Future Generation Computer Systems 100 (2019) 724–735

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Graph-based and scenario-driven microservice analysis, retrieval, and testing

Shang-Pin Ma [a,*], Chen-Yuan Fan [a], Yen Chuang [a], I-Hsiu Liu [a], Ci-Wei Lan [b]

[a] Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan
[b] Taiwan IBM, Taipei, Taiwan

HIGHLIGHTS

- We introduce a novel scheme to analyze, test and reuse microservices.
- Service dependency graphs are produced to visualize and analyze microservices.
- The test cases required to deal with microservice changes can be selected automatically.
- Existing microservices can be retrieved using word2vec-based matching algorithm.

ARTICLE INFO

Article history:
Received 30 January 2019
Received in revised form 10 May 2019
Accepted 15 May 2019
Available online 20 May 2019

Keywords:
Microservice retrieval
Microservice testing
Microservice analysis
Service dependency graph
Behavior-driven development

ABSTRACT

The microservice architecture (MSA) differs fundamentally from the monolithic, layered architecture. The use of microservices provides a high degree of autonomy, composability, scalability, and fault-tolerance. MSA is regarded by many as a promising architecture for smart-city applications; however, a number of issues remain, including (1) the management of complex call relationships among microservices; (2) ensuring the quality of the overall software system even as new microservices are added and existing ones are modified, and (3) locating existing microservices that satisfy new requirements. In this paper, we propose a novel approach to the development of microservice-based systems, referred to as GSMART (Graph-based and Scenario-driven Microservice Analysis, Retrieval and Testing). GSMART enables the automatic generation of a "Service Dependency Graph (SDG)" by which to visualize and analyze dependency relationships between microservices as well as between services and scenarios. It also enables the automatic retrieval of test cases required for system changes to reduce the time and costs associated with regression testing. A microservice retrieval method using VSM and word2vec accelerates the development of new microservices tailored specifically to the needs of users based on user-provided scenarios. Experiment results demonstrate the feasibility, effectiveness, and efficiency of all of the main features of GSMART.

1. Introduction

Software applications built on conventional monolithic architectures can be divided into a number of discrete layers, including the front-end user interface (UI), service-side logic components, and databases. Monolithic architectures struggle when the number of users exceeds the capacity of the server. These systems are also difficult to manage, reuse, and maintain due to a lack of modularization [1]. Microservice architecture (MSA) [2–4] is an emerging cloud-based software architecture [5], which differs fundamentally from monolithic, layered architecture. MSA provides intermediate building blocks, referred to as microservices to handle specific domain-driven functions. Microservices provide autonomy, composability, fault-tolerance, and scalability. Several enterprises, such as Netflix, Amazon, and Spotify, have adopted MSA to create scalable and maintainable software systems.

The concept of the "smart city" is growing in popularity in academia and industry [6]. This vision will eventually be realized through the pervasive implementation of information and communication technologies (ICT) to improve a variety of services and systems [7]. Among the various forms of ICT that have been developed, MSA is regarded by many as a promising architecture for smart-city applications [8,9].

Despite the widespread adoption of MSA, there remain a number of critical issues, which have not been adequately addressed using existing methodologies. The two essential characteristics

* Corresponding author.
E-mail addresses: albert@ntou.edu.tw (S.-P. Ma), 10557009@ntou.edu.tw (C.-Y. Fan), 10557030@ntou.edu.tw (Y. Chuang), 10657019@ntou.edu.tw (I.-H. Liu), lanciwei@tw.ibm.com (C.-W. Lan).

https://doi.org/10.1016/j.future.2019.05.048
0167-739X/

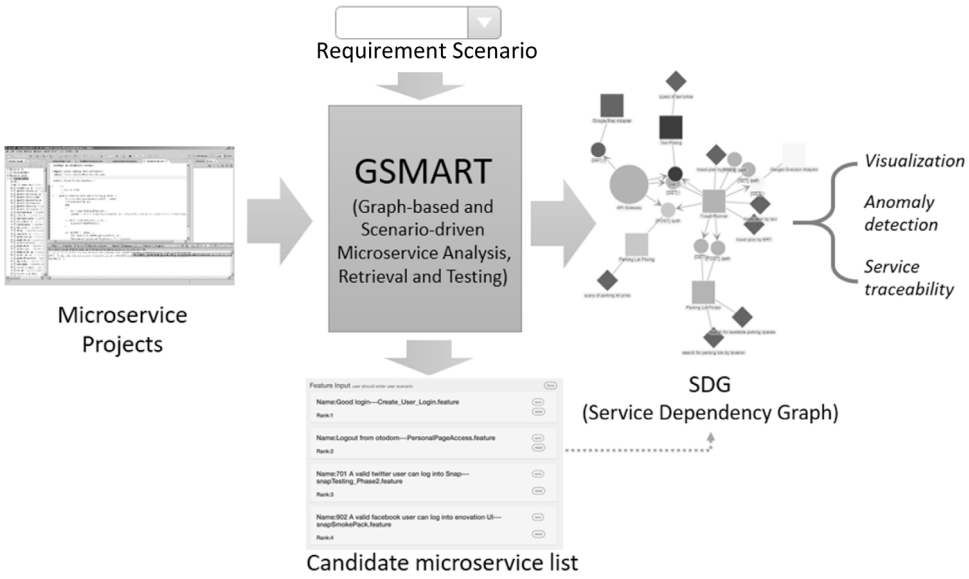


Fig. 1. Overview of proposed GSMART scheme.

of microservices are choreography [10] and evolutionary design [11]. Choreography refers to services communicating with each other without any centralized elements for service composition. Evolutionary design advocates the creation of new services whenever a new set of requirements is encountered, since service-based systems tend to change frequently due to their dynamic nature [12]. As a result, many existing services have multiple versions satisfying the various demands put to them. These characteristics impose a number of issues, including (1) the management of complex call relationships among microservices; (2) ensuring the quality of the overall software system even as new microservices are added or existing ones are modified, and (3) locating existing microservices that satisfy new requirements.

In this paper, we present a novel approach to the development and operation of microservice-based systems, referred to as GSMART (Graph-based and Scenario-driven Microservice Analysis, Reuse, and Testing). GSMART performs three primary functions: (1) facilitating the creation of service dependency graphs (SDGs) to visualize microservice-based systems; (2) providing tools to assist developers in the selection of regression test cases for microservice-based systems and in managing the results of service tests and acceptance tests; and (3) enabling microservice retrieval to facilitate the reuse of existing microservices. Fig. 1 presents an overview of the GSMART system. The proposed scheme accepts two types of input: microservice projects and requirement scenarios for the retrieval of existing microservices. GSMART enables the parsing of microservice projects to produce SDGs for a microservice-based system. The resulting SDG makes it possible to visualize the system as a whole, thereby enabling the detection of anomalies, the tracing of service dependencies, and the selection of regression test cases. GSMART also makes it possible to retrieve microservices that are relevant to the query string and the given requirement scenario in order to facilitate the reuse of microservices. A developer of a microservice-based system can use GSMART to produce SDGs of current microservices, check and trace the relationships of involved microservices in a visualized way, and identify and remove risky service calls. When a requirement is established, the developer is able to retrieve possibly-reusable microservices via GSMART, and develop a new microservice based on the source code of an existing service or build a new version for an existing service by extending its functionality. When a new service or a new version of an existing service is built, the developer can conduct regression tests based on test cases that are automatically selected by GMART. In summary, the aim of GMART is to provide supplementary functionality for the development and maintenance of microservice-based systems.

The remainder of this paper is organized as follows. In Section 2, we present an outline of background knowledge and related work. In Section 3, we provide an in-depth description of the proposed GSMART scheme. Quantitative experiments are presented in Section 4 and conclusions are drawn in Section 5.

2. Background and related work

2.1. Characteristics of microservices

An in-depth survey of the literature allowed us to identify the following essential characteristics of microservices:

(1) Autonomy: Microservices are responsible for the execution of their own underlying processes, which can be initiated, operated, and shut down independently. Each microservice can also be developed and deployed discretely within its own development lifecycle. Microservices are generally executed atop an environment-isolated container, such as Docker [13].

