Service Candidate Identification from Monolithic Systems based on Execution Traces

Wuxia Jin, Ting Liu, *Member, IEEE,* Yuanfang Cai, *Member, IEEE* Rick Kazman, *Senior Member, IEEE,* Ran Mo, Qinghua Zheng, *Member, IEEE,*

Abstract—Monolithic systems increasingly suffer from maintainability and scalability issues as they grow in functionality, size, and complexity. It is widely believed that (micro)service-based architectures can alleviate these problems as each service is supposed to have the following characteristics: clearly defined *functionality*, sufficient *modularity*, and the ability to *evolve independently*. Industrial practices show that service extraction from a legacy monolithic system is labor-intensive and complex. Existing work on *service candidate identification* aims to group entities of a monolithic system into potential service candidates, but this process has two major challenges: first, it is difficult to extract service candidates with consistent quality; second, it is hard to evaluate the identified service candidates regarding the above three characteristics. To address these challenges, this paper proposes the *Functionality-oriented Service Candidate Identification* (*FoSCI*) framework to identify service candidates from a monolithic system. Our approach is to record the monolith's execution traces, and extract services candidates using a search-based functional atom grouping algorithm. We also contribute a comprehensive service candidate evaluation suite that uses interface information, structural/conceptual dependency, and commit history. This evaluation system consists of 8 metrics, measuring functionality, modularity, and evolvability respectively of identified service candidates. We compare *FoSCI* with three existing methods, using 6 widely-used open-source projects as our evaluation subjects. Our results show that *FoSCI* outperforms existing methods in most measures.

Index Terms—Microservice, monolith decomposition, service candidate, execution trace, functionality, modularity, evolvability.

1 Introduction

Monolithic software systems increasingly suffer from maintainability and scalability issues as they grow in functionality, size, and complexity. It is widely believed that *service-based* (including *microservice-based*) architectures can help alleviate these problems as each service is supposed to exhibit well-defined *functionality* and high *modularity*. Most importantly, each service should be able to *evolve independently*[1] [2][3]. To migrate a monolithic system to a services-based system, there are two major options: rewriting from scratch, or extracting services from the legacy source code. If the legacy code has high value, the latter is recommended[2] [3]. Large enterprises, such as Netflix and Amazon, have adopted the second option, which includes two phases, *splitting design* and *code implementation*. *Splitting design* is used to identify the boundaries to split a monolith into multiple functional units, where each separable function becomes a service candidate [2]. *Code implementation* then realizes *service candidates* as physical *services*. Our work focuses on *splitting design*, which is complex and laboring-intensive [4][5].

To alleviate the burden of *splitting design*, researchers have proposed various methods to automate *Service Candidate Identification*, that is splitting entities (e.g., methods, classes) of a monolithic system into multiple function groups, each of which corresponds to a potential service candidate [5][6]. *Service Candidate Identification* is similar to traditional software decomposition or software clustering techniques, such as Bunch [7] and ACDC [8], with the assumption that the clustered entity groups should follow the principles of high cohesion and low coupling [9][10][11]. Existing work for service candidate identification faces two major challenges: (1) **It is difficult to produce high quality service candidates in terms of well-defined functionality, sufficient modularity, and independent evolvability;** (2) **There is no systematic way to measure these qualities quantitatively and consistently.**

This paper introduces our approaches to address the above challenges. We first propose a *Functionality-oriented Service Candidate Identification* (*FoSCI*) framework to identify service candidates, through extracting and processing execution traces. This framework consists of three major steps: extracting representative execution traces, identifying entities using a search-based functional atom grouping algorithm, and identifying interfaces for service candidates. We also present a comprehensive measurement system to

- *W. Jin and Q. Zheng are with Ministry of Education Key Laboratory of Intelligent Networks and Network Security (MOEKLINNS), Xi'an Jiaotong University, China. Email: wx_jin@stu.xjtu.edu.cn, qhzheng@mail.xjtu.edu.cn.*
- *T. Liu is corresponding author. He is with the MOEKLINNS and School of Cyber Security, Xi'an Jiaotong University, China. Email: tingliu@mail.xjtu.edu.cn*
- *Y. Cai and R. Mo are with the Department of Computer Science, Drexel University, USA. Email: yfcai@cs.drexel.edu, rm859@drexel.edu*
- *R. Kazman is with the Department of Information Technology Management, University of Hawaii, USA. Email: kazman@hawaii.edu*
- *This paper is an extended version of our conference paper on IEEE ICWS 2018[1]. We redefine the problem as service candidate identification, design a new framework and enhance the method by introducing search-based functional atom grouping, and improve the measurement system by adding Modularity and designing Independence of Evolvability. Moreover, the experiments have be re-designed, by adding three new large projects and extra discussion on test case coverage.*

1. https://martinfowler.com/articles/microservices.html
2. https://martinfowler.com/articles/break-monolith-into-microservices.html

quantitatively evaluate service candidate quality in terms of functionality, modularity, and evolvability.

FoSCI framework. There is prior research ([12][13][14][15][16]) that aims to split a monolithic system based on source code structure. These methods usually model the structure of the monolith as a graph, in which a node represents a class, and an edge represents either a class-to-class structural relation [12][13][14], or semantic similarity [15][16]. To achieve high cohesion and low coupling [9][10][11], edges with weaker relations were removed to split into subgraphs. These methods only consider graph structure, without considering functionality. Consequently, as we will show, the resulting service candidates may not have well-defined functionality. In addition, these methods cannot be applied when the source code is not available.

FoSCI addresses this challenge by utilizing execution traces collected from logs to guide service candidate identification. The rationale is that execution traces can precisely reveal program functionality [17][18][19]. To evaluate FoSCI, we first collected 3,349,253,852 method call records from execution logs of widely-used projects. After that, we reconstructed, reduced, and extracted representative execution traces. Using these execution traces, FoSCI first generates *Functional Atom*s, each being a coherent and minimal functional unit. The assumption is that a set of classes that frequently appear in the same set of execution traces are dedicated to related functionality. Next, FoSCI assigns functional atoms to service candidates by optimizing four objectives from execution traces. Finally, FoSCI identifies the interface classes with operations for each candidate. Using execution traces, FoSCI can thus identify service candidates even when the source code of the monolithic application is not available.

Evaluation Framework. There are multiple ways to extract service candidates, but there currently exists no systematic evaluation framework to assess the functionality, modularity and evolvability of the resulting services candidates [2][3]. *Functionality* describes visible functions provided by a service, which should be a business capability accessible by external clients. *Modularity* [3] measures if internal entities within a service behave coherently, while entities across services are loosely coupled. *Evolvability* [2] measures a service's ability to evolve independently: a system may have tens or even hundreds of services [20]; if changes to one service frequently affect other services, it would be challenging for services to evolve exclusively. There is prior research on quantifying some of these qualities. For example, Bogner et al. [21] proposes cohesion and coupling measures for services. But measuring the evolvability of service candidates has not been explored.

We have constructed a comprehensive evaluation suite to systematically measure service candidates from the three aspects, using 8 measures: 1) *Independence of Functionality* is measured by integrating *Interface Number* (IFN), *Cohesion at Message Level* (CHM), and *Cohesion at Domain Level* (CHD). 2) *Modularity* is measured by extending the Modularity Quality measure proposed by Mancoridis [22]. These metrics measure both structural and conceptual modularity of service candidates. 3) *Independence of Evolvability* is quantified using 3 measures we designed: *Internal Co-change Frequency* (ICF), *External Co-change Frequency* (ECF), and *Ration of ECF to ICF* (REI). These measures can be derived from the revision history of the original system. A project's revision history, stored in its version-control system, provides a unique view of the actual evolution path of a software system [23][24]. The revision history holds a wealth of software evolution information including the changes and metadata about changes, such as who made each change, what is the purpose of the change, and when the change was made. As we will show, these 8 metrics comprehensively quantify the three critical quality aspects of service candidates.

In summary, our contributions are as follows:

- The FoSCI framework to identify service candidates, including entity and interface identification. FoSCI employs execution traces because they accurately reveal functional groupings in software systems.
- An evaluation suite for service candidates, which consists of 8 metrics to quantify the three quality criteria of service candidates: *Independence of Functionality*, *Modularity* and *Independence of Evolvability*.

The rest of this paper is as follows. Section 2 describes our proposed functionality-oriented service candidate identification framework. Section 3 illustrates the measures for quantifying *Independence of Functionality*, *Modularity* and *Independence of Evolvability*. Section 4 and Section 5 present experiment setup and evaluation results. Section 6 discusses limitations and threats to validity. Section 7 surveys the related works. Section 8 provides conclusions and discusses future work.

2 METHODOLOGY

In this section, we first introduce basic definitions related to service candidates. After that, we illustrate the FoSCI framework using *JPetstore*[3] (Monolithic version 6.0.2) as a running example. The intermediate results and tables of this example can be found in our repository[4]. The symbol notations we use are listed in Table 1.

2.1 Definitions

A *Service Candidate* is an intermediate product in the *splitting design* phase during a migration from a monolithic system into a (micro)service-based architecture. It has the potential and is subject to further refinement to be actually implemented as a physical *Service*. We formally define *Service Candidate* as follows:

$$SC = (E_{ser}, I, O) \tag{1}$$

where, E_{ser} denotes a set of entities which compose *SC*. I is a set of *Interface Classes* of *SC*. O represents a set of fine-grained *Operations* of I. In this paper, an entity is a class of the original monolith. An interface class is a class entity that has the potential to be publicly published. Through the published interface class, a service candidate provides functionality visible to external clients. An operation is a method publicly provided by an interface class.

3. https://github.com/mybatis/jpetstore-6
4. https://github.com/wj86/FoSCI

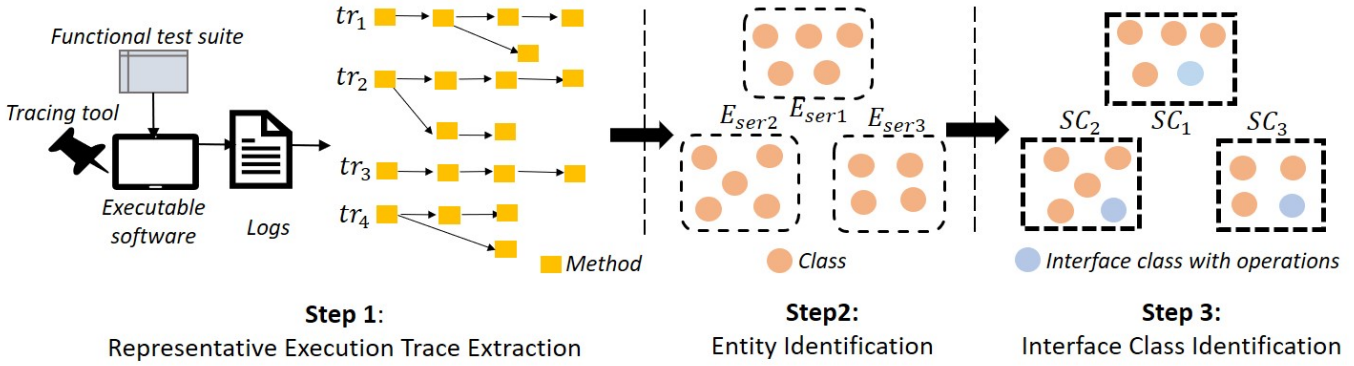


Fig. 1: The FoSCI Method

TABLE 1: The basic symbols globally used

Symbol	Description
m	A *Method*.
c	A *Class*.
$e = (m_i, m_j)$	A *Method Call* in which m_i invokes m_j.
$tr : \langle e_1, e_2, ..., e_n \rangle$	**Execution Trace**: a sequence of *Method Calls*.
$ctr = \{c_1, c_2, ..., c_l\}$	A set of *Classes* involved in tr.
$O_{tr} = \{tr\}$	**Execution Trace Set**: a set of *Execution Traces* collected originally.
R_{tr}	**Representative Execution Trace Set**, reduced from O_{tr}. $R_{tr} \subseteq O_{tr}$.
fa	A *Functional Atom*.
SC	A *Service Candidate*.
E_{ser}	A set of entities that compose SC.
I	A set of **Interface Classes** of SC.
O	A set of **Operations** of I.

2.2 FoSCI Framework Overview

Figure 1 depicts our **F**unctionality-**o**riented **S**ervice **C**andidate **I**dentification (FoSCI) framework with three steps:

Step 1: Representative Execution Trace Extraction. We apply a *tracing tool*, Kieker 1.13 [25], to collect execution traces of a monolithic *executable software*, using a set of pre-determined *Functional test suite*. These executions are recorded in *Log* files, from which we extract a *Representative Execution Trace* Set, R_{tr}.

Step 2: Entity Identification. We first identify *Functional Atoms* based on execution traces, and then modify a Non-dominated Sorting Genetic Algorithm-II, a multi-objective optimization technique, to group *Functional Atoms* as the class entities E_{ser} for each service candidate.

Step 3: Interface Class Identification. For each service candidate, its interface classes I with operations O are identified. Combining them with E_{ser}, a service candidate $SC = (E_{ser}, I, O)$ is produced.

2.3 Step 1: Representative Execution Trace Extraction

This step has the following parts:

Execution Monitoring. We use a tracing tool, Kieker 1.13 [25], to insert probes into the target software. The probes monitor method executions, and the execution paths are recorded in log files.

To obtain execution traces that accurately capture the functionality of the target system, functional test suites and the executable software (executable instance built from the targeted monolithic software) are required. Executing functional test cases under *Functional Testing* will allow us to identify independent functions to be split into service candidates. *Functional Testing*[5] is a black-box testing against the functionality of the system, as opposed to *Unit Testing* that aims to test a class (or class set) or a method (or method set).

