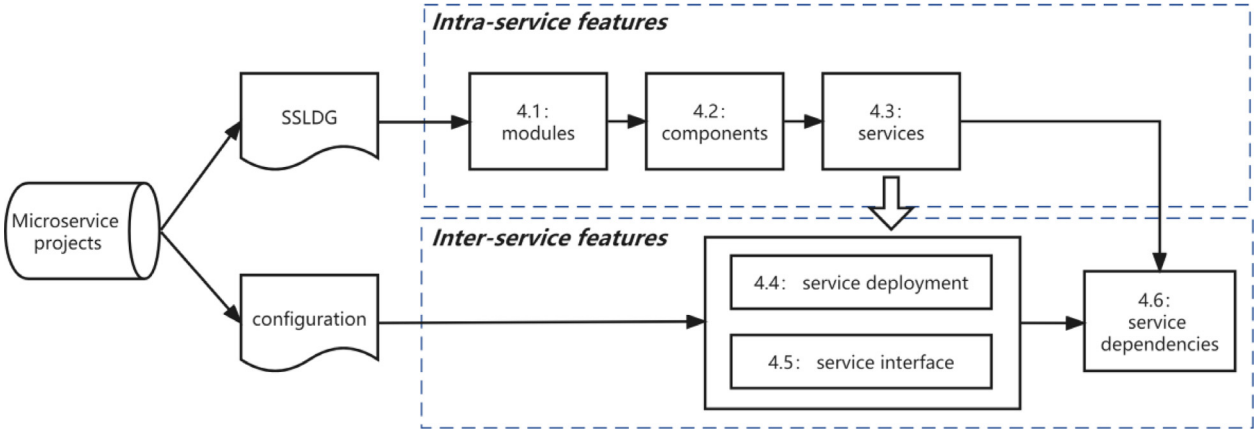

Fig. 3. Process of microservice architecture recovery.

records the configuration information of the subservice. By reading the configuration information, we can statically obtain some possible dynamic information of the service after it is deployed to the server, such as the registered address of the service. This information is conducive to analyzing the possible behavior of the service after deployment.

4. Microservice architecture recovery

The whole process of microservice architecture recovery is mainly based on the theoretical guidance of the microservice architecture feature representation tree to extract the architecture information corresponding to the features from the source code. The microservice architecture recovery process first needs to recover the subservices. That is, the whole microservice is divided into several subservices. Since the subservices are composed of several modules and components, it is necessary to recover the modules, component entities and dependent information of the microservice first. Then the subservice entities are recovered. This process belongs to the structural information recovery process of subservices. Second, the non-structural information of subservices, including service deployment information and service interface information, needs to be recovered. Finally, the relationship between services is recovered, mainly extracting the invocation relationship between services.

All the steps are descried in Fig. 3 and explained in the next sections in detail.

4.1. Module entity and dependency recovery

In the process of microservice project development, for a single subservice, the code organization mode usually adopts the multi module development mode, rather than putting all the code into a single module for development. One advantage of this is that the code organization mode can meet the design principles of high cohesion and low coupling. By extracting common code into common modules, many duplicate codes can be reduced and easy to maintain. Maven provides a complete multi module solution. It is also the most widely used and widely used multi module management tool at present. Therefore, this work mainly uses this tool to recover the dependency of microservice modules. Module entity and dependency recovery mainly includes three processes: the first is the identification process of module entity, i.e., to identify which parts of the target source code belong to the same module and list all modules of the project. The second is the identification of module dependency, i.e., to find out which local dependencies are introduced into each module. The third is the construction of module dependency graph, i.e., to build the overall module dependency graph according to the reference relationship of each module to other dependencies.

1. Module entity identification In Maven multi module management, modules have the following characteristics:

(1) The first level subdirectory of the directory where each module is located must contain the name configuration file (such as POM, NMP et al.);

(2) The configuration of each module records the type of the module, such as POM or jar;

(3) Configuration type indicates that the module is a parent module and contains several sub modules;

Based on the three characteristics, this research designs a module identification algorithm as shown in **Algorithm 1**. The algorithm is a recursive algorithm. The overall idea is to start from the project root directory and traverse the directory tree with the strategy of depth first traversal (DFS) with pruning. When traversing a directory, it is necessary to determine whether the sub-directories contain the configuration file. If it exists, it indicates that the directory is a module, and we record the module into the module list. Further we determine the type of the module by parsing the configuration file, and recursively traverse to find the sub module of the module. If it is a jar type, we prune without traversing the sub module. In this way, after traversal, the whole project can be divided into several sub modules. In addition, when the module is scanned during traversal, it is also necessary to record some metadata related to the module, including the Maven coordinates of the module, the path where the module is located, the class of the module, and other information.

Algorithm 1 Module identification

```
Input: Project path root
Output: Module list modules
Initialization: modules ← ∅
Procedure ResolveModule (root, modules)
Begin
 if (not Exists(root)) return
 if (IsDirectory(root)) then
      subPaths ← GetSubPaths(root)
      module ← null
      for each subPath of subPaths do
       if (GetName(subPath) equal configuration) then
        module ← ParseModule(root)
       endif
      if (module not equal null) then
       if (GetPackageType(module) equal configuration) then
        for each subPath of subPaths do
         ResolveModule(subPath, modules)
       endif
       modules ← modules ∪ module
      endif
 endif
End
```

2. Dependency identification After a project is divided into several modules, the dependencies between modules can be identified and resolved. First, we traverse the module list, and find the file path of the module according to the metadata information of the module. In the Maven multi module management mechanism, all dependencies of each module need to be registered in the module's configuration file, including local dependencies and third-party dependencies. Therefore, when parsing a module, you need to find the dependency list from the module's corresponding configuration file, and then find the existing local module from the dependency list and add it to the module's corresponding local dependency set. In this way, the dependency identification result can be obtained.

3. Build the module dependency diagram The module dependency graph is constructed based on the dependency information of each module on other modules. After the dependency identification process, the dependency set of each module on other local modules is obtained, which can be converted into a module dependency graph. The corresponding process is shown in Algorithm 2.

First, take the module list and the dependency information of each module as input. Note that the dependency information is stored in a two-dimensional array. The first subscript indicates which module is referred to, and the second subscript indicates the dependency list corresponding to the module.