(2) Scalability: Microservices are small, stateless components, which makes them easy to scale up.

(3) Technology heterogeneity: Microservices use unified platform-independent communication protocols for the exchange of information.

(4) Specialization: Each service handles specific domain logic and communicates with other services using well-designed interfaces.

(5) Fault tolerance: The fact that each microservice has its own lifecycle means that it is able to tolerate failures by skipping the invocation when a service fails.

2.2. Testing mechanisms for microservices

Testing is a key step in software development. Within a microservice architecture, testing also plays an important role in the integration of services in a pipeline. The following three testing methods are usually included in the test process: unit test, service test, and user acceptance test (UAT).

```
Feature: Guess the word
    Scenario: Breaker joins a game
        Given the Maker has started a game with the word "silky"
        When the Breaker joins the Maker's game
        Then the Breaker must guess a word with 5 characters
```

Fig. 2. Example feature scenario (number guessing game).

During the initial development, microservices must pass unit tests to ensure their functionality. JUnit and Mockito are commonly used as unit testing tools. Service tests consider the contract between a service provider and the service consumer [14] to ensure successful service invocation. During consumer-driven contract testing, the unit test is implemented on the consumer side to define expected requests and responses for the API provider using a domain-specific language (DSL). Pact is an open-source contract-driven testing tool. The final step is a UAT to confirm that the system works well in all anticipated situations.

Behavior-driven development (BDD) [15,16] is gaining attention in the development and testing of microservices. Gherkin is a business-readable DSL commonly used in BDD. It can be used to describe the behavior of software using simple words such as "given", "and", "when", and "then". A simple example is presented in Fig. 1. BDD can also be used with testing tools at multiple levels of granularity. Cucumber and Serenity BDD are BDD frameworks that support Gherkin as well as a variety of programming languages (see Fig. 2).

Savchenko at al. [17] proposed a process of microservice testing, including three different levels, component testing, integration testing, and system testing. Authors also devised a microservice testing service architecture to integrate various test designs and implementations. This approach does not consider consumer-driven contract tests and regression tests. In [18], Ashikhmin et al. presented a mock service generator based on RAML (RESTful API Modeling Language) [19] to facilitate the testing of service interfaces. Generated mock services are placed in Docker containers to be directly deployed in the microservice environment. The feature of mock service generation is not directly supported by GSMART currently, but could be integrated with Pact in GSMART. Kargar and Hanifizade [20] proposed an automated method in running regression tests of microservices by using input traffic of the running version in production environment as test input and comparing service outputs of the running version to the new developed version in development environment. Rather than comparing service requests and responses of two versions directly, GSMART provides a more systematic way to conduct regression tests by reusing developed service/acceptance test cases. Besides, although all above methods are able to assist users in microservice testing, GSAMRT provides multiple unique features for microservice testing, such as displaying test results on SDG and selecting test cases based on SDG.

2.3. Microservice retrieval

Web service discovery is an important research topic in service computing [21]. Hao et al. [22] proposed a service discovery method based on information retrieval (IR), in which the relevance and importance of a service are used to enhance the precision of service discovery. Serma et al. [23] presented an automated service classification scheme in which service classifiers are trained using an artificial neural network (ANN) or support vector machine (SVM). In [24], Bukhari and Liu presented the WSSE (Web service search engine) to retrieve both SOAP and RESTful services based on discovered semantic meanings by using the probabilistic topic modeling and clustering techniques. Increasingly, researchers have been focusing on the identification and analysis of microservices. Gysel et al. [25] used a review of the literature and experience in industry to guide the development of a service decomposition scheme based on 16 coupling criteria. Their approach enables the identification of candidate microservices, while reducing coupling between services and promoting cohesion within services. Baresi et al. [26] proposed a method by which to identify potential candidate microservices by discovering fine-grained groups of cohesive operations based on the semantic similarity of functionality described using OpenAPI specifications. Obviously, none of the above methods are fully applicable to the problem of microservice retrieval and able to enhance the reusability of microservices.

3. GSMART (graph-based and scenario-driven microservice analysis, retrieval, and testing)

3.1. System requirements

In the following, we present an in-depth description of the proposed GSMART scheme. We begin by outlining the four essential functions that GSMART was developed to address.

1. *Managing and visualizing dependency relationships between microservices as well as between scenarios and services.* As mentioned in [27], dependency management is a crucial part of system and software design. The primary requirement of GSMART is the extraction of service calls (i.e., service invocations) between microservices for the construction and visualization of service dependency graphs. BDD is currently the preferred approach to the development and testing of microservice-based systems; therefore, it is essential that any model developed for visualization represents relationships among BDD scenarios and microservices.
2. *Detecting cyclic dependency references.* Cyclic dependency references between microservices are highly risky from the perspectives of development and maintenance. They commonly cause runtime failures and increase the complexity of new versions of microservices. The ability to detect cyclic dependency is the second system requirement.
3. *Selection of regression test cases.* By default, all service test cases and acceptance test cases undergo regression tests in response to any change in requirements or microservice modification. However, many of the microservices are not affected by the changes. Selecting and prioritizing test cases to reduce the time and costs associated with regression testing is the third system requirement.
4. *Retrieval of existing microservices.* In the development of a new microservice (or a new version of an existing one), it is preferable to reuse or modify an existing microservice in order to speed up development and reduce costs. Providing an effective microservice retrieval method that eliminates the need to manually check all keyword-matched microservices is the fourth system requirement.

3.2. Generation, visualization, and analysis of service dependency graphs (SDGs)

Each microservice provides APIs to other microservices and consumes (or calls) other microservices in the system. Service invocations cause dependency relationships between microservices, and when there is a large number of microservices, the complexity of dependency relationships between the microservices can be difficult to manage. In this study, we employed service dependency graphs (SDGs) to address the above issue.

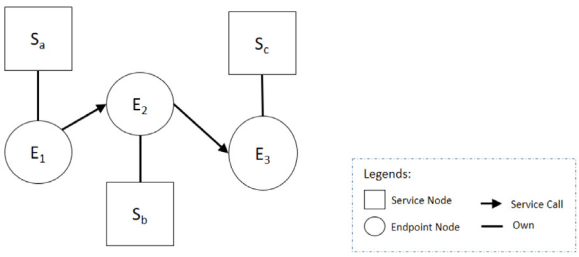


Fig. 3. Conceptual illustration of service dependency graph (SDG).

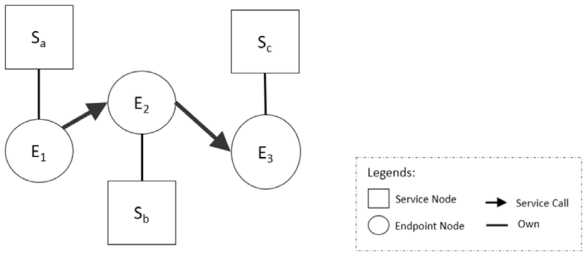


Fig. 4. Conceptual illustration of service invocation chain (SIC).

The basic idea is to collect all service invocation links, construct a service dependency graph representing all of the links, and provide a visual representation of the SDG in a manner that allows users to browse all service dependency relationships and check for anomalies or errors. The construction of an SDG for visualization involves three fundamental steps: (1) gathering all service invocation links from multiple microservice projects associated with a given microservice-based system; (2) constructing an internal representation of the SDG; and (3) displaying the SDG using visualization techniques.

As shown in Fig. 3, the concept of an SDG is simple. In this image, rectangles indicate microservices and circles represent service endpoints. Any given microservice possesses one or more endpoints; i.e., a microservice component provides one or more service operations. The arrows indicate that a microservice invokes an endpoint of another microservice.

In this research, we also employed service invocation chains (SICs) to provide supplementary information for the SDGs. This chain indicates a sequence of service invocations between multiple microservices, as indicated by the thick line in Fig. 4. GSMART is able to find all microservices involved in an SIC, and present the SIC in an SDG when the user clicks on any related service node. Furthermore, in the event of a service test case failure, the SIC associated with the failed invocation is highlighted to facilitate identification of the root error.