Execution Trace Extraction. In an execution log file, each record contains several items, including *ThreadID*—a globally unique ID to identify a thread trace; *Eoi* and *Ess*—the calling order and the depth of the calling stack of the invoked method. From a record set with same *ThreadID*, we can extract an *Execution Trace* $tr : \langle e_1, e_2, ..., e_n \rangle$, a sequence of *Method Calls* that correspond to the executions of a slice of function [1]. By processing all records, we can re-construct an *Execution Trace Set*: $O_{tr} = \{tr\}$.

Execution Trace Reduction. The original execution trace set, O_{tr}, typically contains a large number of redundant execution traces. We created an algorithm, as shown in Algorithm 1, to reduce O_{tr} and form a *Representative Execution Trace Set*, R_{tr}. $R_{tr} \subseteq O_{tr}$.

As shown in Algorithm 1, we use $set(tr_k)$ to denote a set of method calls involved in tr_k. Assume $tr_i \in O_{tr}$, and $tr_j \in R_{tr}$. If $set(tr_i) \subseteq set(tr_j)$, tr_i will not be added into R_{tr}. Conversely, if $set(tr_j) \subset set(tr_i)$, tr_j will be replaced by tr_i. If $set(tr_i) \not\subseteq set(tr_j)$ and $set(tr_i) \not\supseteq set(tr_j)$, then keep tr_j in R_{tr} and add tr_i in R_{tr}. Finally, we obtain the *Representative Execution Trace Set*, R_{tr}.

In the *JPetstore* example, we collected 47 original execution traces. After reduction, we finally obtained 15 representative execution traces.

2.4 Step 2: Entity Identification

To identify class entities, E_{ser}, for a service candidate, we first generate functional atoms based on the reduced execution traces, and then derive four optimization objectives for functional atom grouping. After that, we modify a Non-dominated Sorting Genetic Algorithm-II, a search-based technique, to conduct functional atom grouping. Class

5. https://en.wikipedia.org/wiki/Functional_testing

Algorithm 1: Execution trace reduction algorithm

Input: $O_{tr} = \{tr_1, tr_2, ..., tr_n\}$
Output: $R_{tr} = \{tr_1, tr_2, ..., tr_m\}$, $R_{tr} \subseteq O_{tr}$

```
tr_1 ∈ O_tr, R_tr ← tr_1; // Initialize R_tr.
for each tr_i in O_tr do
    if ∃tr_j ∈ R_tr ∧ set(tr_j) ⊂ set(tr_i) then
        // tr_j is replaced by tr_i.
        R_tr.delete(tr_j);
        R_tr.add(tr_i);
    end
    else if ∃tr_j ∈ R_tr ∧ set(tr_i) ⊆ set(tr_j) then
        continue; // Keep tr_j and ignore tr_i.
    end
    else
        R_tr.add(tr_i); // Keep both tr_i and tr_j.
    end
end
return R_tr;
```

entities in each group will be the class entities within the service candidate.

2.4.1 Functional Atom Generation

We define *Functional Atom*, fa, as a minimal coherent unit, in which all the entities are responsible for the same functional logic. Since an execution trace reflects a slice of software logic, a set of classes that often appear together in the same traces can correspond to a fa.

We employ a classic hierarchical clustering algorithm based on execution traces to generate functional atoms. There are two inputs in the clustering algorithm: *diff*—the threshold condition when the clustering should stop, and $\{ctr\}$: for $\forall ctr_i \in \{ctr\}$, $ctr_i = \{c_1, c_2, ..., c_l\}$ is a set of *Classes* whose methods are involved in the tr_i.

Initially, each functional atom contains just one class: $fa_i = \{c_i\}$. During the clustering process we use a Jaccard Coefficient $f_{jaccard}$ based on T_i of fa_i and T_j of fa_j to compute the similarity between fa_i and fa_j:

$$f_{jaccard}(fa_i, fa_j) = \frac{|T_i \cap T_j|}{|T_i \cup T_j|} \quad (2)$$

where, T_i is a set of ctr that contains c_i.

The fa_i and fa_j with the maximum of $f_{jaccard}$ are merged as a new functional atom, fa'_i. At the same time, T'_i of the new fa'_i is updated: $T'_i = T_i \cup T_j$.

The above clustering is repeated until *newDiff* > *diff*:

$$newDiff = min(\ |T_i \cup T_j| - |T_i \cap T_j|\ ),\ \forall i, j, i < j \quad (3)$$

According to the observations described in Section 4, we suggest setting *diff* = 3. After the clustering, *FA*=$\{fa_1, fa_2, ..., fa_m\}$ is obtained, where $\forall i$, $fa_i = \{c_{i,1}, c_{i,2}, ..., c_{i,i_k}\}$.

In the example of JPetstore, we initialize functional atom with classes: $fa_i = \{c_i\}$. After employing the above algorithm, we finally generate 8 function atoms:

$$fa_1 = \{c_3\},\ \ fa_2 = \{c_4\},\ \ fa_5 = \{c_9\}$$
$$fa_0 = \{c_0, c_1, c_2\},\ \ fa_6 = \{c_{10}, c_{11}, c_{13}, c_{14}\}$$
$$fa_7 = \{c_{12}\},\ \ fa_3 = \{c_5, c_6\}\ ,\ \ fa_4 = \{c_7, c_8, c_{15}\}$$

2.4.2 The Objectives of Functional Atom Grouping

Functional Atom Grouping. This step aims to combine multiple functional atoms into one group, forming E_{ser} for a service candidate. Related functional atoms should be put together while non-related ones should be separated. That is, the grouping process requires maximizing the intra-connectivity inside service candidates while minimizing the inter-connectivity across candidate boundaries. We formalize the grouping objectives based on the information revealed in execution traces.

Structural and Conceptual Intra-Connectivity. In terms of structural and conceptual information recorded in execution traces, we define two objectives based on the intra-connectivity formula in work of Mancoridis [22]. More specifically,

(1) structural intra-connectivity, formulated as $\frac{1}{K}\sum_{i=1}^{K}\frac{u_i}{N_i^2}$. where, N_i is the number of functional atoms inside SC_i. u_i is the number of edges between the functional atoms inside SC_i. In execution traces, a call relationship between a class in fa_i and that in fa_j will indicate an edge between fa_i and fa_j. K is the number of functional atom clusters.

(2) conceptual intra-connectivity. It is similar to the above formula. The only difference is that an edge between fa_i and fa_j exists when the intersection between the term set (a set of textual terms presented in class identifiers) of fa_i and that of fa_j is not empty.

Structural and Conceptual Inter-Connectivity. Similarly, we define other two objectives for functional atom clusters based on the inter-connectivity defined by work of Mancoridis [22]. More specifically,

(1) structural inter-connectivity, formalized as $\frac{1}{K(K-1)/2}\sum_{i\neq j}^{K}\frac{\sigma_{i,j}}{2(N_i \times N_j)}$, where N_i or N_j is the number of elements inside atom cluster i or cluster j. $\sigma_{i,j}$ denotes the number of edges between cluster i and cluster j.

(2) conceptual inter-connectivity. Its formula is similar with the above, except for the edge difference as illustrated in the definition of conceptual intra-connectivity.

Optimization Objectives. In summary, the functional atom grouping has four optimization objectives:

- Maximizing *structural intra-connectivity*.
- Maximizing $-$(*structural inter-connectivity*).
- Maximizing *conceptual intra-connectivity*.
- Maximizing $-$(*conceptual inter-connectivity*).

2.4.3 Search-based Functional Atom Grouping

Functional Atom Grouping aims to produce service candidates through different combinations of functional atoms, by optimizing the above four objectives.

To address this multi-objective problem, we tailored the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [26] to the context of functional atom grouping. NSGA-II has been widely exploited in software remodularization [11]. This efficient genetic technique can search for well-distributed Pareto fronts where approximately optimal solutions are located. The tailoring of NSGA-II to our context of functional atom grouping is described next.

Initial Population. Given the set of functional atoms $FA = \{fa_1, fa_2, ..., fa_m\}$ obtained in section 2.4.1, an *individual* or a *chromosome* is a *partition P* of the set *FA*:

$$P = \{q_1, q_2, ..., q_k\}$$

where, $\forall q_i \neq \emptyset$, $\cup_{i=1,2,...,k}\ q_i = FA$ and $\forall i, \forall j, q_i \cap q_j = \emptyset$.

A partition P of *FA* into N non-empty groups is called a *N-partition* of *FA*. A *N-partition* of *FA* corresponds to producing N service candidates. We randomly generate a set of P from *FA* as *initial population*.

Fitness Functions. We define F as a vector of objective functions, composed of four fitness functions:

$$F = (f_1, f_2, f_3, f_4)$$

where, f_1, f_2, f_3 and f_4 correspond to the previous four optimization objectives, respectively. Functional atom grouping searches for near-optimal solution P to maximize F.

Crossover. We employ single-parent *crossover* to generate *offspring*, each of which is a neighbor partition of an individual P. A *neighbor partition* (*NP*) [22] is still a partition of *FA*, and is generated by moving a functional atom fa from an element of P to a different element of P. The moving fa between elements of P manifests two strategies of monolith splitting: *pull up* and *move* strategies [1]. The move strategy merges one functional atom into a service candidate. The pull up strategy separates out one functional atom to be a new service candidate.

Mutation. With a mutation probability, we randomly select an offspring generated in *Crossover*. Then, we randomly choose one *NP* of this offspring as the mutation result.

Post-processing. The NSGA-II to *Functional Atom Grouping* outputs a Pareto front, consisting of a set of near-optimal solutions (i.e., partitions). We consider one *partition* P as the best solution if P is the *knee point*. As a compromise of all optimization objectives, the knee point P_{knee} [27] presents the smallest euclidean distance from its fitness values to the fitness values (F_{ideal}) of P_{ideal}. P_{ideal} is an ideal solution with best fitness values F_{ideal} in the Pareto front: $F_{ideal} = (max(f_1), max(f_2), max(f_3), max(f_4))$. The classes in each element of the P_{knee} will compose a E_{ser} for a service candidate.

In the example of JPetstore, when initializing populations, we randomly generate N-partitions (Assuming $N = 4$) of *FA* as individuals. An individual is a partition (P) of *FA*:

$$FA = \{fa_1, fa_2, fa_3, fa_4, fa_5, fa_6, fa_7\}.$$

$$P = \{fa_0, fa_5\}, \{fa_1\}, \{fa_3, fa_6\}, \{fa_2, fa_4, fa_7\}\}.$$

After processing the populations through the tailored NSGA-II above, we choose the knee point from the Pareto front as the best solution. By extending the elements of P_{knee} with classes of fa, the class entities in each element will form one identified E_{ser}:

$$E_{ser_3} = \{c_5, c_6, c_9\}$$
$$E_{ser_2} = \{c_7, c_8, c_{15}\}$$
$$E_{ser_1} = \{c_{10}, c_{11}, c_{13}, c_{14}\}$$
$$E_{ser_0} = \{c_0, c_1, c_2, c_3, c_4, c_{12}\}$$

2.5 Step 3: Interface Class Identification

According to R_{tr} and the identified E_{ser_i} for service candidate SC_i, we further recognize the potential interface classes I_i and operations O_i that can be published.

We first detect *entry method*s from execution traces. We consider an entry method always the first m in $tr \in R_{tr}$ to process functional requests from external clients. Methods related with common logic such as configuration and context are excluded from the entry methods, since they are irrelevant to the business capabilities and appear in almost all execution traces.

After that, for a service candidate SC_i, we regard the entry methods located in classes of E_{ser_i} as *Operations* (O_i) to be published.

We identify interface classes of service candidates in a direct way. We intuitively group *Operations* in terms of their owner classes [28], each of which is then regarded as an *Interface Class*. All interface classes compose I_i provided by a service candidate SC_i.

For JPetstore, the identified operations and interface classes are method members and classes with a prefix of `org.mybatis.jpetstore.web.actions`. Table 2 illustrates the identified service candidates from JPetstore. We can observe that the generated service candidates are separated into independent functions: "Catalog service", "Order service", "Account service" and "Cart service". Obviously, even though some classes (e.g., `domain.Account`, `domain.Product`) are organized in the same code package of the original monolith, these classes are divided into different service candidates, each having a clearly-defined functionality.

3 Evaluation Model

In this section, we introduce an evaluation system to assess the quality of service candidates from three aspects:

1) External: *Independence of Functionality*. A service should provide well-defined, independent, and coherent functionality to its external clients, following the *Single Responsibility Principle* (SRP).

2) Internal: *Modularity*. If a service is well-modularized, its internal entities should be cohesive, and entities across service boundaries should be loosely coupled.

3) Evolvability: *Independence of Evolvability*. Being able to evolve independently, without changing other services, is the most desirable property of services, meaning that they can flexibly accommodate future changes.

3.1 Independence of Functionality

To quantitatively and objectively assess functional independence, we leverage information from published interfaces that expose the functionality of a module (e.g., a service) [29]. From the interface classes I of identified service candidates, we use three measures to objectively quantify *Independence of Functionality* [1] as follows.