Next, initialize a graph data structure, which includes a set of node lists and edges. The algorithm is a double loop. The outer loop traverses the module list. During the traversal, each module is incorporated into the node list, and the inner loop traverses the dependency list corresponding to the current module, while adding the edges composed of the current module and the dependent modules to the edge set. The direction of the edge is from the dependent module to the dependent module.

The whole module dependency graph can be constructed within the time complexity of O (N^2).

Algorithm 2 Module dependency graph construction

```
Input:
Module list modules[i]
Dependency of module dependency[i][]
Output:
Module dependency graph depGraph
Initialization:
depGraph[nodes] ← ∅
depGraph[links] ← ∅
Begin
 for i to sizeof modules do
      depGraph[nodes] ← depGraph[nodes] ∪ modules[i]
      for each dep of dependency[i] do
       depGraph[links] ← depGraph[links] ∪ Edge<module[i],
module[dep]>
End
```

4.2. Component entity and dependency recovery

Microservice component entity and dependency recovery is one of the important types of microservice architecture recovery. In this paper, component entity is defined as the third-party dependency introduced by the current project or module. In these components, we need to pay special attention to some components related to microservice. By recovering the entities and dependencies of the microservice component, the following information can be obtained.

1. Microservice component entity information introduced by the current module. By combining component dependency and module dependency, a complete microservice dependency graph can be constructed, which has a more comprehensive description of the characteristics of the service structure, and plays an important role in the subsequent understanding of the microservice code structure;

2. Microservice technology selection . At present, there are many solutions for any technical difficulty of microservice. For example, service load balancing can be implemented by multiple strategies, and components usually have a one-to-one correspondence with them. In this way, we can indirectly judge which strategy has been implemented by judging the component type;

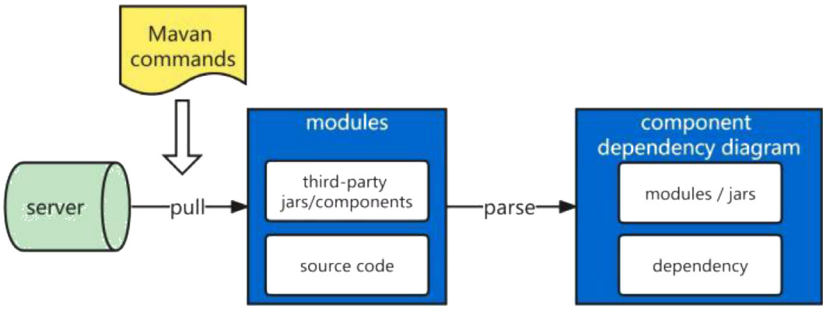

Fig. 4. Component dependency graph construction.

3. Current service type Some microservice components play a role in determining the service type in the current module. For example, the introduction of service registry components often indicates that the service is a service registry, while the introduction of service configuration center components indicates that the service is a configuration server.

There are mainly two technical difficulties in the microservice component entity and dependency recovery, one is how to obtain the component entity and dependency information from the source code, i.e., how to construct the component dependency graph. The other is what strategy to obtain the component entity information that needs attention, i.e., how to obtain the required information from the component dependency graph in the microservice component dependency recovery model. The observations are described in stages below.

4.2.1. Component dependency graph construction

When building a component dependency graph, you first need to locate the required information in the source code. Whether it is local dependency or third-party dependency, it can be obtained in the configuration of the module. However, when extracting the information from a third-party dependency or component, it is slightly different from a local dependency. In addition to the dependency information directly introduced in the configuration file, the component dependency graph also includes many indirectly introduced third-party dependencies. These dependencies are introduced indirectly through the dependencies declared directly in the configuration file. That is, via dependency transmission, which cannot be obtained directly from the configuration file. Therefore, it is necessary to dynamically pull these dependencies locally to obtain complete component information. Here we need to use the Maven invoker tool, which can execute Maven related commands in the environment of the project or module to be tested. In the process of building component dependency diagrams, we need to use the following three commands.

1. **MVN install.** This command is mainly used to package and install modules in the project, and pull remote dependencies to the local;
2. **MVN dependency: analyze.** This command can analyze the dependencies of the module and find the unused dependencies introduced by the module, which is conducive to the evaluation and optimization of the imported dependency list;
3. **MVN dependency: tree.** This command can display the dependencies declared in the configuration file and indirectly introduced dependencies in a tree structure, which is the key to building a component dependency graph.

Fig. 4 shows the construction principle of the component dependency diagram. The whole process can execute the commands in the Maven command set through an invoker under the current project or module. If a module under the current module is declared but not found, you need to send a request to the private server and pull the dependency locally, otherwise you can directly use the existing dependency. The private server is a remote warehouse that manages Maven dependency, it stores toolkits commonly used in development. After being pulled locally, all third-party dependencies or components of the module can be scanned, and the dependency graph of microservice components as shown in the figure can be easily constructed.

4.2.2. Component dependency recovery

Unlike local dependencies, there are usually many third-party dependencies in modules, which are reflected in the component dependency graph as more branch nodes and longer dependency chains. However, in fact, the evaluation model does not need to evaluate all components, but only needs to find some key components. Therefore, a search strategy is needed to filter the nodes in the graph. The components that need to be searched are usually those related to microservices, which is the principle of searching, but the commonly used microservice components are not static, but constantly changing. As time goes by, some new components will continue to be added, while the old components will be eliminated. Therefore, these factors need to be considered when designing component dependency recovery models.

To solve these problems, ***component dependency recovery*** takes the complete component dependency graph as the data source and uses the component registry and scanner to determine the recovery process of component information. The component registry mainly records the metadata of some components that need attention, including component coordinates, component types, versions, descriptive information. The design of a component registry solves the problem of uncertain information of microservice components, and the components that need to be found can be dynamically adjusted through configuration. We check and match the component information recorded in the component registry; the corresponding dependency will be added to the component dependency graph.

4.3. Service entity recovery

Service entity recovery refers to the integration of relevant modules belonging to the same subservice, which can also be regarded as the process of splitting the whole project into several subservices. In the mode of multi module development, a module is often a part of one or more subservices, which is not in one-to-one correspondence with the subservices; this will eventually lead to the inability to divide the subservices simply and intuitively. Therefore, a more accurate and effective method is needed to realize the recovery of service entities. Service entry detection technology refers to detecting the start of subservices or the location of the entry modules. This model provides a method to detect service entries based on static detection technology.