In summary, GSMART makes it possible for developers to manage complex service interactions using SDG, and find the root of errors using SIC. The main tasks performed by GSMART are as follows: (1) extracting information pertaining to service endpoints and service calls to construct service dependency relationships; (2) checking the status of service tests and automatically generating test code (using Pact) in situations where service test cases are not written; and (3) combining all information to produce an SDG document suitable for visual display.

3.2.1. Construction of SDG

In this study, we employed the *Reflection* mechanism for the extraction of information required to create an SDG, such as service endpoints and service calls. Many programming languages, such as Java, Python, and C#, support the *Reflection* mechanism, which allows the extraction of program information without the need for execution. In cases where the implementation language does not support reflection, service dependency information can be obtained directly using Swagger with extended properties.

According to many researchers [28,29], the Java-based Spring Boot Framework is the most mature framework for microservice development. We therefore opted to include support for Spring Boot projects in accordance with the following guidelines:

- GSMART relies on *Spring Boot Actuator* to obtain endpoint information; therefore, developers are strongly recommended to use the *Spring Boot Actuator* to monitor microservices and gather metrics.
- GSMART relies on *Spring Feign*, a declarative HTTP client developed by Netflix, to invoke other microservices, to retrieve information related to service calls. Spring Feign works smoothly with *Spring Eureka* to enhance flexibility. Developers mark *FeignClient* in the interface and specify the name of the microservice, which must match the name registered in *Spring Eureka*. *FeignClient* must also be annotated using *RequestMapping* information to specify the service URL path and service method.
- GSMART generates reports based on Swagger; therefore, developers are recommended to apply *springfox-swagger2* to automatically generate Swagger documents based on annotations.

For projects that follow the above guidelines, GSMART uses the *Java Reflection* mechanism to obtain information pertaining to microservice endpoints and service calls in order to establish relationships between microservices. This involves the use of Java interfaces and their corresponding implementation classes. GSMART inspects the links of all microservice projects associated with the target application in order to build service invocation chains and construct an overall service dependency graph. For details, please refer to [30]. In this research, we employed D3.js[1] (a widely-adopted web visualization framework) to render the SDG, an example of which is presented in Fig. 5 (four microservices with multiple service endpoints and service calls in a cinema application). Representations such as this allow developers to trace linkages between microservices.

3.2.2. Detection of cyclic dependency

GSMART is able to detect instances of cyclic dependency, which are classified into two types: weak and strong (Fig. 6). Weak cyclic dependency indicates that a cycle occurs among multiple services but not among multiple endpoints. It is a possible design error that could lead to resource dependency and competition. The left side of Fig. 6 presents a weak cyclic dependency between S_a and S_b. Strong cyclic dependency indicates that a cycle occurs among multiple endpoints, which could cause unlimited service calls leading to system to malfunction or crash. The right side of Fig. 6 presents a strong cyclic dependency among E_4, E_5, and E_6.

The process of detecting a cyclic dependency starts with the removal of all endpoint nodes from the SDG and the identification of cycles using Tarjan's Strongly Connected Component algorithm [31]. In the examples in Fig. 7, GSMART is able to detect cycles in both cases (A and B). Note that at this point, the type of cyclic dependency (strong or weak) has yet to be determined. For the SDG with cycles, GSMART removes the service nodes from the original SDG (leaving only the endpoint nodes), and again searches for cycles using the Tarjan algorithm. Any cycle that still exists is categorized as a strong cyclic dependency. Cycles that cannot be analyzed are categorized as a weak cyclic dependency. In Fig. 7, no cycle is detected for Case A in step 2 (indicating a weak cyclic dependency), whereas a cycle is detected for Case B (indicating a strong cyclic dependency).

[1] https://d3js.org/.

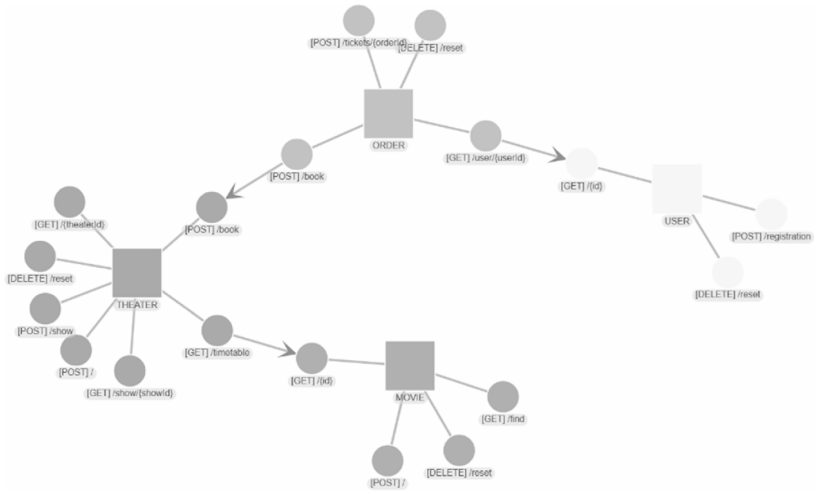


Fig. 5. SDG visualized using D3.js.

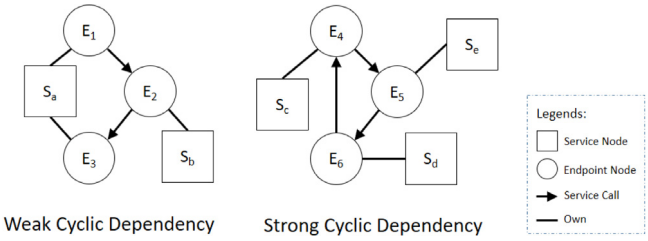


Fig. 6. Concept illustration of cyclic dependency.

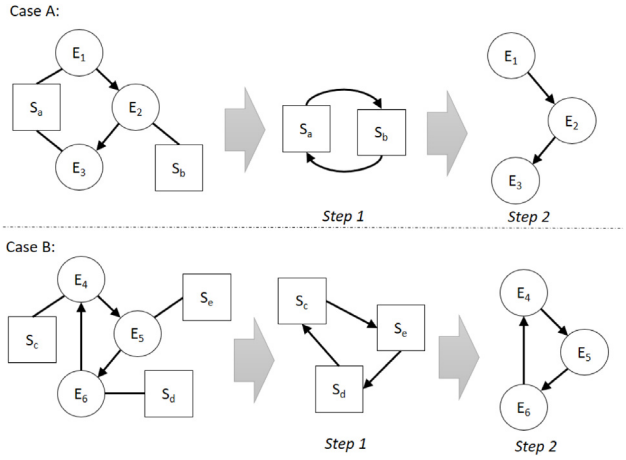


Fig. 7. Schematic illustration showing categorization of cyclic dependency.

3.2.3. Display of test results on SDG and calculation of service test coverage

When using GSMART, developers are expected to test microservices (in accordance with Pact) using the *PactProviderRule* API with the annotations of *PactVerification* and *Pact*. *PactProviderRule* is Pact's mock service, which is used to verify the service call. *PactVerification* is executed by Pact to operate in conjunction with the contract defined in Pact. The annotation is an interactive contract describing capabilities expected of the microservice. The *Reflection* mechanism allows GSMART to inspect the directory code of all microservice projects associated with a system, and determine whether all service calls have been tested using Pact in accordance with the methods listed above. Following this examination, all of the service calls that failed are highlighted in SDG.

After determining whether the microservices satisfy the specified test cases, GSMART calculates the coverage of the service tests based on all of the test results obtained for the microservice-based system under development. Service test coverage is the ratio of the number of microservice calls that have passed the test to the total number of service calls in the system. In other words, GSMART can not only show the results of service tests on the SDG, but also supply a consolidated result, the service test coverage, to help developers understand the current status for all service call relationships. Developers may browse the SDG to visually check for failed service calls. Any identified bugs can be fixed by examining the service invocation chain and identifying the root of the error. Fixed service calls can be re-tested and the GSMART will incorporate the latest test results to the SDG and show the updated service test coverage immediately.