1) ***ifn*** (*i*nter*f*ace *n*umber), measures the number of published interfaces of a service. The smaller the *ifn*, the more likely the service assumes a single responsibility. ***IFN*** is the average of all *ifn*. The formal definitions are as follows:

$$IFN = \frac{1}{N}\sum_{j=1}^{N} ifn_j \tag{4}$$

$$ifn_j = |I_j| \tag{5}$$

TABLE 2: The service candidates extracted from *JPetstore*

SC	E_{ser}	I	O
SC_0	org.mybatis.jpetstore.domain.Category org.mybatis.jpetstore.service.CatalogService org.mybatis.jpetstore.web.actions.CatalogActionBean org.mybatis.jpetstore.domain.Product org.mybatis.jpetstore.domain.Item org.mybatis.jpetstore.domain.Sequence	CatalogActionBean	ForwardResolution viewCategory() ForwardResolution searchProducts() ForwardResolution viewProduct() ForwardResolution viewItem()
SC_1	org.mybatis.jpetstore.domain.LineItem org.mybatis.jpetstore.web.actions.OrderActionBean org.mybatis.jpetstore.service.OrderService org.mybatis.jpetstore.domain.Order	OrderActionBean	Resolution newOrder() boolean isConfirmed() org.mybatis.jpetstore.domain.Order getOrder() Resolution newOrderForm() void clear() void setOrderId(int) Resolution viewOrder() Resolution listOrders()
SC_2	org.mybatis.jpetstore.domain.Cart org.mybatis.jpetstore.domain.CartItem org.mybatis.jpetstore.web.actions.CartActionBean	CartActionBean	void clear() Resolution removeItemFromCart() Resolution updateCartQuantities() org.mybatis.jpetstore.domain.Cart getCart() Resolution addItemToCart()
SC_3	org.mybatis.jpetstore.service.AccountService org.mybatis.jpetstore.web.actions.AccountActionBean org.mybatis.jpetstore.domain.Account	AccountActionBean	boolean isAuthenticated() String getUsername() void setPassword(String) void setUsername(String) Resolution newAccount() org.mybatis.jpetstore.domain.Account getAccount() Resolution signoff() void clear()

where, I_j is the published interfaces (or interface classes) of service j. N is the number of services that provide published interfaces in the service-based system.

2) ***chm*** (*co*hesion at *m*essage level), measures the cohesiveness of interfaces published by a service at the message level. The higher the *chm* of a service, the more cohesive the service is, from an external perspective. ***CHM*** is the average functional cohesiveness. We define *chm* as a variation of LoC_{msg} (Lack of Message-level Cohesion), proposed by Athanasopoulos et al. [30]. $chm + LoC_{msg} = 1$.

$$CHM = \frac{1}{N}\sum_{j=1}^{N} chm_j \tag{6}$$

$$chm_j = \begin{cases} \frac{\sum_{(k,m)} f_{msg}(opr_k, opr_m)}{\frac{1}{2}|O_j|\times(|O_j|-1)}, & if\ |O_j| \neq 1 \\ 1, & if\ |O_j| = 1 \end{cases} \tag{7}$$

$$f_{msg}(opr_k, opr_m) = \frac{(\frac{|ret_k \cap ret_m|}{|ret_k \cup ret_m|} + \frac{|par_k \cap par_m|}{|par_k \cup par_m|})}{2} \tag{8}$$

where, $opr_k, opr_m \in O_j$ are union operations on I_j, and $k < m$. ret_m and par_m are sets of return values and input parameters of opr_m. f_{msg} computes the similarity between two operations at message level, denoting the average of similarity of input messages (parameters) and output messages (return values). N is the same with that of *IFN*.

3) ***chd*** (*co*hesion at *d*omain level), measures the cohesiveness of interfaces provided by a service at the domain level. The higher the *chd*, the more functionally cohesive this service is. Similarly, ***CHD*** is the average of all *chd* within the system. We define *chd* as a variation of LoC_{dom} (Lack of Domain-level Cohesion) defined by Athanasopoulos et al. [30]. $chd + LoC_{dom} = 1$.

$$CHD = \frac{1}{N}\sum_{j=1}^{N} chd_j \tag{9}$$

$$chd_j = \begin{cases} \frac{\sum_{(k,m)} f_{dom}(opr_k, opr_m)}{\frac{1}{2}|O_j|\times(|O_j|-1)}, & if\ |O_j| \neq 1 \\ 1, & if\ |O_j| = 1 \end{cases} \tag{10}$$

$$f_{dom}(opr_k, opr_m) = \frac{|f_{term}(opr_k) \cap f_{term}(opr_m)|}{|f_{term}(opr_k) \cup f_{term}(opr_m)|} \tag{11}$$

where, opr and O are same symbols as those defined in *chm*. f_{dom} computes the similarity between operations. $f_{term}(opr_i)$ describes the set of domain terms contained in the signature of opr_i.

3.2 Modularity

Modularity of a component or service can be measured from multiple perspectives, such as structural, conceptual, history, and dynamic dimensions [11]. Here we extend the Modularity Quality (*MQ*) defined by Mancoridis [22] with structural and conceptual dependencies, using Structural Modularity Quality and Conceptual Modularity Quality to assess the modularity of services candidates.

1) ***SMQ*** (*S*tructural *M*odularity *Q*uality), measures modularity quality from a structural perspective. The higher the *SMQ*, the better modularized the service is.

$$SMQ = \frac{1}{N}\sum_{i=1}^{N} scoh_i - \frac{1}{N(N-1)/2}\sum_{i\neq j}^{N} scop_{i,j} \tag{12}$$

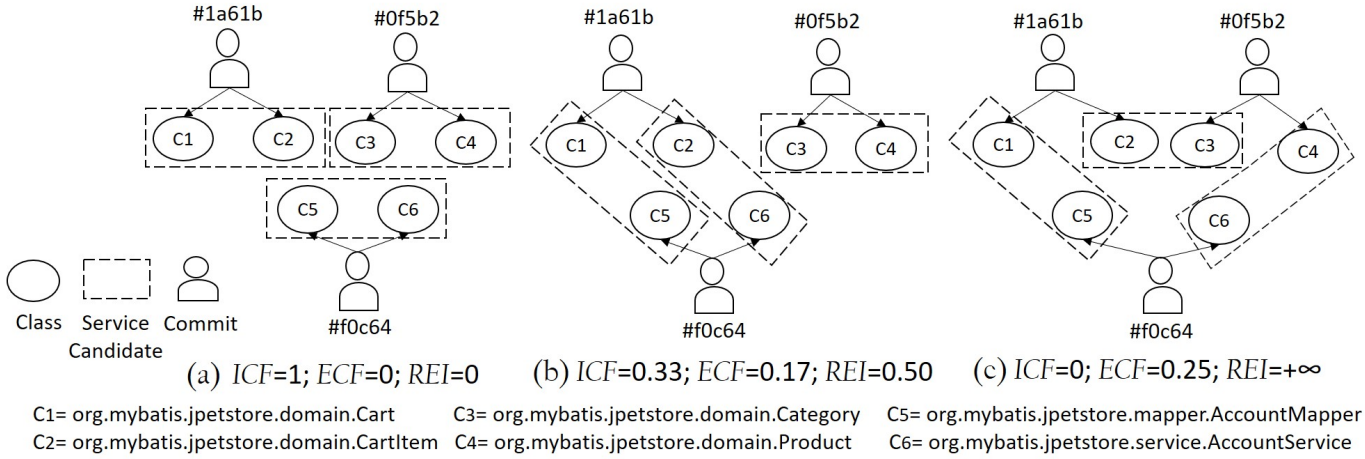


Fig. 2: *Independence of Evolvability* measures of different splitting for `JPetstore` example

$$scoh_i = \frac{u_i}{N_i^2}, \quad scop_{i,j} = \frac{\sigma_{i,j}}{2(N_i \times N_j)}$$

Consistent with intra-connectivity and inter-connectivity [22], ***scoh*** measures the structural cohesiveness of a service, while $\boldsymbol{scop}_{i,j}$ measures coupling between services. u_i is the number of edges inside a service i. $\sigma_{i,j}$ is the number of edges between service i and service j. N_i or N_j is the number of entities inside service i or j. If there is a structural call dependency between two entities, an edge exists. The bigger *scoh*, and the smaller *scop*, the better.

2) ***CMQ*** (Conceptual *M*odularity *Q*uality), similarly measures modularity quality from a conceptual perspective. The higher the *CMQ*, the better.

$$CMQ = \frac{1}{N}\sum_{i=1}^{N} ccoh_i - \frac{1}{N(N-1)/2}\sum_{i \neq j}^{N} ccop_{i,j} \tag{13}$$

$$ccoh_i = \frac{u_i}{N_i^2}, \quad ccop_{i,j} = \frac{\sigma_{i,j}}{2(N_i \times N_j)}$$

The formalisms of ***CMQ***, ***ccoh***, and ***ccop*** are similar to *SMQ*, *scoh*, and *scop*. The only difference is that in the *CMQ* definition an edge between two entities exists if the intersection between the textual term set of the entities is not empty.

3.3 Independence of Evolvability

Ideally, we should evaluate *Independence of Evolvability* of services by examining and tracking the revision history of the implemented services, but this is impossible at the service design phase. We thus propose to measure a service's (or candidate's) *Independence of Evolvability* by studying the revision history of its original monolithic software. The rationale is: if classes that change together frequently are grouped into one service, and other classes (that do not change together with these chosen classes) are grouped into different services, we assume that the designed services will evolve relatively independently. Based on this rationale, we propose three measures as follows.

1) ***icf*** **(*i*nternal *c*o-change *f*requency)**, measures how often entities inside a service change together as recorded in the revision history. Higher *icf* means that the entities inside this service will be more likely to evolve together. ***ICF*** is the average of all *icf* within the system.

$$ICF = \frac{1}{N}\sum_{j=1}^{N} icf_j \tag{14}$$

$$icf_j = \frac{1}{|E_{ser_j}|}\sum_{m=1}^{|E_{ser_j}|}\frac{1}{|E_{ser_j}|}\sum_{n=1}^{|E_{ser_j}|} f_{cmt}(c_m, c_n) \tag{15}$$

where, $f_{cmt}(c_m, c_n)$ is the number of commits in which entity c_m and c_n changed together. $c_m, c_n \in E_{ser_k}$. $c_m, c_n \in E_{ser_j}$. $f_{cmt}(c_m, c_n) = 0$ if $m = n$.

2) ***ecf*** **(*e*xternal *c*o-change *f*requency)**, measures how often entities assigned to different services change together, according to the revision history. A lower *ecf* score means that entity pairs located in different services are expected to evolve more independently. Similarly, ***ECF*** is the average *ecf* of all services within the system.

$$ECF = \frac{1}{N}\sum_{j=1}^{N} ecf_j \tag{16}$$

$$ecf_j = \frac{1}{|E_{ser_j}|}\sum_{m=1}^{|E_{ser_j}|}\frac{1}{|E^c_{ser_j}|}\sum_{n=1}^{|E^c_{ser_j}|} f_{cmt}(c_m, c_n) \tag{17}$$

where, ecf_j computes the co-change frequency between entities in E_{ser_j} of service j and entities in $E^c_{ser_j}$. $E^c_{ser_j}$ is a set of entities not located in E_{ser_j}. $E^c_{ser_j} = \cup_{k=1,2,\ldots,N} E_{ser_k},\ k \neq j$. $f_{cmt}(c_m, c_n)$ is same as that defined in *icf*. $c_m \in E_{ser_j},\ c_n \in E^c_{ser_j}$. $ecf_j = 1$ if $|E_{ser_j}| = 1$.

3) ***REI*** **(Ratio of *ECF* to *ICF*)**, measures the ratio of co-change frequency $across$ services vs. the co-change frequency $within$ services. The ratio is expected to be less than 1.0, if co-changes happen more often inside a service than across different services. The smaller the ratio is, the less likely co-changes happen across services, and the extracted services tend to evolve independently. Ideally, all co-changes should happen inside services.

$$REI = ECF/ICF \tag{18}$$

where, *ECF* and *ICF* are notions defined above.

4) ***An example with JPetstore***. We will use the *JPetstore* example, introduced in section 2, to intuitively explain *ICF*, *ECF*, and *REI*. Figure 2 shows that this example includes 6 classes and 3 commits, and the classes will be grouped into 3 service candidates. Figure 2 shows three splitting scenarios. In Figure 2(a), each commit revision happens within a service, which means each service can change independently. At the other extreme, in Figure 2(c), each commit revision crosses a service boundary, which means changes to each service will always influence another service. Figure 2(b) is an intermediate case: one service can change by itself while the others cannot. From Case a to Case b to Case c, the value of *ICF* becomes smaller while the values of *ECF* and *REI* become larger. It can be seen that the measures (*ICF*, *ECF*, *REI*) are able to reflect the change of *Independence of Evolvability* in different splittings.

4 Experimental Setup

In this section, we first introduce the investigated subjects, and then present how we obtained test cases and collected execution traces. After that, we introduce the parameter configurations used in our method, illustrate the baseline methods, and present the evolution history data we collected. Finally, we show the evaluation configuration in the experiments.

4.1 Subjects

We collected six web applications for our experiments. *Project_Introduction* of Table 3 illustrates these projects. Column *Version* shows the version of the project that we analyzed. Column *Start date* and *End date* denote the time range under examination. *LOC* and *#Class* are the lines of code and the number of classes implemented in Java. Springblog[6] and Solo[7] are blogging systems. JForum[8] is a discussion board. Apache Roller[9] is a full-featured, multi-user and group-blog server. Agilefant[10] is an open-source agile project management tool. Xwiki-platform[11] is a generic wiki platform offering runtime services. Xwiki-platform project contains more than 100 modules and supports extensions. These projects are heterogeneous in their sizes and business domains. Most of these systems—such as JForum, Apache Roller, Agilefant, and Xwiki-platform—are popular and widely used in practice.