Service entry detection can detect the code fragment containing the main method. If the code block is found, the module

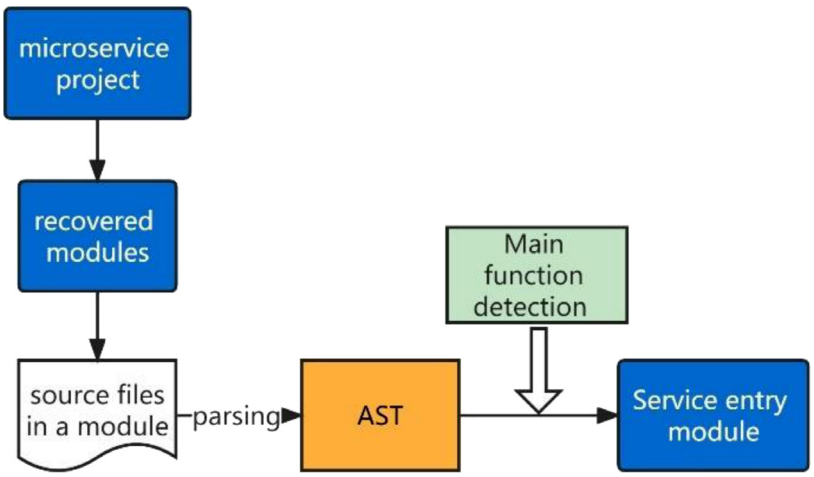

Fig. 5. Service entry module detection.

where the code is located is identified, indicating that the module is an entry module. Its identification principle is shown in Fig. 5. First, we scan the source code files of each module of the project (e.g. .java files), and then convert it into an abstract syntax tree. Second, we find the function node of the abstract syntax tree and carry out the verification process. The verification information includes function name, return value, parameters, annotations, and other information. If a module has a suitable main function, it should be an entry module. If a module does not have any entry files, this indicates that the module is not an entry module.

After identifying the entry module, the service entity recovery process can be carried out according to the module dependency diagram. Fig. 6 shows the results of subservice entity recovery. The model proposed in this paper can effectively and accurately divide the subservices of the microservice project. First, we mark the module nodes of the entry module into nodes of the microservice dependency graph, which are module A, module C and module F. Second, we take this node as the starting point, find all module nodes that directly or indirectly depend on the entry module, and take these nodes as a separate group. Thus, the module graph can be divided into several groups or areas according to the same number of searches as the entry module, These different areas correspond to different subservices, which is based on the fact that each service starts from the entry, and it will rely on many other modules in the development process. These modules are part of the service. In the example in the figure, the module dependency diagram is divided into three areas, namely, three subservices, and the service entry is marked respectively.

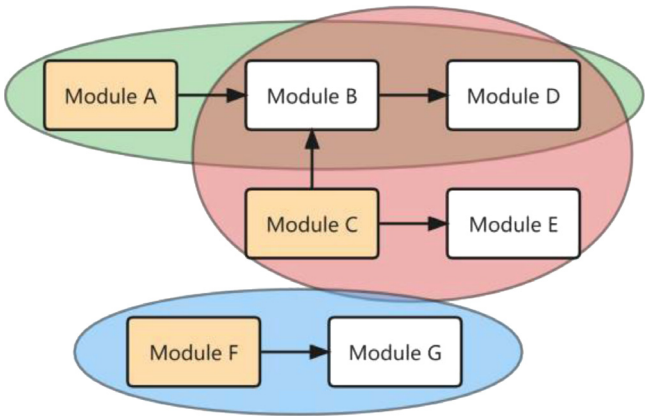

Fig. 6. Service entity recovery based on entry modules.

4.4. Service deployment information recovery

Service deployment information records some configuration information related to services, which plays an important role in obtaining service identification, service description and some dynamic information of services. To extract service deployment information, first, it is also necessary to prioritize the location of service deployment information in the source code. This information is mainly located in the resource directory of the module where the startup file is located. The code organization structure of the module generally includes directory and resource directory. The former is the storage directory of source code files, and the latter is the storage directory of resources and configuration files. According to the development specification of springboot, the microservice will read the application file by default after startup, so the deployment information is mainly stored in the configuration file.

In the process of deployment information parsing, we first need to study the principle of springboot reading configuration files, and then design the reading strategy based on the principle. At present, there are two main file forms including *properties* file and *yml* file. Therefore, different parsing modules need to be designed. At the same time, in addition to the default application file, springboot also reads other configuration files registered in the application. This configuration is determined by the profile field. For example, assuming that the profile field is *dev*, springboot not only needs to read the application file, but also need to read the application *dev* file. The specific service deployment information extraction process is shown in Fig. 7.

The service deployment information extraction diagram shows the whole extraction process, starting with the entry module. First, check the application of the module *yml* file or *properties* file, and then read its profile configuration information. If this information exists, further parse *application-{profile}.yml* file or *application-{profile}. properties* file. The whole parsing process is completed by the *yml* parser and the properties parser. Different parsers are used according to different file extensions. After parsing all files successfully, a merge operation is required. This operation merges all the detected file configurations. If there are conflicting configurations, the later detected configurations overwrite the first detected configurations, and finally complete deployment information can be obtained. For the deployment information itself, the information currently concerned mainly includes the service name, running port, context path, registration address. The service name refers to the name of the service

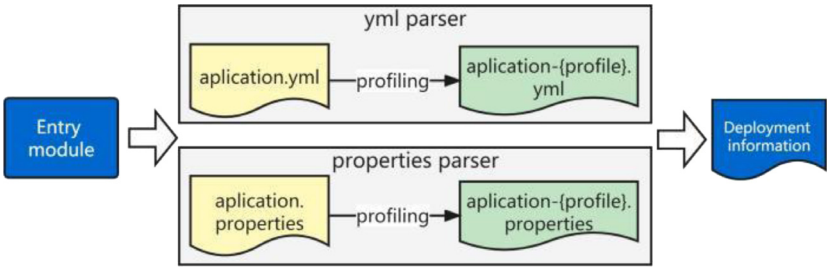

Fig. 7. Service deployment information extraction.

registered in the registry, which is the unique identification of the service. The running port and context path are the common request path prefix for accessing the API provided by the subservice. By combining the identification results of the service interface information, we can get the complete request path of an interface, and the registered address indicates the IP address of the registry to which the service is currently registered. Thus, we can find the relationship between the service and the service registry. This deployment information describes the service in the form of metadata and describes the state of the service itself in detail.