3.3. Scenario-driven microservice testing

BDD development is usually conducted in accordance with the "Red/Green/Refactor" process, where "Red" indicates that the test has been failed, "Green" indicates that the test has been passed, and "Refactor" indicates that refactoring has been conducted. This means that service test cases and UAT cases are developed incrementally from an initial failure status until a pass is earned for the correct implementation. At the end of this process, the design and code are refactored to enhance software quality. During BDD-based development for a microservice-based system, the domain experts, project managers, and developers collaboratively write the microservice requirements revealed in the test scenarios. The developers then iteratively design and test the microservices until the tests are passed. At the end of this process, the developers refine the service design and service code.

In cases where one or more of the requirements change or a microservice is modified, the developers need to manually search for other microservices that may be affected. This type of search is generally based on information related to relationships between system requirements and microservices as well as among microservices; however, this information must be obtained and stored manually. As mentioned previously, BDD is commonly used in the development of microservice-based systems in which Gherkin-based scenarios are used as a test case and as a means to derive requirements. Within a microservice-based system, each service handles its own specific function within clear boundaries

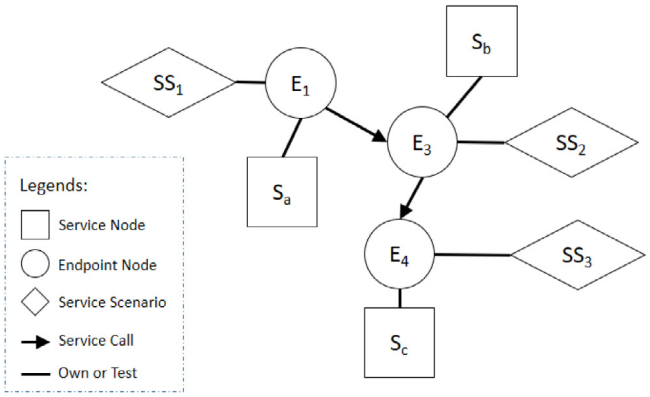


Fig. 8. Conceptual illustration of SDG with test scenarios.

to facilitate service communications. In this study, we introduce traceability among services and between scenarios and services to automate the process of evaluating the impact of microservice changes. GSMART is able to retrieve and prioritize test cases that need to be executed for a modified service.

In building relationships between services and scenarios, we can define the linkages in two ways: (1) adding service endpoints to the scenarios and (2) annotating scenarios in the OAS (OpenAPI Specification)[2] document associated with that service. Both of these methods are useful to users in the establishment of linkages between scenarios and services. Fig. 8 illustrates the concept of SDG using specific scenarios. Scenario SS_1 should be re-tested when S_a, S_b, or S_c is modified, due to the fact that S_a invokes S_b while S_b invokes S_c. In this research, service tests were conducted using the Pact tool, whereas service test cases were managed using Pact Broker. The latest Pact DSL can be obtained through the Pact Broker API. BDD test scenarios were written in the Gherkin format, executed in Cucumber, and managed in a Git project.

Fig. 9 presents an example of a visualized SDG with multiple scenarios (represented as diamonds). This example is a trip planner application within a smart city. This application suggests travel plans based on analysis of the time and cost of (1) taking the MRT and (2) driving and parking a private vehicle. This application includes five microservices (parking lot finder, parking lot pricing, travel planner, Google direction adapter, and Google map adapter) with multiple service endpoints and service calls as well as an API gateway that supports pre-filtering and post-filtering of service calls. The visualization interface of Fig. 9 allows developers to trace the linkages between microservices and scenarios. When a new microservice is developed and added to the system, the SDG presents a visual representation of the new services, new call relationships, and new service scenarios using the visualization interface. Fig. 10 presents an example involving the addition of a new microservice used to calculate the price of taxi rides in order to extend the capability of the travel planner service.

[2] http://swagger.io/.

3.3.1. Priority-based selection of regression test cases

Based on the SDG extension scheme, we developed a priority-based method for the selection of regression test cases. GSMART implements the proposed method in five stages: (1) retrieval of all SICs; (2) filtering-out of SICs that do not involve the target service to be tested (hereafter referred to as TST); (3) filtering-out of services in the SIC that do not require testing (evaluation of test requirements is discussed later); (4) priority analysis of call relationships, each of which includes an invoking service and an invoked service; and (5) selection and prioritization of two types of test case: service test cases (i.e. Pact tests) and UAT test cases (i.e. BDD tests). This process allows GSMART to select test cases that would be valuable in the detection of anomalies or errors.

In most cases, TST is in the middle of an SIC; therefore, GSMART divides service calls into two parts: (1) those that TST depends on and (2) those that depend on TST. Services that are invoked directly by TST must be tested because they are prone to failure in cases where TST is changed. Next-level services do not require retesting because TST does not affect these services. All services that depend on TST (invoked directly and indirectly) should be tested as the behavior of these services can be altered by changes in TST.

Fig. 11 presents an example in which the rectangular nodes represent microservices and the diamond nodes represent test scenarios that link to services. In this example, node C is a modified service. In dealing with services that depend on TST, two service calls are considered: from service C to service D and from service C to service E. GSMART sets the priority scores of these two service calls to −1. Note that the service call between D and D' is not given a score because it should not be affected by any change in service C. In dealing with services that depend on TST, four service calls are considered: service A invokes service C, service B invokes service C, service A' invokes service A, and service A'' invokes service A'. In differentiating the test priorities of these service calls, the direct service calls (from A to C and from B to C) are set to 0, whereas the indirect service calls (from A' to A and from A'' to A') are set to 1 and 2, respectively. The idea underlying priority scoring is as follows: test cases for service calls that are closer to TST are assigned a lower score indicating

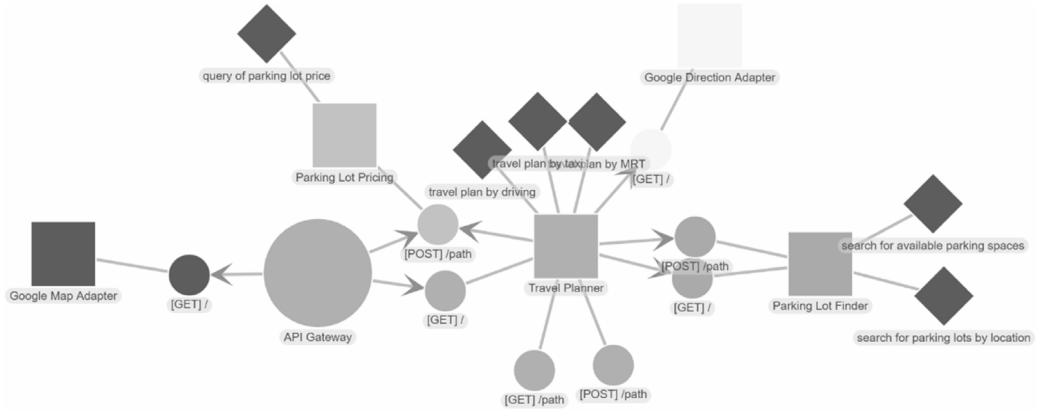


Fig. 9. Visualization of SDG with BDD scenarios.

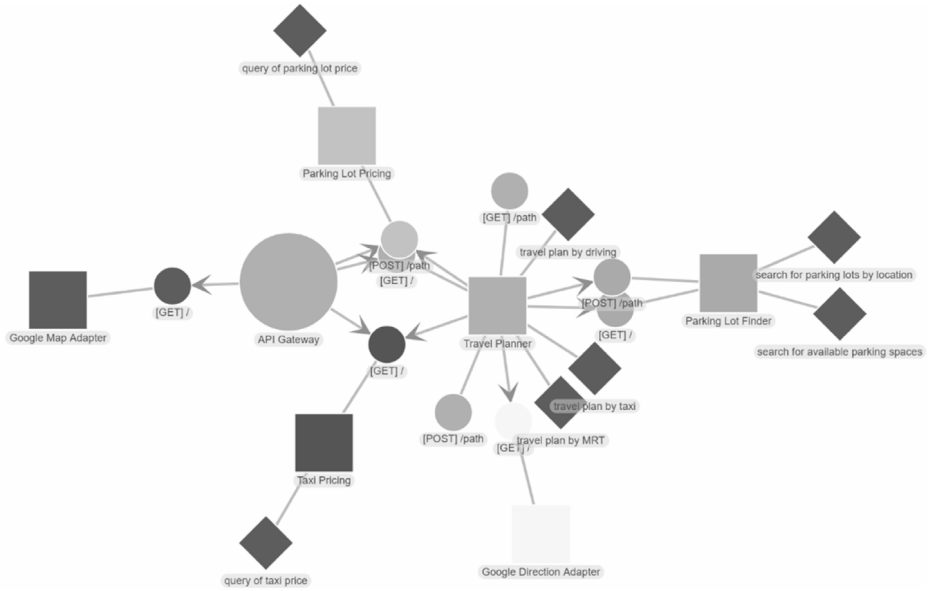


Fig. 10. Visualization of SDG with BDD scenarios involving new services.