These web applications follow classical multi-layered architectures, such as three-layer (Presentation-Business-Persistence) and four-layer (Presentation-Application-Business-Persistence) [31][32]. We chose web applications as subjects because the back end (server side) of a web application is typically packaged into a single unit, such as a WAR or EAR file. Such monolithic web applications usually suffer from maintainability and scalability issues because of their rapid growth.

6. https://github.com/bvn13/SpringBlog
7. https://github.com/b3log/solo
8. https://sourceforge.net/projects/jforum2
9. https://roller.apache.org
10. https://www.agilefant.com
11. http://platform.xwiki.org

Our experiments decompose the back end of these subjects into service candidates, excluding the front end (Presentation Layer) and databases. Various patterns [33] can be adopted for designing the front end decomposition, which is beyond the scope of our current research. Object-Relational Mapping[12] is employed in the investigated systems to interact with the databases, and the schema of the database in each case corresponds to *Entity* [32] classes. Thus, when we decompose software systems at the class level, the database tables would also be naturally re-organized, forming a new schema for each service candidate.

4.2 Test Case and Execution Trace Collection

As mentioned earlier, *Functional Testing* is required in our method. Functional Testing is black-box testing of the entire application. Different from *Unit Testing*, Functional Testing requires that the tested application is physically executable. In the experiment, we use both automatic and manual approaches to conduct the testing:

(1)*Automation Functional Testing for Xwiki-platform*. The repository of the Xwiki-platform project includes functional test cases in 38 modules, which we have confirmed with their developer community. It has an automation test infrastructure, so these test cases can be carried out automatically[13]. In total, we executed 4020 functional test cases in the experiment.

(2)*Manual Functional Testing for Other Projects*. Springblog, Solo, JForum, Apache Roller and Agilefant projects contain unit testing suites, but do not provide test suites for functional testing. We thus designed and manually conducted functional testing for each subject project, following four steps :

a) Determine the functionality that needs to be tested by checking the project specification documentation;

b) For each function to be tested, design the test scenarios to cover the maximum amount of application functionality;

c) Build the project to generate an executable WAR package, and deploy it in Apache Tomcat;

d) Manually execute the test scenarios by manipulating the application through a web browser. We used Apache JMeter[14] to record the operations while executing each test scenario. JMeter supports a fully featured test IDE that allows test plan recording from browsers. We configured JMeter and selected "Functional Testing" for "Test Plan Object", and set the proxy server. As a user explores the GUI through the browser, JMeter intercepts the HTTP(S) requests and records *transactions* in test scripts. Each recorded *transaction* corresponds to a test case.

These recorded test cases can be repeatedly executed using JMeter to automate the testing. In total, we manually executed 250 test scenarios and recorded 1,886 test cases by JMeter in the experiment.

For all subjects, *Collected_Execution_Trace* in Table 3 presents the execution data extracted from the monitoring log. *#TC(#TS)* is the number of functional test cases (Test Scenarios). $|O_{tr}|$ is the number of extracted *Execution Traces* (O_{tr}). $|O_{call}|$ is the number of method calls invoked in

12. https://en.wikipedia.org/wiki/Object-relational_mapping
13. https://dev.xwiki.org/xwiki/bin/view/Community/Testing
14. https://jmeter.apache.org/

TABLE 3: Summary of projects and collected execution traces

Subject	Project_Introduction					Collected_Execution_Trace				
	Version	Start date	End date	#Class*	LOC	#TC(#TS)	$\|O_{tr}\|$	$\|O_{Call}\|$	$\|R_{tr}\|$	$\|R_{call}\|$
Springblog	2.8.0	2015-09-26	2017-12-22	85	3,583	371(28)	102	1,390	33	311
Solo	2.7.0	2012-04-10	2018-03-06	148	17,046	257(42)	166	18,650	71	7,425
JForum	2.1.9	2003	2010-10-05	340	29,550	201(37)	323	484,157	69	119,160
Apache Roller	5.2.0	2005-06-07	2017-11-06	534	47,602	174(78)	511	1,499,763	87	305,064
Agilefant	3.5.4	2006-10-12	2015-07-03	389	26,327	883(65)	859	29,618	113	4,899
Xwiki-platform	10.8	2006-10-13	2018-09-24	2,749	368,432	4,020(-)	26,809	3,347,220,274	2,766	778,162,767
Total						5,906(-)	28,770	3,349,253,852	3,139	778,599,626

* Inner classes and classes located in test suites are excluded.

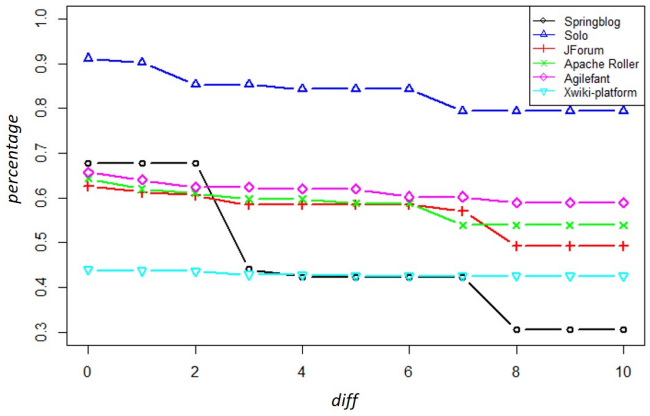


Fig. 3: The percentage of functional atoms change with *diff* increases

O_{tr}. $|R_{tr}|$ and $|R_{call}|$ count the *Representative Execution Trace* (R_{tr}) and method calls in R_{tr}. In total, 3,139 representative execution traces were obtained from 28,770 original execution traces.

To ensure reproducibility, all projects, specification documents, the description of test scenarios, the test plan files containing test cases recorded by JMeter, and the collected execution traces can be found at *https://github.com/wj86/FoSCI*.

4.3 Parameter Setting in Our Method

The FoSCI method requires several parameters: *diff, population size, crossover probability, mutation probability, maximum generations*.

diff is the stopping condition of *Functional Atom Generation*. We have observed that the minimum and coherent functional atoms are formed when *diff* is equal to 3. Figure 3 illustrates the change in the number of functional atoms (as a percentage of the initial number of classes) as *diff* is set from 1,2,..., to 10 in all investigated subjects. It can be seen that the number of functional atoms flattens when *diff*=3. Thus, we recommended and set *diff* to be 3.

Search-based Functional Atom Grouping involves parameters due to the NSGA-II technique. We calibrated these parameters based on existing work on NSGA-II. We set the *population size* equal to 20 individuals, since this setting achieves a balance between effectiveness and efficiency according to our experiments. We used 0.8 as *crossover probability* and $0.04 \times log_2(n)$ as the *mutation probability* as suggested by Candela et al.[11]. We configured *maximum generations* to be 200. The search process is repeated 30 times to reduce the bias caused by the randomness of the genetic algorithm. As a result, we got 30 sets of the nearest optimal solutions after completing our method execution for each subject. We merged the solutions and chose the non-dominated individuals. Among them, we selected the *knee point* (as explained in section 2.4.3) as the solution of service candidate identification.

TABLE 4: The class percentage by different methods

Subject	LIMBO	WCA	MEM	FoSCI
Springblog	91.76%	91.76%	85.88%	72.94%
Solo	98.65%	98.65%	74.32%	68.92%
JForum	97.15%	97.15%	60.76%	61.47%
Apache Roller	96.22%	96.22%	77.94%	77.15%
Agilefant	94.34%	94.34%	74.81%	61.44%
Xwiki-platform	97.93%	97.93%	81.41%	46.82%

4.4 Baseline Methods

We compare FoSCI with three baseline methods: **LIMBO** [34], **WCA** [12], and **MEM** (Microservice Extraction Model) [5]. **WCA** is a hierarchical clustering method that leverages two measures to determine the similarity between classes: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). We employed UENM in our work since it outperforms UE [13]. **LIMBO** uses information loss to measure the distance between classes for software clustering. **MEM** cuts the class graph of the original monolith, and the relations on the edges are extracted using three strategies: logical, semantic, and contributor couplings. In this paper, the semantic coupling-based approach was selected in MEM because the other two strategies cover a very small number of the classes in our target projects.

Table 4 shows the class percentage covered by all four methods: WCA, LIMBO, MEM and FoSCI. Class percentage refers to the proportion of classes that are analyzed by various methods. The class percentage rarely equals 100%. The class percentage in LIMBO and WCA is always the same, as both methods are based on structural dependencies parsed from source code. The percentages in MEM and FoSCI are less than those of LIMBO and WCA. We will discuss the uncovered classes in FoSCI in Section 5.

To ensure that all methods identify service candidates from the same set of classes and the comparisons are unbi-

TABLE 5: Analysis of commit history

Subject	#Total	#Co-change	#Class (%)
Springblog	148	84	57 (67.06%)
Solo	2,044	282	147 (99.32%)
JForum	10	0	0 (0%)
Apache Roller	4,126	337	379 (70.91%)
Agilefant	5,166	1,853	314 (80.72%)
Xwiki-platform	35,290	12,045	1,827 (66.46%)
Total	46,784	14,601	-

ased, we use the intersection of the classes of all methods as our data standard. In the experiments, we first run these methods using all classes, and then filter out the results based on the classes shared by all methods to evaluate their performance.

4.5 Revision History Collection

We collected commit information from the revision history to evaluate the *Independence of Evolvability*. As shown in Table 5, *#Total* is the number of total commits during the evolution period. Many commits are irrelevant to class evolution, such as adding new files, modifying front-end files (.css, .html), modifying configuration files (.xml), and updating licenses[15]. By removing these irrelevant commits, $14601/46787 = 31.21\%$ of all commits are selected. Because of the incomplete revision history of JForum, the number of co-change commits and involved classes is 0.

4.6 Evaluation Configuration

The baseline methods require configuring the number N of service candidates to be identified. Similar to the software modularization, it is not a trivial task to define N, since it is highly related to the targeted subjects[11]. Modularization work in [11] uses $\frac{M}{2}$ (M is the number of elements or classes to be clustered) as the maximum number of generated modules. For identifying services from a monolithic software, the domain experts of the investigated project should ideally recommend N in terms of their knowledge of their system's business logic[20]. We set N based on the functionality features (as illustrated in our data repository) for the subjects in study. To address the comparison more rigorously, we supplemented four extra settings by using $N-2$, $N-1$, $N+1$ and $N+2$ as inputs. As a result, Springblog, Solo, JForum, Agilefant, Apache Roller and Xwiki-platform were split into 6-10, 7-11, 7-11, 9-13, 14-18, and 50-54 service candidates respectively.

Thus we conducted $5 \times 6 = 30$ group experiments in our evaluation. For each group, WCA, LIMBO, MEM and our method are applied. In each group, our method is repeated 30 times, with the configurations described in section 4.3.

5 Evaluation

The objective of our evaluation is to assess whether FoSCI can produce effective service candidates, in terms of *Independence of Evolvability*, *Independence of Functionality*, and *Modualarity*. We compare FoSCI with three baseline methods, LIMBO, WCA, and MEM, and explore how the performance of FoSCI changes when the coverage of execution traces differs.

As described in section 4.6, we conducted 5 test groups for each subject. Section 5.1 illustrates all 5 groups for each subject. Due to the consistent performance of methods in different N for a subject, section 5.2 and 5.3 just illustrate 3 groups ($N-2, N, N+2$). The complete tables with all groups are available in our repository.

5.1 The Evaluation of Evolvability

Table 6 presents the *ICF,ECF* and *REI* measures for service candidates generated by all four methods. Each row corresponds to the measures of one case (group). For example, the first row of *Springblog* shows the evaluation results when identifying $N = 6$ service candidates. To rigorously compare FoSCI with other methods, we employ Wilcoxon's signed-rank test, a non-parametric statistical hypothesis to test whether the sample of one metric measurement from FoSCI is significantly better than the sample of that from each of the other methods over all cases. The P-values are illustrated in the last row of the table.

5.1.1 Analysis of Results

Recall that *REI* measures the evolvability feature by integrating *ICF* and *ECF*. The smaller the *REI*, the better. From Table 6, we observe that only FoSCI has all *REI* scores less than 1.0 except for one outlier (in gray color). Most scores in LIMBO, WCA and MEM are much larger than 1.0, indicating poor evolvability of the service candidates they generated. The P-value is less than 0.001 (denoted as ***) respectively when FoSCI is compared with LIMBO, WCA and MEM with regard to *REI*, suggesting that FoSCI significantly outperforms the other methods.

According to Table 6, service candidates produced by FoSCI perform by far the best in terms of *REI* among all methods. Moreover, FoSCI can ensure that co-changes are better constrained within services instead of crossing service boundaries. In addition, Table 6 shows that, even though for different values of N of one subject, the performance of each method is quite consistent.

ICF, *ECF* and *REI* measures the overall level of evolvability. It is interesting to investigate the performance of each individual service candidate (measured by *icf,ecf*) in each case. We selected 4 large subjects to present their distribution of icf and ecf: Solo, Apache Roller, Agilefant, and Xwiki-platform. Since the performance of each method is consistent under different target number of services, N, as shown in Table 6, we present the results using a randomly picked N, as shown in Figure 4.

Figure 4 shows that the co-changes (indicated by icf) inside service candidates are more frequent than those across services (indicated by ecf) in FoSCI. Its positive difference from icf to ecf by FoSCI is significantly higher than others. Moreover, WCA and MEM may generate service candidates with negative difference from icf to ecf, indicating that these services can hardly evolve independently.

The box plots reveal that the difference from *icf* to *ecf* by FoSCI is bigger (with expected positive value) than that by the baseline methods. This observation is consistent with that from Table 6.

15. In project Solo, *commit #5bcd9d6* shows the commit message "update license", modifying 128 classes out of total 147 (87.04%).