4.5. Service interface information recovery

Service interface information refers to the API information provided by the service externally, which is like the traditional web server. The client can call the interface provided by the service through remote call or sending HTTP request. The difference is that in the microservice architecture, the subservice usually does not directly expose these APIs to the client but provides them to other service calls as a service provider, The API interface that the client needs to use is provided by the service gateway, which is also one of the reasons why the microservice architecture is highly available. At present, from the perspective of the development process of a single subservice, most of its development is based on the traditional MVC three-tier architecture. That is, the view layer, the business layer, and the model layer. Springboot integrates the common development components in MVC architecture and is a very popular development framework at present. Based on the architecture specification, all interface information is defined by the controller class, and the request method of the interface is based on HTTP protocol. In this way, the goal of the whole service interface information extraction process is to extract the API interface information list of each subservice, and the interface information of each subservice is in the location defined by the controller class in the source code.

We extract the interface information based on the parsing of the abstract syntax tree.

1. AST generation. Like the deployment information extraction, the first step starts from the entry module, traverses the source code files in this module, and then converts the source code into the abstract syntax tree.
2. Controller filtering. It is necessary to detect whether a class belongs to the controller class. The principle of detection is to parse the annotation information on the class in the abstract syntax tree. If the annotation is *RestController*, then the class should be a controller class, otherwise, this class will be filtered out.
3. Interface information parsing. The interface information to be parsed includes the **request type**, **request path**, **request parameters**, **return parameters** and other information of the interface, which can completely describe the interface. Due to the flexibility of code writing, the extracted information categories often do not have a one-to-one correspondence to the way the code is written. For example, in the process of identifying the request type, there are two different ways to set the request type, *ReqquestMapping* and *PostMapping* annotations. Therefore, we need to consider different implementations so the recovered result is reliable.

In addition, while identifying the request path, it should be noted that for an interface, its path not only exists in the annotation corresponding to the method, but also has a relationship with the annotation information on the class. Its complete path is a combination of the path information annotated on the class and the path information annotated on the method. Therefore, when reading each interface information item, it is also necessary to fully understand the principle of the framework that implements the interface, so that the accuracy and completeness of the interface information identification can be guaranteed.

4.6. Service dependency recovery

Service invocation relationships are a very important part in the recovery process of microservice architecture. In the previous section, the information extraction work for microservice architecture is mostly to analyze the internal information and structure of a single subservice. From a macro perspective, the result of the analysis is that services are isolated nodes, and there is no inevitable connection between services, but in fact, this is not the case; there are two important characteristics that distinguish microservice architecture from other architectures. One is that it is composed of many subservices, and the other is the relationship network between subservices. These topological relationships may be simple or complex, but they are crucial in the recovery process of microservice architecture. By extracting and restoring the invocation relationship, we can get a complete topology map of the microservice architecture.

When extracting the service invocation relationship, it is necessary to clarify how to describe and characterize the relationship. Starting from the service invocation process and invocation principle, this paper summarizes the single invocation process into three focuses: (1) how to judge which service the caller or initiator of this invocation process is, (2) how to judge which service the callee or provider of this invocation process is, (3) which implementation way is used to realize the call of services. Since a single service is one-way, the caller is set to a and the callee is set to b, so the final call relationship can be represented by a binary in the form of <a, b>.

There are many ways to invoke services, such as HTTP or RPC. When we study service interface information, we are mainly using an HTTP call protocol; the same is true of the extraction process of a call relationship. Unlike calling directly through URL, calls between services often encounter more complex problems, such determining the load balancing strategy of service calls, so it is usually necessary to provide a more perfect call mechanism.

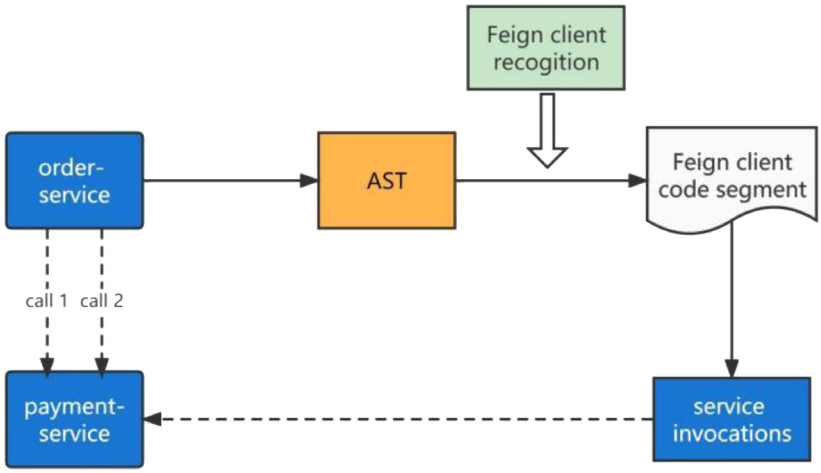

Fig. 8. Service invocation extraction.

FeignClient, a service call component that realizes this function, is provided in *SpringCloud*. By using *FeignClient*, calls between services can be easily realized without paying attention to the details of the service calls. This is a commonly used service invocation solution in enterprise development. We will extract *FeignClient* code segment from the code and analyze the details of service invocation based on this code segment.

Fig. 8 shows the principle of extracting a service invocation. It is assumed that there are two subservices, *order-service,* and *payment-service*, where the former is responsible for the management of orders, and the latter is responsible for the management of account balances. *Order-service* needs to call the payment service to complete the payment process. According to the three core concerns mentioned before:

(1) We need to determine the initiator of the service. Because the initiator invokes the service through *FeignClient*, we need to find the code block related to *FeignClient* at the code level, extract the code block of *FeignClient*, and traverse each subservice from the entry module. When the target code block is retrieved, it means that the service where the module is located is the service initiator, which is the service caller.

(2) We analyze the service calling component *FeignClient*, and its abstract syntax tree, as shown in the figure. The service name of the callee can be found through annotation, and the specific service interface of the callee can be found through parsing method.

(3) The service interface called in *FeignClient* should be a subset of the service interface information of the called service, and the complete service interface information of each subservice can be extracted through the method mentioned in Chapter 4.5.

In this way, through the above process, we can get the call relationship of *order-service* and *payment-service*, which is a single service invocation. By repeating this process, we can find all the service call relationships and build the service call relationship topology network.