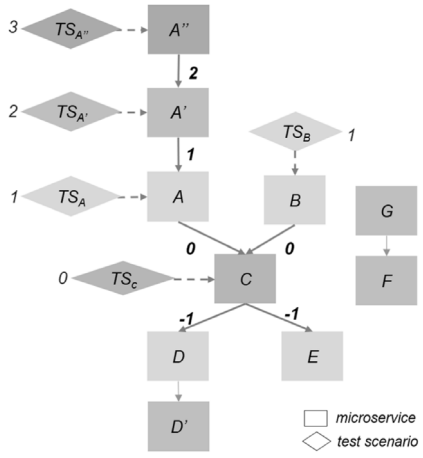


Fig. 11. Conceptual illustration of process used to assign priority scores.

higher priority. Note that a service call from service G to service F is not assigned a priority score due to the fact that this SIC does not include service C.

As mentioned previously, UAT tests are implemented using BDD and UAT test cases are developed as test scenarios in the Gherkin format. The rules determining the assignment of priority scores for test scenarios is simple: (1) the score of a test scenario that links directly to TST is set to zero; (2) the score of other test scenarios is set to one plus the score of the service call linked to that scenario depends (directly or indirectly) on TST. In the example in Fig. 11, if service C is modified or scenario TS_c is changed, then five test scenarios (TS_C, TS_B, TS_A, $TS_{A'}$, and $TS_{A''}$) are selected (the others in the figure are disregarded) and assigned scores between 0 and 3. Note that if a test scenario links to multiple services, then the test priority score is set to the lowest one.

After test priorities have been established, regression testing can be conducted. GSMART conducts the service tests and UAT tests sequentially. Obviously, a test case (e.g., T_1) that includes a failed test case (e.g., T_2) will no doubt fail; therefore, there is no need to conduct test case T_1 ahead of test case T_2. Thus, for each type of test, test cases with lower priority scores are executed before those with higher scores. In the event that a test case fails, the remaining test cases do not need to be executed until the test case in question earns a pass.

The proposed approach to regression testing provides two benefits: (1) only a partial set of test cases need to be executed, thereby lowering computational overhead; and (2) test cases are prioritized to eliminate unnecessary testing and to make it easier to find the root of errors.

3.4. Scenario-driven microservice retrieval

The widespread adoption of microservice architecture has prompted numerous microservice projects, many of which have an enormous number of microservices stored in their service repositories. When developing a new microservice or a new version of an existing one, it is preferable to reuse or modify an existing microservice in order to reduce costs and speed up development. Unfortunately, there is at present no effective tool for the discovery and/or retrieval of suitable microservices. As a result, developers must comb through projects manually or use a keyword search that returns only exact matches. Over the past two decades, researchers have proposed numerous service discovery methods based on a variety of service descriptions; however, none of these methods were developed specifically for microservices. As a result, the preparation of description documents (DDs) for microservices using OpenAPI Specification (OAS) [32] or RESTful API Modeling Language (RAML) is time-consuming. Furthermore, microservices emphasize domain-driven design, which means that a large number of words in a variety of application domains are used in DD and test scenarios, and relevance of these words is not suitably considered within the context of microservice retrieval for software development.

In this study, we equipped GSMART with a microservice retrieval mechanism based on two simple precepts: (1) we can be confident that most test scenarios for microservices (hereafter called service scenarios) are carefully prepared, due to the fact that microservices are often developed using BDD; and (2) words that are always used together in the context of software development could be used as clues to find microservices to address a new set of requirements. We leverage word2vec [33], a widely-used machine-learning method based on Natural Language Processing (NLP), to train a model that represents the

```
Scenario: Login the bookstore system
  Given a user inputs his user name "abcde" and his password "12345678"
  Then the user gets a success message
  And the user enters the member page
```

Fig. 12. Scenario example with arguments.

relevance of words based on the BDD scenarios for existing microservices. For a new requirement scenario, GSMART performs service filtering and service similarity calculation to retrieve microservices based on words expanded from a trained model. This means that users can retrieve behaviorally-similar microservices to address new requirements, rather than checking all microservices manually or settling for the narrow set of solutions with exact keyword matches.

3.4.1. Pre-processing

3.4.1.1. Argument replacement, tokenization, and stop word removal. To avoid specific arguments that could affect the result of microservice matching, GSMART conducts a preparatory task in which arguments enclosed in quotes are replaced using general words to avoid the effects of false negatives.

Fig. 12 presents an example of argument replacement in which "abcde" and "12345678" are replaced by <term>. Without this conversion, a user scenario might appear similar to a service scenario, despite the fact that they do not match in terms of arguments. Clearly, these arguments should not be considered in the matching process.

Next, GSMART performs tokenization for service scenarios based on the method proposed in [34], in which nouns and verbs are retained, and stop words are removed to reduce the negative effects of service matching. Stop words include common words, such as *say*, *go*, *have*, and *do*, and reserved keywords in Gherkin or OAS documents, such as *feature*, *scenario*, *service*, and *operation*.

3.4.1.2. Word2vec model training. The precision of service retrieval often depends on the effectiveness of detecting text similarity. Over the past two decades, there have been two types of method for the assessment of text similarity: (1) conventional IR methods that do not consider semantics, such as the TF-IDF method, and (2) word similarity methods that consider semantics, such as multiple text similarity methods based on WordNet [35]. IR methods are unable to match similar or related words, whereas word similarity methods are unable to handle many words that appear in specific domains. Thus, we developed a third method in which word relevance is determined using machine-learning methods.

As mentioned previously, we employed the word2vec method to train a two-layer neural network for the reconstruction of the linguistic context. We employed tokenized service scenarios (Gherkin documents) as training data to analyze the relevance of words within the context of software development. If only Gherkin documents were used, then it is likely that general words would not be recognized. Thus, we adopted Wikipedia as a supplementary corpus. This gave the trained word2vec model the ability to aggregate words that are not necessarily semantically similar but highly relevant in the context of software development. For example, the degree of similarity between "city" and "country" using WordNet is not high, but in the context of software development, the relevance of these two words should be high, as they are often used together. By taking into account the relevance of words, we were able to improve the precision of service matching. Note that we collected 7,935 Gherkin Features, all of which were imported into word2vec for machine learning.

3.4.1.3. Construction of BDD vectors. GSMART performs word tokenization on all existing microservices and constructs six types of word vectors (called BDD vectors) as follows: "category", "feature", "scenario", "given", "when", and "then". GSMART first uses LDA (Latent Dirichlet allocation) [36] to extract keywords related to a microservice in order to construct a "category" vector $v_c(S)$. GSMART then constructs a "feature" vector $v_f(S)$, one or more "scenario" vectors $V_s(S)$, zero or more "given" vectors $V_g(S)$, zero or more "when" vectors $V_w(S)$, and zero or more "then" vectors $V_t(S)$ from the service scenario. The text extraction/ vector construction process for each kind of vector is a trivial matter involving the retrieval of noun and verb tokens for sentences starting with a corresponding keyword (e.g., feature, scenario, given, when, and then) and the construction of a corresponding word vector. Note that the stop words have been removed; i.e., they are not included in these vectors.