TABLE 6: Measurement results of *ICF*, *ECF* and *REI*

Subject	LIMBO			WCA			MEM			FoSCI		
	ICF	ECF	REI	ICF	ECF	REI	ICF	ECF	REI	ICF	ECF	REI
Springblog	0.0856	0.1417	1.6554	0.0270	0.3966	14.6701	0.0940	0.5154	5.4829	0.3795	0.1304	0.3435
	0.0719	0.1295	1.8000	0.0228	0.3689	16.2013	0.1584	0.4575	2.8883	0.4347	0.1488	0.3423
	0.0691	0.2455	3.5554	0.0228	0.3689	16.2013	0.1424	0.5250	3.6878	0.4037	0.1312	0.3251
	0.0692	0.2318	3.3518	0.0228	0.3689	16.2013	0.1300	0.5773	4.4410	0.3645	0.1559	0.4277
	0.0654	0.2141	3.2748	0.0288	0.3253	11.2890	0.1736	0.5289	3.0477	0.3229	0.1415	0.4382
Solo	0.1780	0.1718	0.9650	0.0706	0.1960	2.7745	0.0968	0.7640	7.8958	0.2460	0.1877	0.7627
	0.1632	0.1706	1.0455	0.0288	0.3105	10.7822	0.1479	0.6733	4.5535	0.2789	0.2133	0.7646
	0.1687	0.1712	1.0149	0.0288	0.3105	10.7822	0.1314	0.7106	5.4096	0.3369	0.1669	0.4954
	0.1682	0.1781	1.0584	0.0252	0.5205	20.6533	0.1186	0.7386	6.2265	0.2311	0.1571	0.6797
	0.1687	0.1764	1.0454	0.0252	0.5205	20.6533	0.1082	0.7616	7.0381	0.4001	0.1956	0.4888
Agilefant	0.2214	0.2310	1.0433	0.0509	0.0366	0.7197	0.0264	0.8964	34.0120	2.5298	0.2389	0.0944
	0.2256	0.2319	1.0280	0.0452	0.1422	3.1442	0.0239	0.9060	37.8620	1.8916	0.2528	0.1336
	0.2301	0.2345	1.0190	0.0452	0.1422	3.1442	0.0219	0.9144	41.7175	1.9864	0.2314	0.1165
	0.2264	0.2306	1.0183	0.0407	0.1291	3.1725	0.0203	0.9211	45.4483	2.0392	0.2471	0.1212
	0.2164	0.2216	1.0241	0.0375	0.1180	3.1493	0.0189	0.9268	49.1187	1.5147	0.2254	0.1488
Apache Roller	0.7703	0.7920	1.0282	0.2777	0.6959	2.5064	0.2619	0.9338	3.5661	1.1587	0.9099	0.7853
	0.7714	0.7953	1.0310	0.2182	0.7173	3.2883	0.2444	0.9386	3.8407	0.9236	0.8128	0.8800
	0.7631	0.7910	1.0366	0.2182	0.7173	3.2883	0.2291	0.9423	4.1125	1.0313	0.8567	0.8307
	0.7646	0.7935	1.0378	0.1703	0.7386	4.3377	0.2157	0.9674	4.4855	0.8970	0.7920	0.8829
	0.7816	0.8063	1.0317	0.1596	0.7902	4.9502	0.2035	0.9699	4.7652	0.7591	0.7647	1.0074
Xwiki-platform	0.0083	0.0072	0.8632	0.0402	0.1036	2.5760	0.0621	0.6807	10.9657	0.1461	0.0055	0.0376
	0.0081	0.0071	0.8764	0.0394	0.1016	2.5772	0.0609	0.6869	11.2879	0.1045	0.0060	0.0571
	0.0081	0.0071	0.8797	0.0387	0.1189	3.0747	0.0597	0.6930	11.6101	0.1059	0.0055	0.0516
	0.0083	0.0071	0.8609	0.0379	0.1543	4.0688	0.0714	0.6799	9.5243	0.1038	0.0058	0.0554
	0.0086	0.0071	0.8338	0.0362	0.1517	4.1904	0.0701	0.6858	9.7888	0.1190	0.0052	0.0441
P-value			$>$, ***			$>$, ***			$>$, ***			–

$>$ means the value of former (LIMBO,WCA,MEM) is statistically bigger than the latter (FoSCI), and $<$ vice versa.
$=$ means there is no statistical difference between the two group of results.
*** means the 0.001 significant level (p-value < 0.001).
** means the 0.01 significant level (p-value < 0.01).
* means the 0.05 significant level (p-value < 0.05).
Note: JForum is excluded from this table, since the co-change revision data was missing, as shown in Table 5.

5.1.2 Summary of the Evaluation of Evolvability

As indicated by *ICF (and icf), ECF (and ecf)* and *REI*, FoSCI can aggregate frequently co-changed entities within a monolith into one service candidate, while placing infrequently co-changed entities into different candidates. In contrast, for service candidates generated by LIMBO, WCA and MEM, change across service boundaries are more common than those within services. Consequently, these services are unable to evolve independently. In conclusion, FoSCI can generate service candidates with significantly greater Independence of Evolvability than the other three baseline methods.

5.2 The Evaluation of Functionality

The analysis in this section is similar to that in section 5.1. Table 7 shows *IFN*, *CHM* and *CHD* measures for service candidates identified by the four methods. To further observe individual service candidates, Figure 5 shows the distribution of ifn, chm and chd measures of service candidates in four cases, the same as we reported in section 5.1.

5.2.1 Analysis of Results

Recall that the smaller the *IFN*, the better a split will be. Column *IFN* of Table 7 shows that, compared with WCA and MEM, FoSCI and LIMBO have the better performance considering *IFN*. The *IFN* value of FoSCI and LIMBO can be even about 10 times smaller than that of WCA and MEM in Agilefant and Apache Roller cases. The P-values indicate that, in all statistical tests, except for the comparison with LIMBO, FoSCI significantly outperforms WCA and MEM in terms of *IFN*.

Similarly, Figure 5(a)s shows ifn distribution for each service candidate in the same four cases as reported in section 5.1. Consider a large-size Xwiki-platform case: WCA assigns most published functionality into a singe service, so that this service candidate has to provide more than 700 interfaces ($ifn > 700$). The same phenomenon can be seen in other cases too. Both FoSCI and LIMBO, however, identify services with less interfaces. We can see these results from ifn box plots are consistent with those from *IFN* analysis.

In terms of Column *CHM* and *CHD*, Table 7 shows that the three baseline methods perform the worst in several cases, but FoSCI never ranks the worst except for one case highlighted in *gray*. The P-values indicate that FoSCI significantly surpasses LIMBO. But there is no statistical difference between the results of FoSCI and MEM (and WCA).

To understand these observations, we investigate the distribution of chd and chm shown in Figure 5(b)s and (c)s. In particular, for Xwiki-platform, the result of MEM is much higher than that of FoSCI, because MEM produced one large service (providing more than 700 interfaces) and a lot of tiny services (providing 1 interface), as indicated in

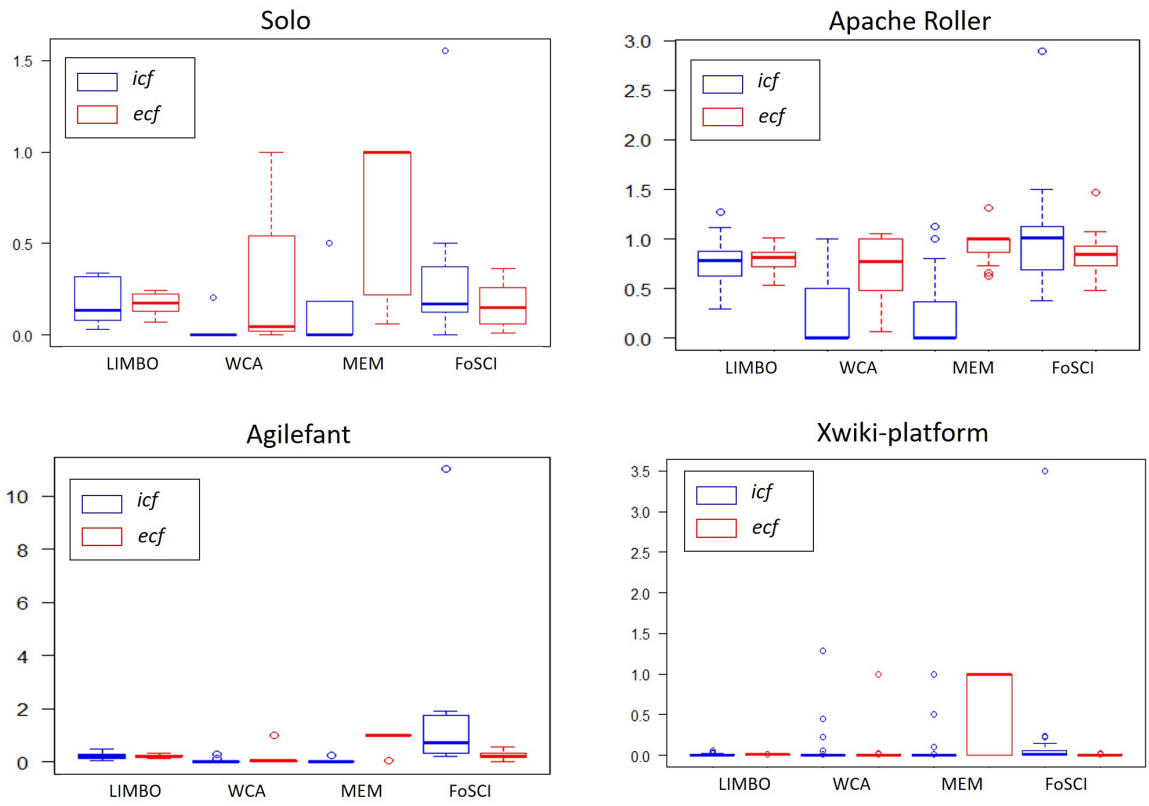


Fig. 4: The distribution of *icf*,*ecf* measures for service candidates

TABLE 7: Measurement results of *IFN*, *CHM* and *CHD*

Subject	IFN				CHM				CHD			
	LIMBO	WCA	MEM	FoSCI	LIMBO	WCA	MEM	FoSCI	LIMBO	WCA	MEM	FoSCI
Springblog	2.600	13.000	4.000	2.600	0.494	0.297	0.656	0.681	0.472	0.308	0.556	0.632
	2.167	13.000	3.000	2.167	0.512	0.297	0.747	0.573	0.468	0.308	0.670	0.558
	2.167	13.000	3.000	2.167	0.512	0.297	0.747	0.589	0.468	0.308	0.669	0.607
Solo	3.286	23.000	23.000	5.750	0.823	0.847	0.847	0.864	0.452	0.449	0.449	0.508
	2.875	23.000	23.000	4.600	0.849	0.847	0.847	0.880	0.526	0.449	0.449	0.595
	2.556	23.000	23.000	4.600	0.868	0.847	0.847	0.894	0.579	0.449	0.449	0.648
JForum	3.857	13.500	27.000	4.500	0.531	0.758	0.508	0.701	0.227	0.578	0.152	0.463
	3.375	13.500	27.000	3.857	0.526	0.758	0.508	0.610	0.233	0.578	0.152	0.440
	3.000	13.500	27.000	3.857	0.575	0.758	0.508	0.559	0.225	0.578	0.152	0.457
Agilefant	3.889	17.500	35.000	4.375	0.727	0.619	0.735	0.721	0.282	0.433	0.199	0.388
	3.889	17.500	35.000	4.375	0.727	0.619	0.735	0.861	0.282	0.433	0.199	0.486
	3.500	17.500	35.000	4.375	0.740	0.619	0.735	0.809	0.283	0.433	0.199	0.509
Apache Roller	1.667	15.000	15.000	2.143	0.744	0.779	0.779	0.748	0.558	0.385	0.385	0.579
	1.667	15.000	15.000	2.143	0.744	0.779	0.779	0.724	0.558	0.385	0.385	0.525
	1.500	15.000	15.000	1.667	0.753	0.779	0.779	0.791	0.602	0.385	0.385	0.585
Xwiki-platform	16.580	21.256	27.667	16.939	0.176	0.367	0.701	0.256	0.091	0.372	0.751	0.253
	16.255	21.256	27.667	16.600	0.177	0.367	0.701	0.231	0.091	0.372	0.751	0.216
	15.942	20.725	26.774	16.275	0.178	0.361	0.711	0.233	0.092	0.378	0.759	0.250
P-value	<, ***	>, ***	>, ***	–	<, ***	=	=	–	<, ***	=	=	–