5. Experiment

This section presents an empirical evaluation of the performance (efficiency) of microservice architecture recovery.

5.1. Environment and benchmarks

Our experiment is performed in the following environment: the opensource projects are analyzed with IBM x3850 x6 32 GB RAM, and SI-TECH projects are analyzed with SI-TECH devops system with 128 GB RAM.

We queried "microservice" on opensource community and selected the projects with the most stars. We manually checked if the projects are typical or not (for example, some are student exercises or some consist of toy programs). We selected nine open-source microservice projects from Git.com and Gitee.com. We also selected three projects from SI-TECH Co., which were created by engineers with more expert knowledge and provide detailed notes.

The details for all twelve projects are listed in Table 1. The details are used as the input for the target projects of our recovery technique.

5.2. Research question

The microservice architecture recovery result is used to help developers understand the logical structure of the system. Its quality should be quantified based on accuracy and simplicity of use. Thus, our experiment focuses on following research questions:

Q1: Is the recovered microservice architecture accurate?

We assume the SI-TECH projects conform with company design decisions. So, the design documents are taken as the ground truth of the architecture to measure the quality of recovery result using MoJoSim. We also implement a new recovery technique DDP (directory-based dependency processing recovery Kong et al., 2018) for comparison.

Q2: Can the recovered microservice architecture help improve human understanding the system?

For each benchmark: one expert **carefully** understands the benchmark based on source code, online documents, comments, or developed information (from SI-TECH Co.), then designs **seven** questions; two other experts **quickly** understand the project and answer the questions with and without the recovered architecture; the correctness of their answers are used to measure how much recovered architecture is useful.

Q3: Can the recovered microservice architecture help speed up human understanding of the system?

For the same questions in Part 2, the time it takes to answer questions is used to measure how much recovered architecture can speed up a worker's understanding.

5.3. Results

5.3.1. Results on Q1

MoJoSim is used in the experiment to measure the effectiveness of the microservice architecture recovery, which is a

Table 1
Microservice benchmarks.

	Project owner/name	Stars	#files	#LOC
Open-source	toopoo/SpringCloud	1.1k	368	19,241
	smallc/SpringBlade	16.3k	360	2,034,756
	durcframework/SOP	1.6k	1355	98,233
	zlt2000/microservices-platform	3.6k	860	2,037,356
	moxi624/mogu_blog_v2	1k	4028	804,078
	dafanshu/bigfans-cloud	0.5k	972	46,864
	macrozheng/mall-swarm	8.8k	745	156,392
	lengleng/pig	33.2k	833	2,323,258
	wenMN1994/Xueyuan	0.1k	148	32,937
SI-TECH	auto-platfrom@STQ	–	478	328,081
	multi-task@STQ	–	496	328,769
	ci-platform@STQ	–	1701	217,641
Total			12,344	8,427,606

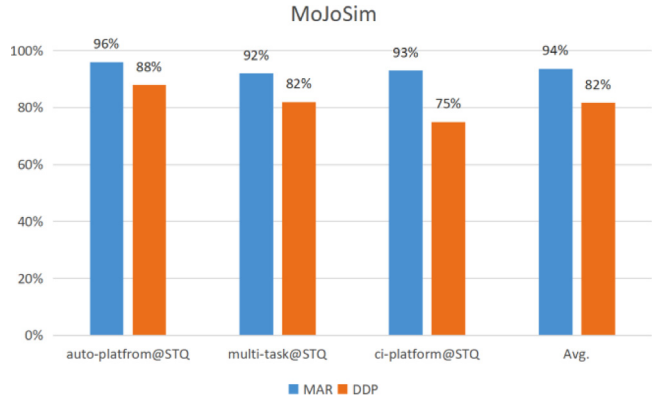

Fig. 9. Recovery accuracy testing result.

widely used index to measure the effectiveness of architecture recovery technology (Wu et al., 2005; Lutellier et al., 2015, 2018; Bazylevych and Burtnyk, 2015; Bittencourt and Guerrero, 2009). This index can measure the similarity between the restored software architecture and the real architecture. The higher the similarity, the more effective the architecture recovery method adopted:

$$MoJoSim(A, B) = (1 - \frac{mno(A, B)}{n}) \times 100\%$$

where "A" represents the restored software architecture, "B" represents the actual architecture of the software, and "$mno(A, B)$" is the minimum number of move/join required to transform A into B; "n" is the number of code entities in the software architecture. When the MoJoSim value is 100%, this indicates that the restored software architecture is completely consistent with the real software architecture. If the MoJoSim value is 0%, the two are completely different.

The recovery results on three SI-TECH projects are measured with MoJoSim based the ground-truth (for example, the ground-truth architecture of ci-platform@STQ is shown in Appendix A) from the software design document and compared with DDP as shown in Fig. 9.

5.3.2. Results on Q2

As explained in the experimental steps, experts are required to design questions based on their understanding of the microservice project; this evaluates the consistency in understanding the same project correctly. Based on typical scenarios in software maintenance such as debugging, refactoring and work assignment, four typical questions are set to test one's understanding of a project:

Table 2
Questions of benchmarks.

Project owner/name	WH	HM	CP	EX
toopoo/SpringCloud	5	1	1	0
smallc/SpringBlade	4	2	0	1
durcframework/SOP	5	1	1	0
zlt2000/microservices-platform	3	0	0	4
moxi624/mogu_blog_v2	4	1	0	2
dafanshu/bigfans-cloud	5	1	1	0
macrozheng/mall-swarm	1	2	0	4
lengleng/pig	4	3	0	0
wenMN1994/Xueyuan	2	1	0	4
auto-platfrom@STQ	2	1	0	4
multi-task@STQ	2	2	0	3
ci-platform@STQ	3	2	1	1
Total	40	17	4	23

1. **WHICH question** (WH for short). The questions asks the expert to choose one correct target, such as "**Which** of the following services is called remotely by the **consumer-feign** service in project Springcloud?"
2. **HOW MANY** (HM) **questions**. Asks the expert to give correct number, such as "**How many interfaces** does the **consumer-feign** service provide in project Springcloud?"
3. **COMPARSION** (CP) **question**. Asks the expert to compare options and select the extreme value, such as "**Which** of the following microservices has the **most** controllers?"
4. **EXCLUSIVE** (EX) **question**. Asks the expert to eliminate the wrong choice, such as "Which of the following statements about **business-service** is **wrong** in the project Microservice-platform?"