3.4.1.4. Provision and initial processing for user requirements. In this research, user requirements are presented as scenarios in Gherkin format with expected service keywords and optional filtering parameters, such as microservice implementation language, framework, and data format. We expect that most user scenarios would include the structures of *feature*, *scenario*, *given*, *when*, and *then*, but it is unlikely that they would specify implementation details, such as linking services and test details. Upon submission of a user requirement, GSMART first performs the preparatory tasks (argument replacement, tokenization, and stop word removal) and then builds initial BDD vectors to address the requirement. GSMART then uses the pre-trained word2vec model to expand the words pertaining to the six vectors, find the N words of greatest similarity (currently N is set to 5), and place the expanded words back into the vectors. Note that each word has its own multi-dimensional word-embedding vector in word2vec, and the similarity between two words is equal to the cosine similarity between their corresponding word-embedding vectors. We set the value of the original word in the word vector to 1 and the expanded word to 0.5. The expanded terms were assigned lower values because they are expected only to increase the possibility of matching related words, and should not have a negative effect on exact matching.

3.4.2. VSM-based microservice matching

The Vector Space Model (VSM) is a well-known IR method [37] used to calculate the degree of cosine similarity between the vectors of two documents. In this research, we use VSM to calculate the degree of similarity between a stipulated requirement and a microservice.

3.4.2.1. Service filtering based on word2vec. From the perspective of system operation, GSMART first conducts preparatory tasks and preprocessing for all microservices, while allowing the continuous submission of user requirements. GSMART performs the initial processing on the submitted requirements and then begins the microservice matching process by performing service filtering, which proceeds through two steps: (1) retrieval of microservices containing any word in the six vectors of the user requirement; (2) selection of microservices (i.e., candidate services) from those that were previously retrieved based on a number of parameters provided by the user (e.g., language, framework, and data-format). Only microservices with parameter values that are consistent with the user-given constraints are retained.

3.4.2.2. Microservice similarity score: Calculation and ranking. GSMART uses VSM to calculate the degree of similarity between the user requirement and all of the candidate microservices retrieved in the previous task [38]. Similarity is calculated by considering all six vectors (category, feature, scenario, given, when, and then),

as outlined in Eq. (1). If the calculated Sim(R,S) score is lower than a given threshold, then candidate service S is discarded. Microservices with similarity scores exceeding the threshold (i.e., qualified services) are ranked according to score and returned to the user. Note that GSMART permits the inclusion of multiple scenarios featuring *given*, *when*, and *then* sentences, and a feature description in Gherkin format.

$$Sim\,(R,S) = w_1 \times Sim_c\,(R,S) + (1-w_1) \times Sim_{BDD}\,(R,S) \quad (1)$$

$$Sim_{BDD}\,(R,S) = w_2 \times Sim_f\,(R,S) + (1-w_2) \times Sim_{SC}\,(R,S) \quad (2)$$

$$Sim_{SC}\,(R,S) = max\{sim\,(i,j)\,|i = 1\ldots n, j = 1\ldots m\} \quad (3)$$

$$sim\,(i,j) = w_3 \times Sim_s\left(RS_i, SS_j\right) + (1-w_3) \times \frac{Sim_g\left(RS_i, SS_j\right) + Sim_w\left(RS_i, SS_j\right) + Sim_t\left(RS_i, SS_j\right)}{3} \quad (4)$$

where $Sim\,(R,S)$ is the final similarity between requirement R and microservice S; $Sim_c\,(R,S)$ is the cosine similarity between category vectors of requirement R and microservice S; $Sim_f\,(R,S)$ is the cosine similarity between feature vectors of requirement R and microservice S; $Sim_s\left(RS_i, RS_j\right)$ is the cosine similarity between a requirement scenario RS_i and a service scenario SS_j; $Sim_g\left(RS_i, SS_j\right)$, $Sim_w\left(RS_i, SS_j\right)$, and $Sim_t\left(RS_i, SS_j\right)$ are the cosine similarities of *given* vectors, *when* vectors, and *then* vectors between a requirement scenario RS_i and a service scenario SS_j; and parameters w_1, w_2, and w_3 are less than 1, respectively indicating the importance of service keywords, feature description, and scenario description. In this case, w_1, w_2, and w_3 were set to 0.1, 0.2, and 0.5.

For all requirements issued by the user, GSMART calculates similarity scores for all qualified microservices and sorts them accordingly. Finally, GSMART returns the sorted list of qualified microservices to the user for selection. The user is free to select their microservice(s) of choice using the UI of the GSMART system to obtain a service plan showing the issued requirements (scenarios, keywords, and parameters) and the selected microservices. The plan provides a helpful reference for the development of the target microservice-based system.

4. Experimental evaluations

To assess the feasibility of the proposed GSMART system, we devised three quantitative experiments to test the efficiency of SDG generation, the efficacy of regression test case selection, and the precision of the microservice retrieval. The hardware and software used in the experiments were as follows:

- OS: Arch Linux
- CPU: Intel Core i7-4790
- RAM: 16 GB
- Software Version: Java 1.8, Spring Boot 1.5.2, and Neo4j 3.1.1
- Neo4j Driver: Bolt
- Neo4j Memory Pagecache Size: 4G
- Neo4j Memory Heap Max Size: 8G

4.1. Efficiency testing of SDG generation

In this experiment, the scale of an MSA system can range from less than ten microservices to hundreds of microservices (e.g., Netflix includes more than five hundred microservices [4, 39]). Thus, we evaluated the performance of the GSMART system in microservice projects of various size. We were unable to obtain the source code and configurations of real-world microservice-based systems; therefore, we generated a set of representative test data for the simulations. The setup of the experiment was as follows:

1. Simulate 15 microservice-based applications with various numbers of microservices: 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000, 2000, 3000, 4000, and 5000.
2. Randomly generate service endpoints according to the distribution of service endpoints for services published in APIs.guru,[3] the most popular public service repository.
3. Randomly allocate service calls between microservices according to the distribution of service calls in the Netflix system.
4. Create a visualization of the SDG for each project and record the computation times.

Each experiment was conducted five times, the results of which were averaged. Fig. 13 presents data related to the time required to create SDGs for microservice-based applications of various scope. When using fewer than 200 microservices, SDG creation time was less than seven seconds, and even when this was increased to 1,000, the creation time was less than 1.5 min. Clearly, the GSMART system yields acceptable performance in the creation of SDGs, regardless of the scale of the application with one to hundreds of services. Only when the number of services is extremely large (larger than 1000), the creation time becomes more and more unacceptable.

Table 1
Experimental setup for regression test case selection.

Number of microservices	Number of service calls	Number of pact DSLs	Number of BDD scenarios
20	36	34	10

4.2. Efficacy testing of regression test case selection

To evaluate the efficacy of the proposed GSMART system in terms of test case selection, we developed 20 microservices in the domain of travel, established call relationships between microservices, and constructed the service test cases (using Pact) and acceptance test cases (using Cucumber). The setup of this experiment is listed in Table 1. We developed 20 real microservices, 36 invocations between microservices with 34 Pact DSLs that define the expected service requests and responses, and 10 BDD scenarios that describe the steps of acceptance testing.

Priority-based selection was applied to each microservice (under the assumption that the service had been modified), and the number of service test cases and user acceptance test cases (UAT) to be selected and conducted was recorded. These results are listed in Fig. 14. It can be observed that 10%–70% of the UAT cases were required, whereas only 2.78%–41.67% of the service test cases were required. This illustrates how GSMART can reduce UAT test costs by roughly 50% and service test costs by roughly 80%.

4.3. Precision testing of microservice retrieval

The aim of this experiment is evaluating the precision of the proposed microservice retrieval scheme. The experiment setup was as follows:

1. Preparation of comparison targets: we employed two alternative approaches, IR-based approach and WordNet-ext approach.

[3] https://APIs.guru/openapi-directory/.