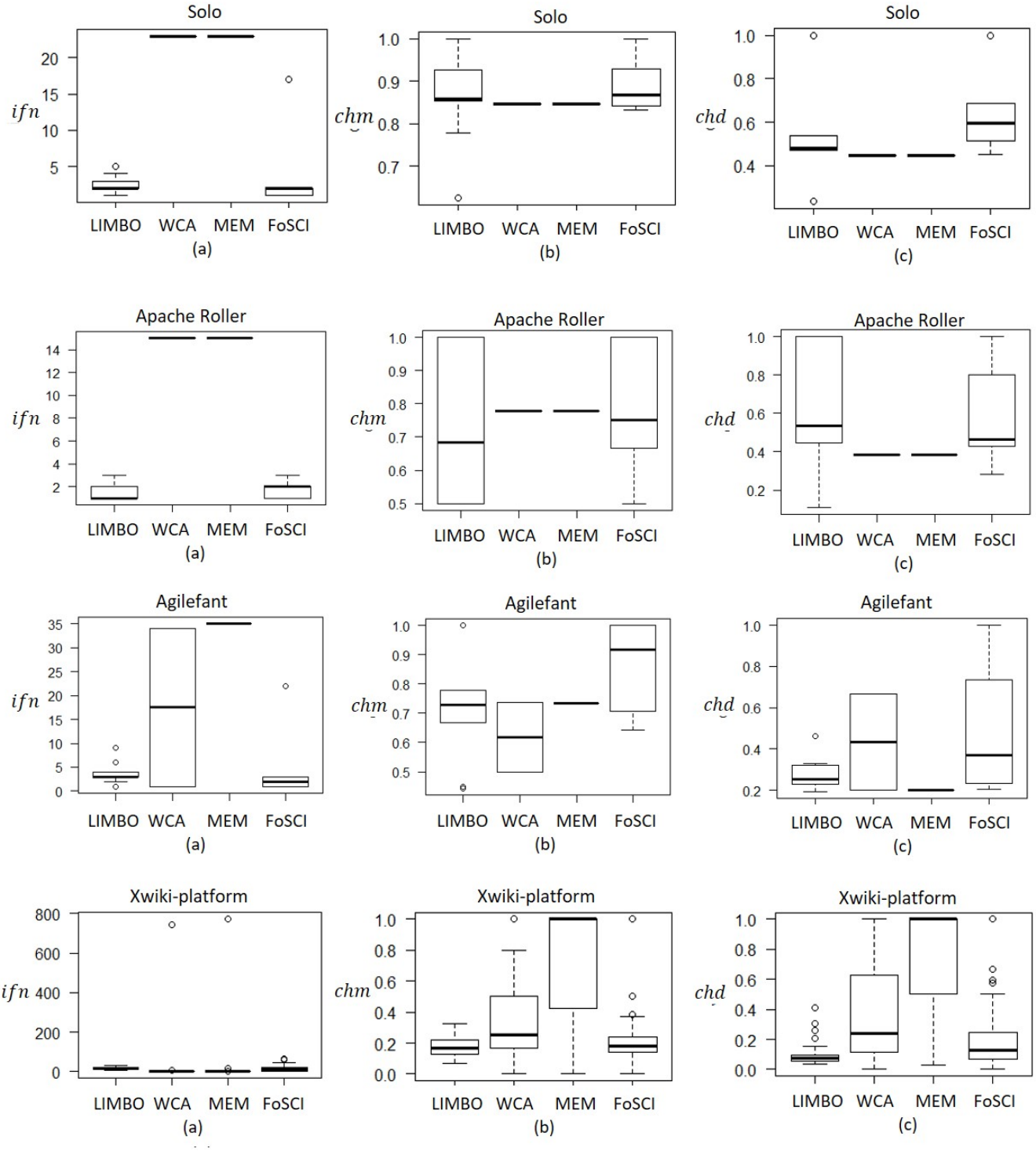


Fig. 5: The distribution of *ifn,chm,chd* measures for service candidates

Figure 5(a). These tiny interfaces have the best $chm = 1$ and $chd = 1$, shown in Figure 5(b) and (c). Consequently, the general results of MEM have exceptionally higher *CHM* and *CHD* than those of FoSCI.

5.2.2 Summary of the Evaluation of Functionality

As indicated by *IFN*(and *ifn*), FoSCI and LIMBO are more capable of splitting the business responsibilities within a monolith into reasonable services candidates, while MEM and WCA tend to heavily mix the functions together into fewer services, and even one very large service. Furthermore, in terms of *CHM*,*CHD* (and *chm*,*chd*), FoSCI performs better than LIMBO. For WCA and MEM, their *CHM* and *CHD* may be better than those of FoSCI when they decompose a monolith into one super larger service (i.e., still a "monolith") and lots of tiny services. This phenomenon will be more obvious when decomposing a large-scale software system (such as the Xwiki-platform).

5.3 The Evaluation of Modularity

Table 8 presents the measures of *SMQ* and *CMQ* for all cases. Concretely, Figure 6(a) illustrates the distribution of $scoh$ and $scop$ of individual service candidates in four cases, same as those in section 5.1. Likewise, the distribution of $ccoh$ and $ccop$ are in Figure 6(b).

5.3.1 Analysis of Results

In the column *SMQ* of Table 8, FoSCI always has the best value among all cases. For *CMQ*, FoSCI outperforms others except for two outliers in gray color. In particular, the *SMQ* and *CMQ* of LIMBO, WCA and MEM even have some measures less than 0, indicating poor modularity. Statistically, all the P-values in the last row indicate statistical significance of the comparison between FoSCI and others.

Figure 6(a) shows the structural cohesion ($scoh$) and coupling ($scop$) for individual service candidates. We observe that service candidates produced by FoSCI exhibit distinctively positive differences from $scoh$ to $scop$. The other methods are unable to achieve high cohesion and low coupling. In terms of conceptual cohesion ($ccoh$) and coupling ($ccop$), the observation is similar, as shown in Figure 6(b). It can bee seen that the observations from Figure 6 and Table 8 are consistent.

5.3.2 Summary of the Evaluation of Modularity

Both structural and conceptual modularity measurements suggest that FoSCI can split a monolith into service candidates with considerably better modularity. Entities inside service candidates by FoSCI tend to function more coherently than those generated by the other three methods.

5.4 The Influence of Coverage on FoSCI

Since FoSCI depends on functional test suites, we investigated how the test coverage level may influence the results. First, through manually inspecting the source code, we categorized the entities not covered by the collected execution traces. After that, we designed experiments to investigate the service candidates generated by FoSCI and how their performance differ under different level of coverage.

5.4.1 Category of Non-covered Entities

To figure out why some classes in our experiments were not covered by execution traces as shown in Table 4, we manually inspected the source code of four[16] investigated projects: Springblog, Solo, JForum, and Apache Roller. We categorize these uncovered classes into 8 types, as shown in Table 9. The examples of these types can be found in our data repository mentioned before.

Based on the categories, Figure 7 illustrates the distribution of classes in the four projects. Taking Springblog as an example, the execution traces collected in experiment cover 72.94% of all classes. Among the uncovered classes, *3rdPartyService* classes account for 9.41% of all classes. *Other* classes account for 2.35%, 10.88%, 11.49% and 3.75% in Springblog, JForum, Solo and Apache Roller. We can see that it is difficult for execution traces to cover all code entities.

5.4.2 The Coverage Influence on FoSCI

It is interesting to observe how FoSCI will perform when the coverage of execution traces changes. First, we introduce two variables, $Coverage_{tr}$ and $Coverage_c$:

$Coverage_{tr}$ denotes *the coverage of execution traces*, the ratio of the number of used execution traces to the entire execution traces collected.

$Coverage_c$ denotes *the coverage of classes*, the ratio of the number of covered classes to the number of classes covered by the entire execution traces.

We use all 6 projects as investigation subjects, and conduct experiments with different $Coverage_{tr}$: 20%, 40%, 60%, 80%, 100%, that is, $5 \times 6 = 30$ experimental groups in total.

For each experimental group, we observe how $Coverage_c$ will change along with the increase of $Coverage_{tr}$. For each group, we repeated 30 times to randomly select the execution traces by a specified $Coverage_{tr}$. Figure 9 presents the statistics labeled with the median of $Coverage_c$. We observe that, individually, the $Coverage_c$ in the lower $Coverage_{tr}$ sometimes may be larger than that in the higher $Coverage_{tr}$. But, statistically, the median of $Coverage_c$ tends to increase with the increase of $Coverage_{tr}$. Therefore, we selected the execution trace set of a specified $Coverage_{tr}$ in experiment if the corresponding $Coverage_c$ is equal to the median labeled.

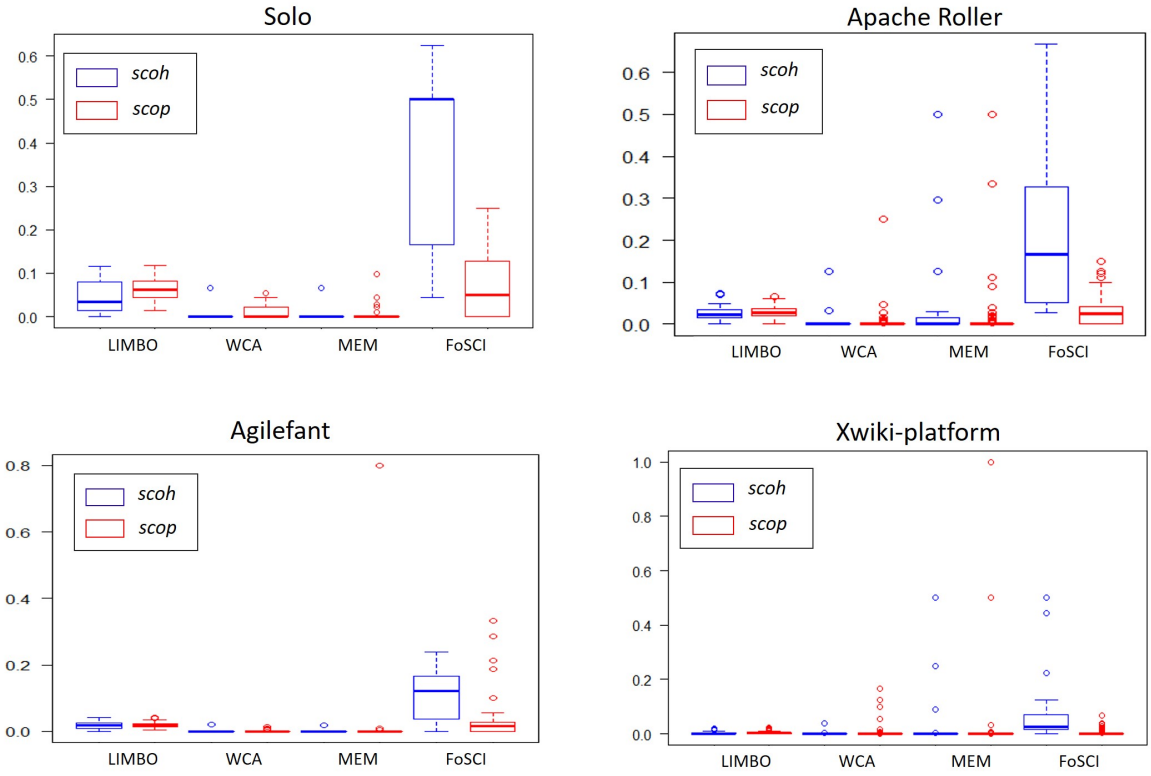
To rigorously conduct the evaluation, we set the number of service candidates being identified (N) for different groups based on $Coverage_c$. As shown in Figure 9, assume $N = k$ in Solo project is set when $Coverage_{tr} = 100\%$. When $Coverage_{tr} = 20\%$, $N = int(58.33\% \times k)$ will be set. k is the same value with the N configured as introduced in section 4.6. Other parameters in FoSCI are the same with those in section 4.3.

How many entities in a service candidate remain the in same service candidate when coverage changes? To answer this question, we define $H_{i,p}$ for service candidate SC_i generated under $Coverage_{tr} = p$:

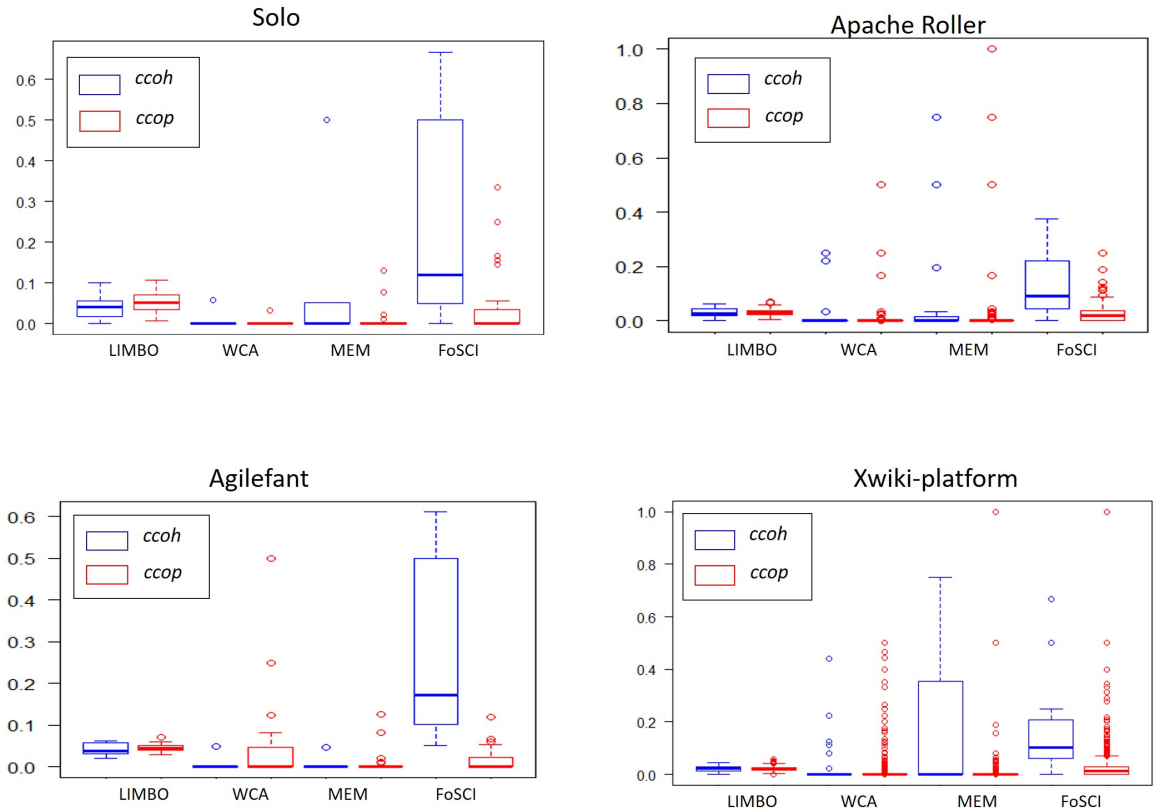
$$H_{i,p} = \frac{MAX}{|SC_i|}$$

where, $|SC_i|$ is the size of service candidate SC_i; MAX is the maximum number of classes ($\forall c \in SC_i$) that remain in the