After all 84 questions are designed (seven for each benchmark), their types are listed in Table 2.

We check the answers from two experts who quickly understand the system independently, and compute their accuracy, as reported in Fig. 10, where *Accuracy* represents correctness **without** recovery results and *Improvement* represents degree **with** recovery results.

Fig. 10 reports that in 8/12 projects there is obvious improvement, from 14.29% to 57.14%, and in Project 8, 9, 11 and 12, both experts answer all questions right. Overall average improvement is 23.81%.

This test analyzed if the accuracy is scale dependent. Fig. 11 shows the improvement, where the X-axis is the number of files in each benchmark project, the Y-axis is the LOC in each benchmark project, and the bubble size is the improvement from Fig. 10.

Fig. 11 reports that it is not true that the larger the project is, the better the improvement should be.

Furthermore, we separate the results on different question types. The statistical result is shown in Fig. 12, where:

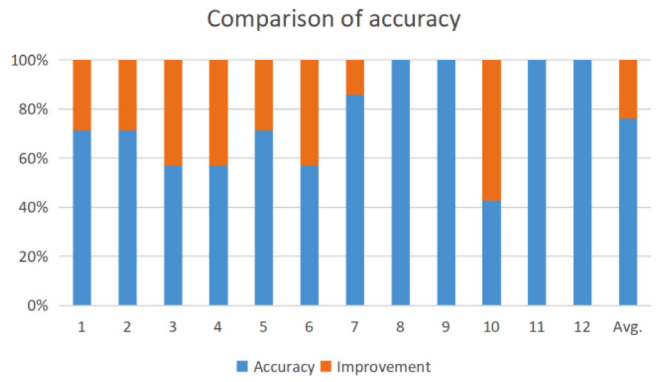

Fig. 10. Results of accuracy with/without recovery results.

Fig. 11. Statistical results of improvement on project scale.

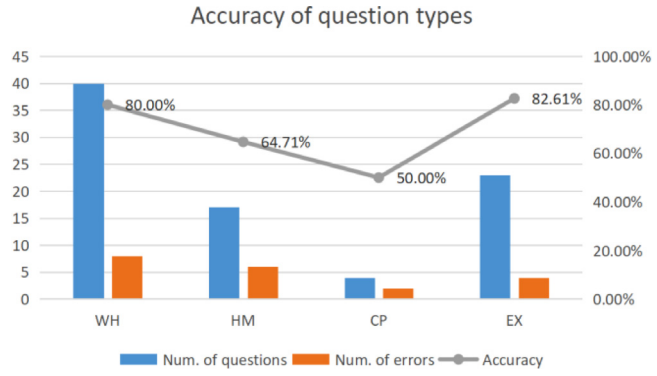

Fig. 12. Statistical results of improvement on question types.

Num. of questions: the number of all questions of corresponding type

Num. of errors: the number of questions answered wrong of corresponding type

Accuracy: the proportion of questions answered right to *Num. of questions*

Fig. 12 reports that the WH and EX questions are easy to answer, while the CP questions are the most difficult.

5.3.3. Results on Q3

Also, in the process of Part 2, we also record the time it takes for an expert to answer questions (temporal cost). Fig. 13 reports that recovery results also help the experts find key information to provide the answers, reduces the temporal cost to at least 54.93% and at most 85.39%, and 65.43% on average.

Like Part 2, we also separate the results of different question types. The statistical results are shown in Fig. 14, which reports

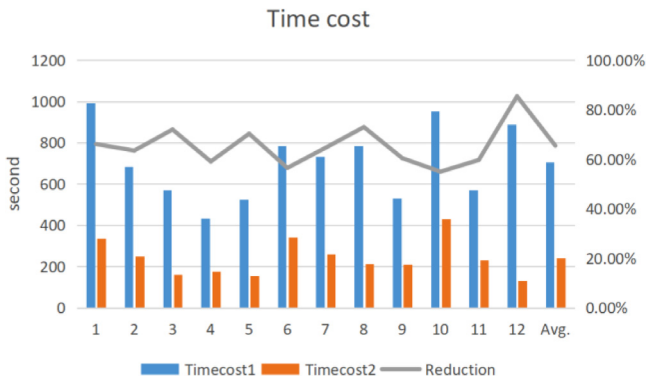

Fig. 13. Time cost with/without recovered results.

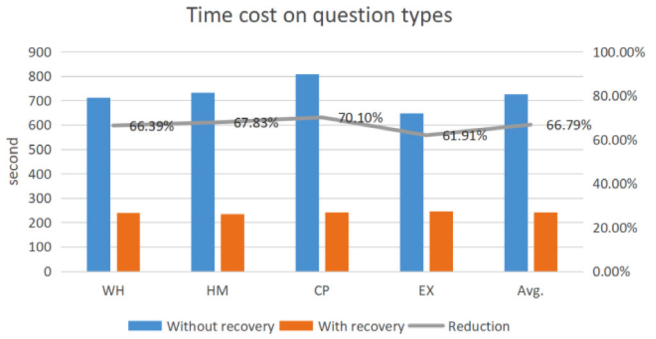

Fig. 14. Statistical results of time cost on question types.

that the reduction rates are roughly similar for different question types.

5.4. Conclusion and discussion

The experimental results demonstrate that:

1. **Q1:** Microservice design architecture provides a benefit to those unfamiliar with the project. This is validated by an increase expert answering accuracy (23.81% on average) and a reduction in learning rate (65.43% on average).
2. **Q2:** The accuracy improvement is seen on most projects (8/12), while the efficiency improvement is seen on all projects.
3. **Q3:** Even if the recovered result does not help an expert understand simple projects anymore, the time spent in looking up the result is still cost-effective.

We also have following discussions based on interesting observations:

Fig. 11 reports that the project scale is oddly not obviously related to improvement results. One possible reason is that large-scale projects may not have a complex design. In our experiment, the experts are instructed not to overly complicate the problem.

For question types, we thought "EX" questions would be the hardest to answer, but it turns out to not be the case. Some tricks other than actual understanding could help answer such questions.