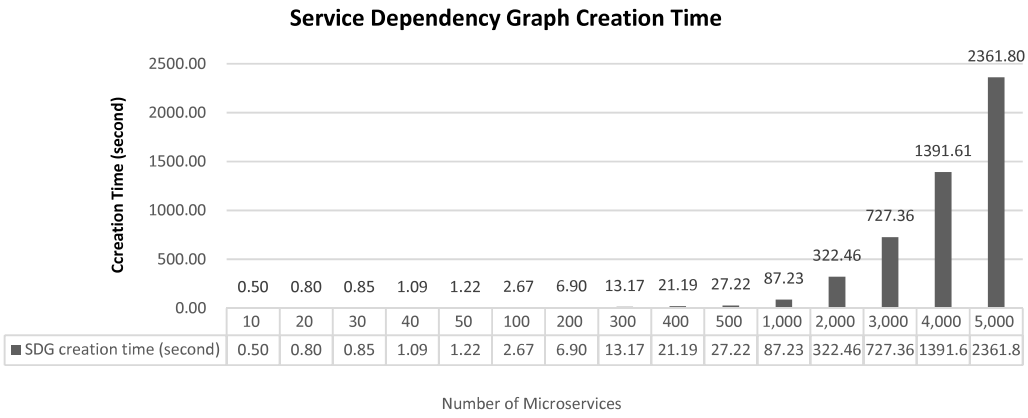


Fig. 13. SDG creation time.

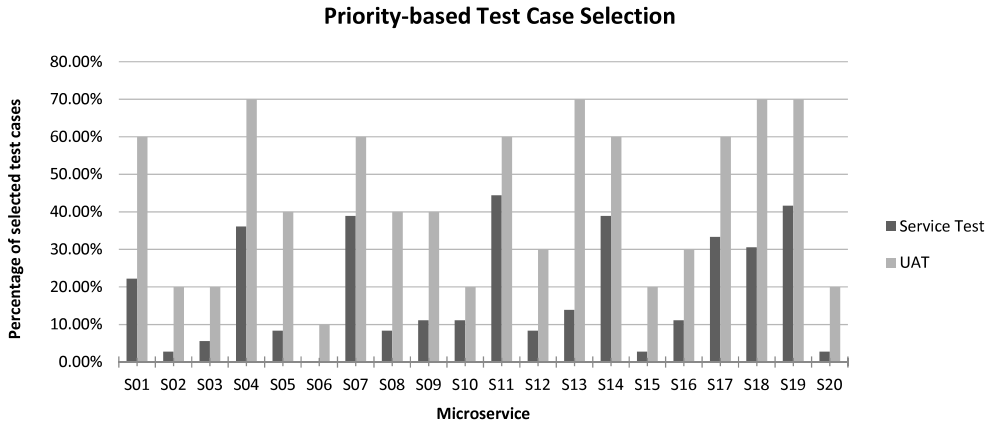


Fig. 14. Assessment of priority-based test case selection.

(1) IR-based approach: We developed a microservice search method based on conventional IR techniques as the first comparison target. This approach does not classify word vectors according to the characteristics of BDD scenarios, whereas GSMART divides word vectors into several types. This approach also lacks word expansion. The IR-based approach includes the following steps

- Performance of word tokenization and stemming,
- Use of VSM to calculate similarities between the requirement and all microservice test scripts,
- Ranking of all microservices according to the degree of similarity.

(2) WordNet-ext approach: The second method employed the IR scheme as the baseline, and then expanded the word vectors using WordNet, which is a large English dictionary with an emphasis of semantic relationships between words. In the field of service-oriented computing, WordNet is commonly used to expand semantically similar words to increase the likelihood of finding service matches. We adopted the word expansion method proposed in [34] to include words whose similarities (calculated by Li's edge counting method [40]) to the original word are larger than a given threshold (0.9).

2. Collection of test data: by crawling GitHub,[4] we were able to collect 7935 service scenarios (in Gherkin format), and then prepared 25 requirement documents selected at random from the service scenarios. We then applied the average Top-K precision indicator to verify the performance of microservice retrieval using the IR-based approach, the WordNet-ext approach, and the proposed GSMART system. Top-K precision is defined in Eq. (5). In this research, we set the value of K from 1 to 10.

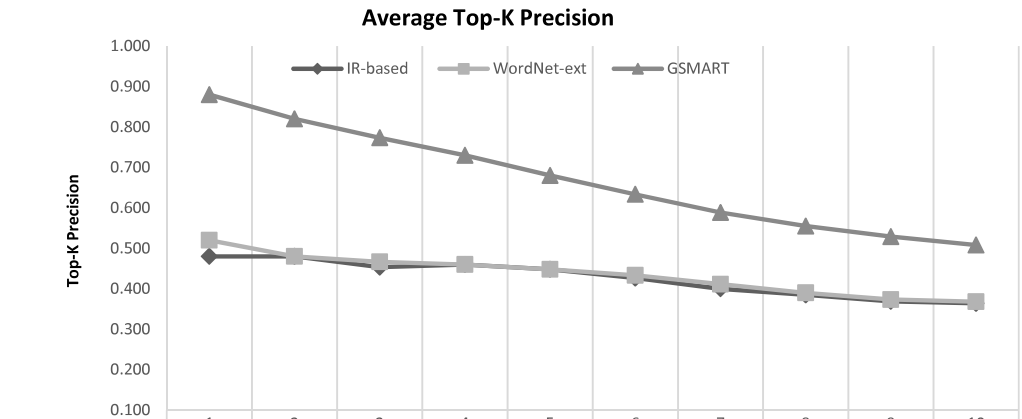
$$Precision^{K}(R_i) = \frac{\left|Rel(R_i) \cap Rank^{K}(R_i)\right|}{|Rank^{K}(R_i)|} \quad (5)$$

where R_i is the ith user requirement, $Rank^K(R_i)$ is the Top-K microservices returned (i.e., the first K microservices returned), and $Rel(R_i)$ is a set of relevant services with R_i (i.e., answers to the ith user requirement).

3. Relevance determination: in this experiment, the rules used to determine the degree of relevance between a microservice and a user requirement were as follows: (1) the domain of the required service is similar to the retrieved service, and the stipulated functionality is similar to the service operation; or (2) from the perspective of system development, making small changes to the retrieved microservice would be sufficient to enable a match with the user requirement.

Using the aforementioned experiment setup, we performed microservice retrieval for the three alternatives and recorded all

4 https://github.com/.



	1	2	3	4	5	6	7	8	9	10
IR-based	0.480	0.480	0.453	0.460	0.448	0.427	0.400	0.385	0.369	0.364
WordNet-ext	0.520	0.480	0.467	0.460	0.448	0.433	0.411	0.390	0.373	0.368
GSMART	0.880	0.820	0.773	0.730	0.680	0.633	0.589	0.555	0.529	0.508

Fig. 15. Average Top-K precision.

retrieved services. We then invited three well-trained software engineers to determine whether the retrieved services met the needs of the user based on a vote. Finally, we calculated the Top-K precision. Note that at least two of the experts had to agree that a given service matched the requirement in question.

Fig. 15 presents experiment results showing Top-K precision. These results show that the Top-1∼Top10 precisions of GSMART are clearly superior to those obtained using the other two methods. This also demonstrates that using word2vec to discover words of relevance within the context of software development is beneficial to microservice retrieval.

5. Conclusions

In this paper, we introduce a novel scheme by which to facilitate the development of microservice-based systems, referred to as GSMART (Graph-based and Scenario-driven Microservice Analysis, Retrieval and Testing). The main features of GSMART are as follows:

- "Service Dependency Graphs (SDGs)" are generated for the analysis and the visualization of dependency relationships among microservices as well as between services and scenarios. This makes it far easier to manage complex service interactions and detect faults, particularly those associated with cyclic dependency.
- The test cases required to deal with system changes can be retrieved quickly and easily using the proposed test case selection scheme to reduce the time and effort typically associated with regression testing.
- Users are provided with a list of microservices that meet their needs, based on user-provided scenarios. The use of VSM and word2vec enhance the reusability of existing microservices and accelerate the development of new systems.

Experiments demonstrate the feasibility, effectiveness, and efficiency of all three features when applied to small-scale as well as large microservice-based systems. In summary, the current version of the GSMART system enables the analysis, testing, and retrieval of microservices during the development stage. Note that the limitations of applying GSMART include: (1) GSMART mainly supports Java-based Spring Boot Framework. Users are expected to follow the guidelines described in Section 3.2.1 to allow GSMART to retrieve required information for producing and analyzing SDGs; and (2) GSMART uses Gherkin documents to specify test scenarios and service requirements. Users are also expected to adopt Gherkin to conduct behavior-driven development when using the functionality of regression testing and microservice retrieval in GSMART.