16. Due to the manual labor, large project, such as Agilefant and Xwiki-platform, were not included.



(a) The distribution of (*scoh, scop*) measures of service candidates



(b) The distribution of (*ccoh, ccop*) measures of service candidates

Fig. 6: The distribution of *(scoh,scop)* and *(ccoh,ccop)* measures for service candidates

TABLE 8: Measurement results of *SMQ* and *CMQ*

Subject	SMQ				CMQ			
	LIMBO	WCA	MEM	FoSCI	LIMBO	WCA	MEM	FoSCI
Springblog	0.0116	−0.0180	0.0928	0.3403	−0.0106	−0.037	0.2071	0.4713
	−0.0028	−0.0130	0.1361	0.3725	−0.0165	−0.0060	0.2207	0.3762
	0.0186	−0.0141	0.1572	0.3713	−0.0045	0.0347	0.2071	0.3368
Solo	−0.0166	−0.0004	−0.0009	0.3007	−0.0110	0.0060	0.0742	0.2494
	−0.0188	−0.0001	−0.0007	0.3033	−0.0145	0.0052	0.1103	0.1863
	−0.0170	0.0001	0.0002	0.2500	−0.0117	0.0049	0.0914	0.2215
JForum	−0.0048	0.0053	0.0359	0.1110	0.0024	−0.0060	0.1713	0.1929
	−0.0024	0.0036	0.0172	0.0756	0.0042	−0.0077	0.1921	0.1334
	0.0005	0.0032	0.0069	0.3872	0.0048	−0.0059	0.1800	0.3199
Agilefant	−0.0035	0.0003	−0.0549	0.1337	−0.0072	−0.0234	0.0018	0.3264
	−0.0021	0.0004	−0.0358	0.1099	−0.0045	−0.0358	−0.0011	0.3348
	−0.0021	0.0004	−0.0250	0.0849	−0.0070	−0.0341	−0.0146	0.3466
Apache Roller	−0.0013	0.0070	0.0557	0.1609	0.0013	0.0355	0.0878	0.0500
	0.0003	0.0069	0.0471	0.1750	0.0003	0.0084	0.0703	0.1045
	−0.0002	0.0064	0.0421	0.1202	−0.0008	0.0011	0.0545	0.3965
Xwiki-platform	0.0001	0.0004	0.0486	0.0489	0.0001	0.0002	0.1124	0.1276
	0.0004	0.0005	0.0468	0.1121	−0.0003	0.0010	0.1024	0.1556
	0.0005	0.0004	0.0471	0.0816	0.0000	0.0083	0.1151	0.1152
P-value	<, ***	<, ***	<, ***	–	<, ***	<, ***	<, ***	–

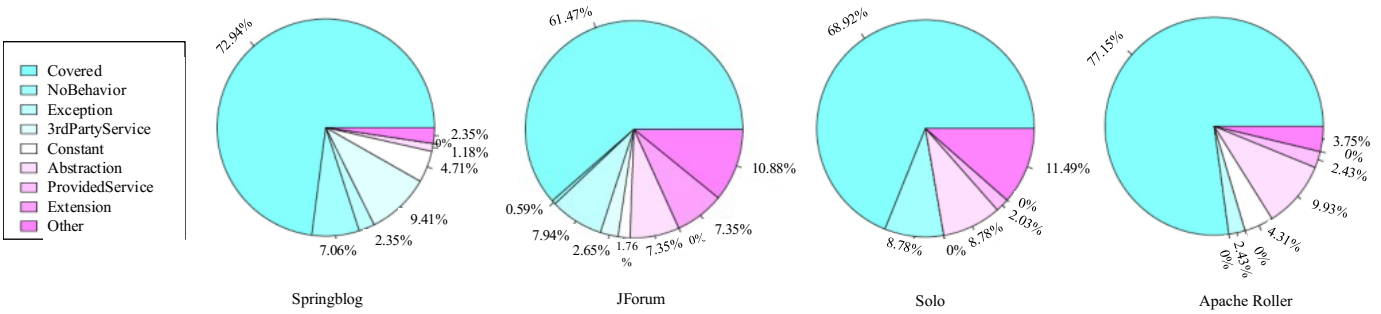


Fig. 7: The distribution of classes in four subjects

TABLE 9: Categories of uncovered classes

Type	Description
NoBehavior	A class only having member variables or constructors, but having no member methods.
Exception	A class responsible for exceptions.
3rdPartyService	A class accessing third party services.
Constant	A class only having static variables which are initialized with constants.
Abstraction	An abstract super-class.
ProvidedService	A class responsible for providing APIs or web-services for external systems.
Extension	A class for extending modules.
Other	A class only used by developers, or other reasons.

same candidate under $Coverage_{tr} = 100\%$. For example, in Figure 8, $H_{1,80\%}$ of service candidate SC_1 is 66.67%.

In Figure 10, the distribution of $H_{i,p}$ indicates that over 50% (the value of $H_{i,p}$) classes of a service candidate remain in the same candidate when $Coverage_{tr}$ differs. In the largest project (Xwiki-platform), $H_{i,p}$ is a little lower with the median equal to 40% when $Coverage_{tr} = 20\%$. In general, the average of $H_{i,p}$ for Springblog, Solo, JForum, Apache Roller, Agilefant, and Xwiki-platform is 71.36%, 64.72%, 59.48%, 55.75%, 74.15%, and 49.21%, respectively.

In terms of the three quality criteria, how does our method perform differently when the coverage differs? For each investigated subject, Table 10 illustrates the *mean* and *standard deviation* (σ) of the sample that is composed of measures under different coverages. We observe that $(45 - 6)/45 = 86.67\%$ of the σ values is "$<= 0.2$".

Summary. These data show that the service candidates and the quality evaluation results are influenced when the coverage differs. However, the results suggest that our method offers acceptable stability, as indicated by $H_{i,p}$ ((71.36% +64.72%+59.48%+55.75%+74.15%+49.21%)/6=62.45%) and σ ($<= 0.2$). In addition, when test coverage varies from 20% to 80% for a subject, its impact on FoSCI is not linear, nor following an obvious pattern. One possible reason is that, when the execution traces differ, other control variables change too, such as the set of classes being covered and being changed. One observation is that, coverage rate has most impact on the largest project (Xwiki-platform). We will further investigate the coverage influence in our future work.

TABLE 10: Measures in different coverage

Subject/Metric	IFN	CHD	CHM	SMQ	CMQ	ICF	ECF	REI
Springblog	**2.399±0.56**	0.459±0.10	0.523±0.11	0.301±0.13	0.336±0.19	**0.531±0.29**	0.140±0.09	0.241±0.15
Solo	**3.441±0.58**	0.665±0.07	0.905±0.02	0.282±0.11	0.358±0.13	0.210±0.04	0.145±0.03	0.715±0.20
JForum	**3.286±0.47**	0.416±0.07	0.681±0.09	0.375±0.06	0.394±0.05	-	-	-
Roller	**2.848±1.56**	0.592±0.09	0.773±0.06	0.219±0.10	0.248±0.10	1.048±0.20	0.843±0.09	0.825±0.13
Agilefant	**4.567±0.50**	0.419±0.05	0.798±0.08	0.147±0.07	0.344±0.05	**1.419±0.62**	0.278±0.13	0.206±0.08
Xwiki-platform	**16.539±0.24**	0.217±0.03	0.235±0.02	0.070±0.02	0.158±0.04	0.168±0.07	0.006±0.00	0.043±0.02

Note: ICF, ECF, REI value for JForum are invalid since JForum's revision data was missing shown in Table 5.

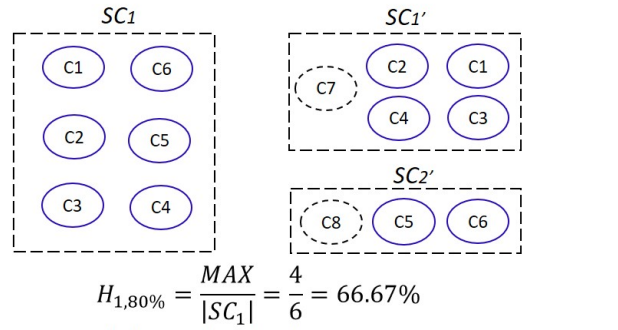


Fig. 8: An example for $H_{i,p}$ definition

5.5 Summary

To sum up, it is evident that our FoSCI can produce efficient service candidates that can consistently exhibit reasonable functionality, modularity and evolution characteristics. Furthermore, FoSCI shows acceptable stability in terms of split results and quality evaluation when the coverage of execution traces differs.

6 Limitations and Threats to Validity

The FoSCI and service candidate evaluation framework proposed in this paper only focus on functionality, modularity, and evolvability, without considering other quality attributes, such as performance, security, or reliability. The FoSCI framework is open to incorporate other quality attribute assessments in the future.

Our method relies on (black-box) execution traces instead of (white-box) source code. This black-box method is appropriate in the following scenarios: a) The source code of the monolithic application is not available. b) Only part of the functionality within a monolith needs to be extracted into service candidates, such as core business capabilities, commonly-used functions, or frequently-updated features [4][35]. In practice, the transition process is usually incremental. c) Both the executable monolith and its sufficient functional test suite are available.

Even though migration from monolithic application to service-based architecture brings benefits such as better maintainability and scalability, the process is complex and costly. Most importantly, microservice architecture is not a silver bullet, nor is the only way to improve maintainability. Not all applications should be designed as microservices. If the objective is to improve the quality of a poorly-designed system, the architect should first consider refactoring rather than migrating to microservices.

In this paper, we selected widely-used web applications as evaluation subjects. Web applications often suffer from maintainability and scalability issues because of their rapid evolution and growth in size and complexity. However, it is hard to guarantee that the evaluation results can be generalized to other types of systems, such as embedded systems. To mitigate this threat, we selected web projects of different sizes, architectural structures, and technology stacks. We plan to conduct experiments using more types of systems in the future, to further address this threat to validity.

Due to the lack of comprehensive and reliable quality evaluation methods to assess and compare service candidates [36][37], we constructed a systematic measurement suite to quantitatively and consistently assess service candidates against three quality criteria (i.e., functionality, modularity, evolvability) using 8 metrics. These metrics are derived from the service interface information, structural/-conceptual dependency, and revision history. The evaluation indicated that our method outperforms other three baseline methods with respect to these metrics, but it still deserves further evaluation if more reliable metrics become available.

7 Related Work

7.1 Software Decomposition

Decomposition methods. The goal of software decomposition is to split a large system into small, manageable modules or components (e.g. Parnas et al. [38], Bavota et al. [39]). WCA and LIMBO are classic methods that take static structural relations extracted from source code as input [12][13][14]. They both employ hierarchical clustering but use different distance measurements. Other studies employed information retrieval techniques [15][16]. These techniques considered source files as their text corpora, ignoring the structure of their programming languages. Using the natural language approach, each source code file is represented as a vector of keywords or a distribution of topics, extracted from source code and comments. Lutellier et al. [40] and Garcia et al. [13] performed a comparative analysis of six clustering techniques.

Search-based approaches also have been explored. Mancoridis et al. treated module clustering as a single-objective optimization problem using modular quality as the optimization objective [7]. Praditwong et al. defined software modularization as a multi-objective optimization problem [41]. The objectives included cohesion, coupling and others.