6. Related work

Traditional single architectures require architecture recovery. The recovery techniques could be mainly classified into two categories:

Structure-based architecture recovery techniques. Bunch (Mamaghani and Meybodi, 2009) is a clustering approach that optimizes the objective features according to an optimization function called Modularization Quality (MQ). Bunch uses two kinds of hill climbing algorithms to resolve the optimization problem, i.e., nearest and steepest ascent hill climbing (NAHC and SAHC). ACDC (Tzerpos and Holt, 2000) is a pattern-based approach that groups code entities according to a developers' description of the components of a software system. LIMBO (Andritsos et al., 2004) is a scalable hierarchical clustering approach that quantifies the relevant information preserved based on the information bottleneck framework. The weighted combined algorithm (Maqbool and Babri, 2004) is a hierarchical clustering approach that groups code entities according to inter-cluster distance. Structure-based software architecture recovery techniques usually consist of two parts (i.e., extraction and grouping of structural information) and are easily automated. But there are still some structure-based techniques that involve varying degrees of manual work; we categorize these into semiautomatic structure-based software architecture recovery techniques. Many modern techniques are typically automatic techniques. Bowman et al. obtain conceptual architecture manually and use it to improve concrete architecture (Bowman et al., 1999). Focus (Ding and Medvidovic, 2001) applies manual work to generate idealized architecture evolution and uses it to address affected components. Tool-assisted recovery techniques are frequently used as semiautomatic architecture recovery techniques; they are usually used to obtain ground-truth architectures. The tools can help users to extract and visualize software dependencies, e.g., Rigi (Storey et al., 1997) and AOVis (Koch and Cooper, 2011). Since all these techniques start from a specific level dependency graph, and our approach can produce a sub module-level dependency graph, our approach has the potential to improve both automatic and semi-automatic structure-based software architecture recovery techniques.

Knowledge-based architecture recovery techniques. Kuhn et al. (2007) proposed a Latent Semantic Indexing based clustering approach that groups code files containing similar terms in the comments. Garcia et al. (2011) proposed a machine learning based technique that can identify components and connectors according to a variety of concerns. It considers a program as a set of textual information and extracts concerns based on the Latent Dirichlet Allocation (LDA) model. Corazza et al. (2011) proposed a nature language processing-based clustering approach that divides code files into different zones. The zones are weighted based on an Expectation–Maximization algorithm, and then grouped by a Hierarchical Agglomerative Clustering technique.

In addition, there is also some recent work close to the topic of **microservice architecture recovery**. Alshuqayran et al. (2018) developed support for Microservice Architecture Recovery (MiSAR) using a Model Driven Engineering approach. They have defined a meta model for microservice-based architectures and a set of mapping rules which map between the software and the meta model. Granchelli et al. (2017) present the first prototype of our Architecture Recovery Tool for microservice-based systems called MicroART. MicroART following Model-Driven Engineering principles; it can generate models of the software architecture of a microservice-based system, which can be managed by software architects for multiple purposes.

Our work is based on source code analysis, provides recovered result automatically, and does not rely on UML models,

7. Conclusion and future work

Microservice architecture is very suitable to *devops* platforms and has been widely used. Microservice architecture facilitates cooperation in a distributed team, such as co-design, co-programming, and co-testing. However, in long-term maintenance projects, such a convenience encourages quick tasks and ultimately results in an unwieldy system. Job attrition rates inevitably creates the need for reverse understanding of a multi-person collaborative project.

Based on analyzing the requirement of understanding how the services relate to each other in a system, we present a novel microservice architecture feature model to classify the recovery information and perform the recovery process to achieve the goal of automated microservice architecture recovery from system source code, and verify if the recovered result is helpful in reverse engineering.

Our recovery technique still has much to improve:

Not all dependencies are closely related to microservice understanding. Different types of dependencies could be treated differently. In this way, a smarter microservice architecture model could be found and used.

In addition to the projects in our experiment, more microservice systems could be tested with new ways to test the degree of understanding (other than expert questions).

Microservice benchmarks should be checked for suitability; this will allow for influence factor analysis, which is not found in our experiment yet.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

This work was supported by National Natural Science Foundation of China (Grant Nos. 61872078, 61402103), the National Key Research and Development Program of China under Grant 2019YFE0105500, the Research Council of Norway under Grant 309494, the Key Research and Development Program of Jiangsu Province under Grant BE2021002-3, and SI-TECH Information Technology Co.,Ltd.

References

Akbulut, A., Perros, H.G., 2019. Software versioning with microservices through the API gateway design pattern. In: 2019 9th International Conference on Advanced Computer Information Technologies. ACIT, pp. 289–292.

Alshuqayran, N., Ali, N., Evans, R., 2018. Towards microservice architecture recovery: An empirical study. In: IEEE International Conference on Software Architecture 2018. IEEE.

Andritsos, P., Tsaparas, P., Miller, R.J., Sevcik, K.C., 2004. LIMBO: Scalable clustering of categorical data. In: International Conference on Extending Database Technology. pp. 183–199.

Ball, T., Horwitz, S., 1993. Slicing programs with arbitrary control-flow. In: Automated and Algorithmic Debugging. Springer, Berlin Heidelberg, pp. 206–222.

Bazylevych, R., Burtnyk, R., 2015. Algorithms for software clustering and modularization. In: Xth Int Scientifific and Technical Conf Computer Sciences and Information Technologies. pp. 30–33.

Bittencourt, R.A., Guerrero, D.D.S., 2009. Comparison of graph clustering algorithms for recovering software architecture module views. In: 13th European Conf on Software Maintenance and Reengineering. pp. 251–254.