The future plans of GSMART include (1) devising a microservice monitoring mechanism to manage multiple versions of microservices based on SDGs during runtime; and (2) establishing a maintenance mechanism for the trained word2vec model to keep its accuracy and precision.

Acknowledgments

This research was sponsored by the IBM Faculty Award and by the Ministry of Science and Technology in Taiwan under grant MOST 105-2221-E-019-054-MY3.

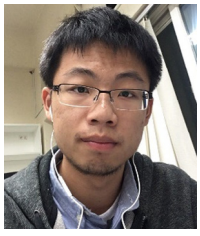
References

[1] C.Y. Fan, S.P. Ma, Migrating monolithic mobile application to microservice architecture: An experiment report, in: 2017 IEEE International Conference on AI & Mobile Services (AIMS), 2017, pp. 109–112.
[2] J. Lewis, M. Fowler, Microservices: a definition of this new architectural term, 2014, Available: http://martinfowler.com/articles/microservices.html.
[3] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc., 2015.
[4] T. Mauro, Adopting microservices at Netflix: Lessons for architectural design, 2015, Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
[5] L.M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S.N. Srirama, M.F. Zhani, Research challenges in nextgen service orchestration, Future Gener. Comput. Syst. 90 (2019) 20–38.
[6] V. Albino, U. Berardi, R.M. Dangelico, Smart cities: Definitions, dimensions, performance, and initiatives, J. Urban Technol. 22 (2015) 3–21.
[7] M. Vögler, J.M. Schleicher, C. Inzinger, S. Dustdar, R. Ranjan, Migrating smart city applications to the cloud, IEEE Cloud Comput. 3 (2016) 72–79.
[8] A. de M. Del Esposte, E.F.Z. Santana, L. Kanashiro, F.M. Costa, K.R. Braghetto, N. Lago, et al., Design and evaluation of a scalable smart city software platform with large-scale simulations, Future Gener. Comput. Syst. 93 (2019) 427–441.
[9] A. Krylovskiy, M. Jahn, E. Patti, Designing a smart city internet of things platform with microservice architecture, in: 2015 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 25–30.
[10] T. Cerny, M.J. Donahoo, M. Trnka, Contextual understanding of microservice architecture: current and future directions, SIGAPP Appl. Comput. Rev. 17 (2018) 29–45.
[11] C. Pahl, P. Jamshidi, Microservices: A systematic mapping study, presented at the Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2, Rome, Italy, 2016.
[12] P. Xiu, J. Yang, W. Zhao, Change management of service-based business processes, Service Oriented Computing and Applications, 13, March 01 2019, pp. 51–66.

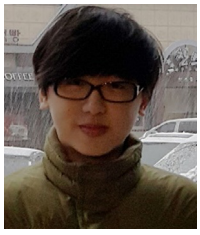
[13] What is Docker, Available: https://www.docker.com/what-docker-old.
[14] I. Robinson, Consumer-driven contracts: A service evolution pattern. Available: https://martinfowler.com/articles/consumerDrivenContracts.html.
[15] C. Solis, X. Wang, A study of the characteristics of behaviour driven development, in: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2011, pp. 383–387.
[16] M. Soeken, R. Wille, R. Drechsler, Assisted Behavior Driven Development Using Natural Language Processing, Berlin, Heidelberg, 2012, pp. 269–287.
[17] D. Savchenko, G. Radchenko, T. Hynninen, O. Taipale, Microservice test process: Design and implementation, Int. J. Inf. Technol. Secur. 10 (2018) 13–24.
[18] N. Ashikhmin, G. Radchenko, A. Tchernykh, RAML-Based Mock Service Generator for Microservice Applications Testing, Cham, 2017, pp. 456-467.
[19] RAML (RESTful API Modeling Language), Available: https://raml.org/.
[20] M.J. Kargar, A. Hanifizade, Automation of regression test in microservice architecture, in: 2018 4th International Conference on Web Research (ICWR), 2018, pp. 133–137.
[21] S.-P. Ma, C.-H. Li, Y.-Y. Tsai, C.-W. Lan, Web Service Discovery Using Lexical and Semantic Query Expansion, in: 2013 IEEE 10th International Conference on e-Business Engineering (ICEBE), 2013, pp. 423–428.
[22] Y. Hao, Y. Zhang, J. Cao, Web services discovery and rank: An information retrieval approach, Future Gener. Comput. Syst. 26 (2010) 1053–1062.
[23] S. Sharma, J.S. Lather, M. Dave, Semantic approach for web service classification using machine learning and measures of semantic relatedness, Serv. Oriented Comput. Appl. 10 (2016) 221–231.
[24] A. Bukhari, X. Liu, A web service search engine for large-scale web service discovery based on the probabilistic topic modeling and clustering, Serv. Oriented Comput. Appl. 12 (2018) 169–182.
[25] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: A systematic approach to service decomposition, in: European Conference on Service-Oriented and Cloud Computing, 2016, pp. 185–200.
[26] L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in: European Conference on Service-Oriented and Cloud Computing, 2017, pp. 19–33.
[27] S.E. Ghirotti, T. Reilly, A. Rentz, Tracking and controlling microservice dependencies, Commun. ACM 61 (2018) 98–104.
[28] O. Shelajev, Why Spring is Winning the Microservices Game, 2016, Available: https://zeroturnaround.com/rebellabs/why-spring-is-winning-the-microservices-game/.
[29] H. Schlosser, Microservices trends 2017: Strategies, tools and frameworks, 2017, Available: https://jaxenter.com/microservices-trends-2017-survey-133265.html.
[30] S.P. Ma, C.Y. Fan, Y. Chuang, W.T. Lee, S.J. Lee, N.L. Hsueh, Using service dependency graph to analyze and test microservices, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, pp. 81–86.
[31] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.
[32] OpenAPI Specification (OAS), Available: https://swagger.io/docs/specification/.
[33] T. Mikolov, K. Chen, G. Corrado, J. Dean, L. Sutskever, G. Zweig, word2vec, 2013, Available: https://code.google.com/archive/p/word2vec/.
[34] S.-P. Ma, C.-W. Lan, C.-H. Li, Contextual service discovery using term expansion and binding coverage analysis, Future Gener. Comput. Syst. 48 (2015) 73–81.
[35] G.A. Miller, WordNet: a lexical database for English, Commun. ACM 38 (1995) 39–41.
[36] S. Ma, H. Lin, C. Lan, W. Lee, M. Hsu, Real-World RESTful service composition: a transformation-annotation-discovery approach, in: 2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA), 2017, pp. 1–8.
[37] A. Bagga, B. Baldwin, Entity-based cross-document coreferencing using the vector space model, in: Proceedings of the 17th International Conference on Computational Linguistics-Volume 1, 1998, pp. 79–85.
[38] G. Salton, A. Wong, C.-S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (1975) 613–620.
[39] B. Wong, The Case for Chaos, 2014, Available: https://www.slideshare.net/BruceWong3/the-case-for-chaos.
[40] Y. Li, Z.A. Bandar, D. McLean, An approach for measuring semantic similarity between words using multiple information sources, IEEE Trans. Knowl. Data Eng. 15 (2003) 871–882.

Shang-Pin Ma received his MS degree and Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 2007. Dr. Ma is currently an associate professor in the Department of Computer Science and Engineering at National Taiwan Ocean University. His research interests include software engineering, service computing, mobile cloud application, semantic web and open data. He is an IEEE member.

Chen-Yuan Fan received his master's degree in Computer Science and Engineering from National Taiwan Ocean University in 2017. His research interests include software engineering, service computing, and mobile cloud application.



Yen Chuang received his master's degree in Computer Science and Engineering from National Taiwan Ocean University in 2018. His research interests include software engineering, service computing, and machine learning.



I-Hsiu Liu is a gradu ate student in the Department of Computer Science and Engineering at National Taiwan Ocean University. His research interests include service computing and information visualization.

Dr. Ci-Wei Lan is an R&D manager at IBM Taiwan. He received Ph.D. degree from National Central University, Taiwan. His research interests are service-oriented computing, big data management and analytics. He is the core organizing member of IEEE ICEBE conference for advocating service-oriented technologies and e-business applications.