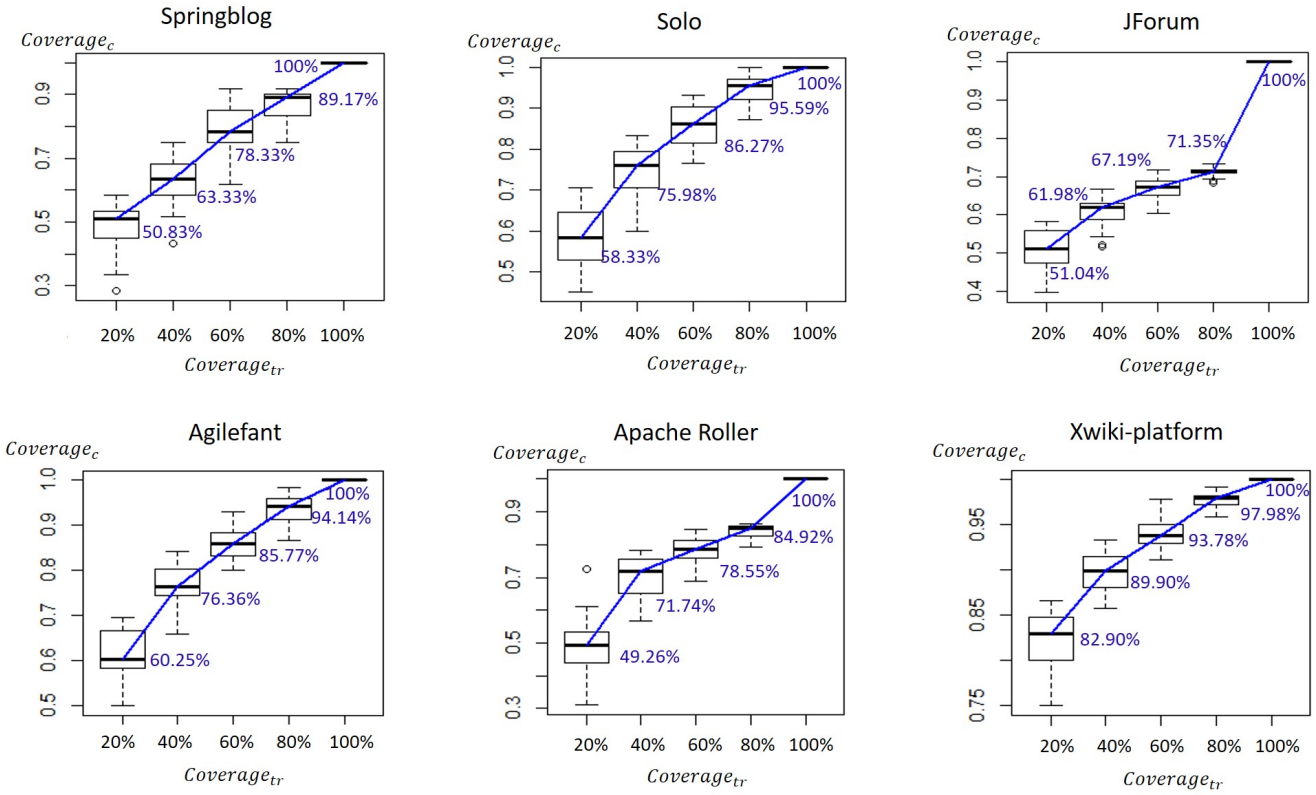


Fig. 9: The statistic of $Coverage_c$ under different $Coverage_{tr}$

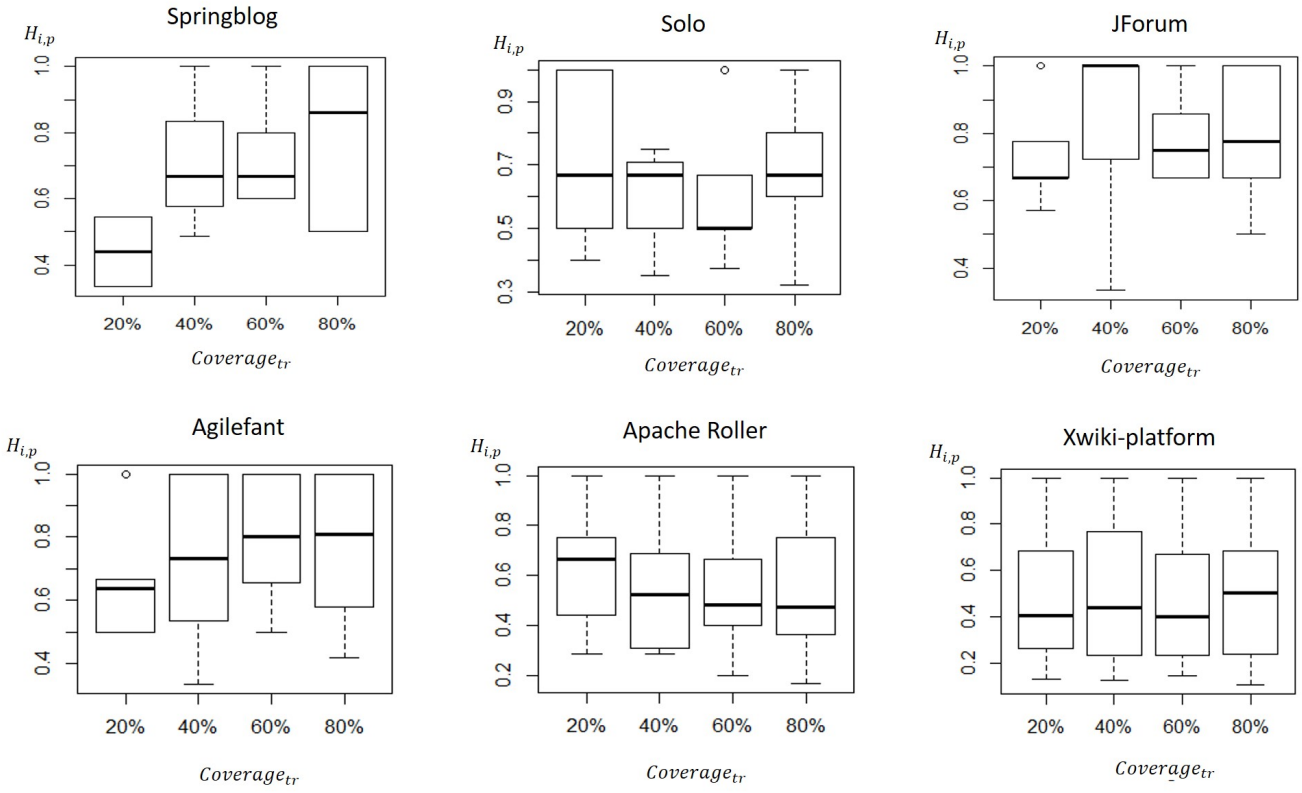


Fig. 10: $H_{i,p}$ distribution under different $Coverage_{tr}$

These existing methods rarely consider software functionality. Most of them are motivated by the assumption that developers pursue modules with high cohesion and low coupling [9][10][11]. However, Candela et al. [11] pointed out that factors other than cohesion and coupling might need to be taken into account. In contrast, our method is functionality-oriented, generates functional atoms before further clustering, and models multiple optimization objectives based on execution traces.

Evaluation of methods. Bavota et al. [16] and Praditwong et al. [41] evaluated their methods by measuring the improvement of cohesion and the reduction of coupling. Garcia et al. [13] and Lutellier et al. [40] proposed a suite of metrics to assess the accuracy of architecture recovery by comparing with benchmark architectures.

As mentioned above, in addition to cohesion and coupling, other quality criteria for evaluation should be considered. In addition, oracle architectures normally do not exist. By contrast, our work employs measures extracted from service candidate interfaces and revision history, which is novel and more objective.

7.2 Service Candidate Identification

Identification methods. Service candidate identification is a form of software decomposition in the realm of service-based architecture, a counterpart of traditional software decomposition. We classify existing methods into two categories: data-oriented and structure-oriented.

Data-oriented methods begin with splitting a data source or database. Levcovitz et al. [6] partitioned database tables into several groups. Then they gathered classes, which access the same group of tables, to compose a service candidate. Chen et al. [42] presented a data-flow based method. The data flow diagrams of business logic need to be provided by the users. Structure-oriented methods employ structural relations from source code. Gysel et al. [43] and Mazlami et al. [5] designed graph-cutting algorithms to generate service candidates.

In general, data-oriented methods start with database partitions or the analysis of data flow graphs of business logic, which cannot be automated. Structure-based methods can relieve manual operation, but these may come at the expense of ignoring high-level business functionality.

Evaluation of methods. Since the research in service extraction especially microservice extraction is still in its infancy, only a few works have conducted evaluations to validate their methods. Mazlami et al. [5] concluded that their methods could produce microservices with the benefits of team size reduction and less domain redundancy. They have not compared their proposed method with others yet. Chen et al. [42] conducted experiments on two use cases.

None of the prior work conducted comparative and comprehensive evaluations. In this paper, we have conducted experiments using 6 widely-used open-source projects, and extracted execution traces from 3,349,253,852 records in execution logs. More importantly, our method has been evaluated against 8 metrics, assessing *Independence of Functionality*, *Modularity* and *Independence of Evolvability* respectively.

7.3 Cloud Service Extraction

Another branch of related work is cloud service extraction. It aims to transitioning an application to use cloud-based services, taking the advantage of cloud resources. Kwon et al. [44] described two mechanisms to recommend which class should be transformed into cloud-based services by taking into factors such as application performance penalty and business functionality. Moreover, they developed and implemented a set of refactoring techniques and reduced the manual efforts involved in the code transformation. Gholami et al. [45] surveyed the research of migrating legacy applications to the cloud, and discussed relevant issues, approaches and concerns. Tilevich et al. [46] addressed the problem of cloud offloading, i.e., executing the energy-intensive portion of a mobile application in a remote cloud server. The work of Zhang Y. et al. [47] is similar, with the purpose of mobile performance improvement or energy optimization.

7.4 Dynamic Analysis Based on Execution Traces

Dynamic analysis based on execution traces has a rich history in program comprehension [48]. It has been recognized that execution data can not only accurately expose actual software behavior [49], but can also reveal the specific functionality of programs [17] [18].

Many works have utilized execution traces for feature localization [48]. Rohatgi et al. [50] combined static analysis and dynamic analysis for feature location. Safyallah et al. [51] analyzed patterns from execution traces. Li et al. [17] relied on analysis of executions of test cases to accurately recognize entities that contribute to a function.

Some research employs execution traces to help form a high-level view of a program. Hamou-Lhadj et al. [52] summarized the content of execution traces, generating UML sequence diagrams. Alimadadi et al. [18] inferred hierarchical motifs from execution traces, which can be used to discover specific functions of the program. Based on the same rationale, our work leverages execution traces to extract service candidates.

8 Conclusion and Future Work

In this paper, we proposed the FoSCI framework for service candidate identification from monolithic software systems. Moreover, we created a comprehensive measurement system to evaluate service candidates. Compared with three baseline methods, our evaluation results indicate that FoSCI can identify service candidates with better *Independence of Functionality, Modularity*, and *Independence of Evolvability*. Identifying service boundaries from monolithic software is a complicated task. Our research will focus on two areas to improve our method in the future:

Guided and interactive service candidate identification. In practice, service identification from monoliths is an iterative and incremental process. To better assist architects or developers, it is important to consider expert knowledge during the design of services. We intend to improve our methods by allowing the user to integrate feedback and guidance.

Further refining service candidates to create executable services. Service candidates are just intermediate products that

have the potential to be further implemented as services. The transition from service candidates to deployable services is another goal of our future work.

REFERENCES

[1] W. Jin, T. Liu, Q. Zheng, D. Cui, and C. Yuanfang, "Functionality-oriented microservice extraction based on execution trace clustering," in *Web Services (ICWS), 2018 IEEE International Conference on*. IEEE, 2018, pp. –.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[3] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[4] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, 2018.

[5] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 524–531.

[6] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *CoRR*, vol. abs/1605.03175, 2016. [Online]. Available: http://arxiv.org/abs/1605.03175

[7] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 50–59.

[8] V. Tzerpos and R. C. Holt, "Accd: an algorithm for comprehension-driven clustering," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE, 2000, pp. 258–267.

[9] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[10] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 469–478.

[11] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 24, 2016.

[12] M. Chatterjee, S. K. Das, and D. Turgut, "Wca: A weighted clustering algorithm for mobile ad hoc networks," *Cluster computing*, vol. 5, no. 2, pp. 193–204, 2002.

[13] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2013, pp. 486–496.

[14] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 27–36.

[15] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 552–555.

[16] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, p. 4, 2014.

[17] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, 2018.

[18] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Inferring hierarchical motifs from execution traces," 2018.

[19] P. B. Kruchten, "The 4+ 1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

[20] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[21] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service-and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. ACM, 2017, pp. 107–115.

[22] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. IEEE, 1998, pp. 45–52.

[23] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software: Evolution and Process*, vol. 19, no. 2, pp. 77–131, 2007.

[24] H. P. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.

[25] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012, pp. 247–248.

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[27] J. Branke, K. Deb, H. Dierolf, and M. Osswald, "Finding knees in multi-objective optimization," in *International conference on parallel problem solving from nature*. Springer, 2004, pp. 722–731.

[28] S. Allier, S. Sadou, H. Sahraoui, and R. Fleurquin, "From object-oriented applications to component-oriented applications via component-oriented architecture," in *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*. IEEE, 2011, pp. 214–223.

[29] A. Shatnawi, H. Shatnawi, M. A. Saied, Z. Alshara, H. A. Sahraoui, and A. Seriai, "Identifying components from object-oriented apis based on dynamic analysis," in *Program Comprehension (ICPC), 2018 IEEE International Conference on*. IEEE, 2018, pp. –.

[30] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2015.

[31] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[32] S. Millett, *Patterns, Principles and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[33] H. Harms, C. Rogowski, and L. Lo Iacono, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 902–907.

[34] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[35] M. Stine, "Migrating to cloud-native application architectures," 2015.

[36] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 44–51.

[37] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 21–30.

[38] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[39] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[40] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, 2017.

[41] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[42] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 2017, pp. 466–475.
[43] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2016, pp. 185–200.
[44] Y.-W. Kwon and E. Tilevich, "Cloud refactoring: automated transitioning to cloud-based services," *Automated Software Engineering*, vol. 21, no. 3, pp. 345–372, 2014.
[45] M. F. Gholami, F. Daneshgar, G. Low, and G. Beydoun, "Cloud migration processa survey, evaluation framework, and open challenges," *Journal of Systems and Software*, vol. 120, pp. 31–69, 2016.
[46] E. Tilevich and Y.-W. Kwon, "Cloud-based execution to improve mobile application energy efficiency," *Computer*, vol. 47, no. 1, pp. 75–77, 2014.
[47] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *Acm Sigplan Notices*, vol. 47, no. 10. ACM, 2012, pp. 233–248.
[48] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software:Evolution and Process*, vol. 25, pp. 53–95, 2013.
[49] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.
[50] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 236–241.
[51] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 84–88.
[52] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 181–190.