Bowman, I.T., Holt, R.C., Brewster, N.V., 1999. Linux as a case study: Its extracted software architecture. In: ACM International Conference on Software Engineering. pp. 555–563.
Cheng, J., 1993. Process dependence net of distributed programs and its applications in development of distributed systems, computer software and applications conference. In: 1993. COMPSAC 93. Proceedings. Seventeenth International. pp. 231–240.
Corazza, A., Di Martino, S., Maggio, V., Scanniello, G., 2011. Investigating the use of lexical information for software system clustering. In: Proceedings of the European Conference on Software Maintenance and Reengineering. pp. 35–44.
Ding, L., Medvidovic, N., 2001. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In: Working IEEE/IFIP Conference on Software Architecture. pp. 191–200.
Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependency graph and its use in optimization. ACM Trans. Program. Lang. Syst..
Ford, N., 2018. The state of microservices maturity.
Franklin, P.H., 1995. An analysis of system level software availability during test. In: International Symposium on Software Reliability Engineering. IEEE, pp. 360–365.
Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., Cai, Y., 2011. Enhancing architectural recovery using concerns. In: International Conference on Automated Software Engineering. pp. 552–555.
Granchelli, G., Cardarelli, M., Francesco, P.D., et al., 2017. MicroART: A software architecture recovery tool for maintaining microservice-based systems. In: IEEE International Conference on Software Architecture Workshops. IEEE.
Harrold, M.J., Malloy, B., Rothermel, G., 2000. Efficient construction of program dependency graphs. Acm Sigsoft Softw. Eng. Notes 18 (3), 160–170.
He, D., Xue, Y., Feng, Z., et al., 2018. A probabilistic model for service clustering - Jointly using service invocation and service characteristics. In: 2018 IEEE International Conference on Web Services. ICWS, IEEE, pp. 302–305.
Horwitz, S., Prins, J., Reps, T., On the adequacy of program dependency graphs for representing programs. In: POPL 1998: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages. pp. 146–157.
Horwitz, S., Reps, T., Binkley, D., 1988. Interprocedural slicing using dependency graphs. In: PLDI 88: ACM Sigplan Conference on Programming Language Design & Implementation. pp. 35–46.
Horwitz, Susan, Thomas, The use of program dependency graphs in software engineering. In: ICSE 1992: Proceedings of the 14th International Conference on Software Engineering, Vol. 31, no. 14. pp. 392–411.
Koch, J., Cooper, K., 2011. AOVis: A model-driven multiple-graph approach to program fact extraction for AspectJ/Java source code. Softw. Eng.: Int. J. 60–71.
Kong, X., Li, B., Wang, L., et al., 2018. Directory-based dependency processing for software architecture recovery. IEEE Access PP, 1.
Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M., 1981. Dependency graphs and compiler optimizations. In: ACM Sigplan-Sigact Symposium on Principles of Programming Languages. pp. 207–218.
Kuhn, A., Ducasse, S., Gîrba, T., 2007. Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. 49 (3), 230–243.
Larsen, L., Harrold, M.J., 1996. Slicing object-oriented software. In: International Conference on Software Engineering. IEEE, pp. 495–505.
Lei, C., Dai, H., 2020. A heuristic services binding algorithm to improve fault-tolerance in microservice based edge computing architecture. In: 2020 IEEE World Congress on Services. SERVICES, IEEE, pp. 83–88.
Lewis, J., Fowler, M., 2014. Microservices a definition of this new architectural term. [Online]. Available: http://martinfowler.com/articles/microservices.html.
Li, B.X., Fan, X.C., Pang, J., Zhao, J.J., 2004. A model for slicing Java programs hierarchically. J. Comput. Sci. Tech. 19 (6), 848–858.
Livadas, P.E., Johnson, T., 2000. An optimal algorithm for the construction of the system dependency graph. Inform. Sci. 125 (1–4), 99–131.
Lutellier, T., Chollak, D., Garcia, J., et al., 2015. Comparing software architecture recovery techniques using accurate dependencies. In: IEEE/ACM 37th IEEE Int Conf on Software Engineering. pp. 69–78.
Lutellier, T., Chollak, D., Garcia, J., et al., 2018. Measuring the impact of code dependencies on software architecture recovery techniques. IEEE Trans. Softw. Eng. 44 (2), 159–181.
Mamaghani, A.S., Meybodi, M.R., 2009. Clustering of software systems using new hybrid algorithms. In: IEEE International Conference on Computer & Information Technology. IEEE, pp. 20–25.
Maqbool, O., Babri, H., 2004. The weighted combined algorithm: A linkage algorithm for software clustering. In: European Conference on Software Maintenance and Reengineering. pp. 15–24.
Orso, A., Sinha, S., Harrold, M.J., 2001. Incremental slicing based on data-dependencies types. pp. 158–167.
Panda, S., Mohapatra, D.P., 2015. ACCo: A novel approach to measure cohesion using hierarchical slicing of Java programs. Innov. Syst. Softw. Eng. 11 (4), 243–260.
Parimalam, T., Sundaram, K.M., 2017. Efficient clustering techniques for web services clustering. In: 2017 IEEE International Conference on Computational Intelligence and Computing Research. ICCIC, IEEE, pp. 1–4.
Ranganath, V.P., Hatcliff, J., 2004. Pruning interference and ready dependence for slicing concurrent Java programs. In: Compiler Construction. Springer, Berlin Heidelberg, pp. 39–56.
Rasheedh, J.A., Saradha, S., 2021. Reactive microservices architecture using a framework of fault tolerance mechanisms. In: 2021 Second International Conference on Electronics and Sustainable Communication Systems. ICESC, pp. 146–150.
Sinha, S., Harrold, M.J., Rothermel, G., 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In: International Conference on Software Engineering. ACM, pp. 432–441.
Storey, M.A.D., Wong, K., Muller, H.A., 1997. Rigi: A visualization environment for reverse engineering. In: International Conference on Software Engineering. pp. 606–607.
Tzerpos, V., Holt, R.C., 2000. ACDC: An algorithm for comprehension-driven clustering. In: Proceedings of the Seventh Working Conference on Reverse Engineering. WCRE'00, IEEE.
Wang, L., Li, B., Kong, X., 2020. Type slicing: An accurate object oriented slicing based on sub-statement level dependency graph. Inf. Softw. Technol. 127, 1–16.
Wolff, E., 2018. Microservices - A practical guide.
Wu, X.J., 2021. Design and development of service registry in microservice framework. Ind. Control Comput. 34 (08), 130–132.
Wu, J., Hassan, A., Holt, R., 2005. Comparison of clustering algorithms in the context of software evolution. In: 21st IEEE International Conference on Software Maintenance, Budapest, Hungary. pp. 525–535.
Xia, L., Wang, Q., 2005. Study on application of a quantitative evaluation approach for software architecture adaptability. In: Quality Software, 2005. (QSIC 2005). Fifth International Conference on.. IEEE, pp. 265–272.
Zhao, J., 1999. Multithreaded dependency graphs for concurrent Java programs, software engineering for parallel and distributed systems. In: 1999. Proceedings. International Symposium on. IEEE, pp. 13–23.
Zhao, J., Cheng, J., Ushijim, K., 1996. Static slicing of concurrent object-oriented programs. In: COMPSAC'96: 20th Computer Software and Applications Conference. pp. 312–320.
Zhou, X., Peng, X., Xie, T., et al., 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. Softw. Eng. 243–260.