2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Datalog-Based Language-Agnostic Change Impact Analysis for Microservices

Qingkai Shi[1], Xiaoheng Xie[2], Xianjin Fu[2], Peng Di[2], Huawei Li[3], Ang Zhou[2], Gang Fan[2]
[1]*The State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[2]*Ant Group, Hangzhou, China* [3]*Alibaba Inc., Hangzhou, China*
qingkaishi@nju.edu.cn, {xiexie, fuxianjin.fxj, dipeng.dp, zhouang.za, fangang}@antgroup.com, huawei.lihw@alibaba-inc.com

***Abstract*—The shift-left principle in the industry requires us to test a software application as early as possible. In particular, when code changes in a microservice application are committed to the code repository, we have to efficiently identify all public microservice interfaces affected by the changes so that the impacted interfaces can be tested as soon as possible. However, developing an efficient change impact analysis is extremely challenging in microservices due to the multilingual problem: microservice applications are often implemented using varying programming languages and involve diverse frameworks and configuration files. To address this issue, this paper presents MICROSCOPE, a language-agnostic change impact analysis that uniformly represents code, configuration files, frameworks, and code changes by relational Datalog rules. MICROSCOPE then benefits from an efficient Datalog solver to identify impacted interfaces. Experiments based on the use of MICROSCOPE in Ant Group, a leading software vendor, demonstrate that MICROSCOPE is both effective and fast, as it successfully identifies interfaces affected by 112 code commits, with moderate time overhead, and could reduce 97% of interfaces to test and save 73% of testing time after code changes.**

***Index Terms*—change impact analysis, Datalog-based analysis, microservices**

I. INTRODUCTION

The microservice architecture has experienced a steady increase in popularity in recent years. In a real-world production environment, multiple microservices interact as a cohesive system, each responsible for a specific and well-defined functionality. This development model enables software developers to focus on realizing the requirements of individual microservices independently from each other. However, the increased complexity of the interaction among microservices has concurrently escalated the potential for the introduction of bugs. The inherent dynamism of microservices-based systems, which are continuously modified and deployed to adapt to evolving business requirements or to address bugs, creates an environment prone to significant disruptions. Recent reports indicate that changes are a major cause of incidents, with approximately 50% at China Guangfa Bank [1], 54% at Baidu [2], and 70% at Google [3] being change-related. Further evidence from 2022 indicates that 55% of online incidents were due to code changes [4], underscoring the impact of system modifications.

This work was partially supported by Ant Group. Peng Di and Gang Fan are the corresponding authors.

To deal with the dynamism of microservices and avoid bugs introduced by code changes, change impact analysis (CIA) [5], [6] is widely used throughout the software industry to assess the potential impact of code changes. However, a few significant challenges affect the deployment of CIA for microservice applications. On the one hand, the first major challenge is the multilingual problem (**Challenge 1**). Specifically, microservices are often implemented in diverse programming and configuration languages, and on top of varying programming frameworks, which are hard to describe uniformly. While there have been a few existing works on CIA, e.g., [7]–[9], they often work for a single programming language or framework, thus falling short of addressing the multilingual problem.

On the other hand, an additional complication is presented by the shift-left principle in the industry, which induces performance-related issues (**Challenge 2**). That is, the shift-left principle in the industry requires us to test a software application as early as possible. As such, we have to identify the impact of code changes early so that the impacted software components can be tested as soon as the changes are made. The substantial program size of microservices in the industry poses significant challenges for the CIA, complicating the task of conducting analyses within a reasonable time frame while upholding the shift-left strategy.

This paper presents a Datalog-based CIA for microservice applications in response to these challenges. Our key insight is that code elements, such as classes, interfaces, functions, expressions, and their relationships, can be uniformly represented via relational data models, regardless of what specific programming or configuration languages are used. As such, our approach, agnostic to specific languages via a layer of relational data abstraction, is devised based on Datalog techniques to adeptly maneuver the complexities of identifying change-impacted microservice interfaces. Particularly, our approach has two promising features. First, we suggest a uniform Datalog-based representation that standardizes how different languages or frameworks are interpreted, thereby mitigating Challenge 1. Second, the Datalog-based representation allows us to benefit from modern Datalog solvers to deal with industrial-sized microservices, exhibiting a high efficiency to identify change impacts and alleviating Challenge 2.

Our approach to CIA has been implemented as a tool named MICROSCOPE, which currently supports microservices written in **nine** programming or configuration languages, including Java, XML, JavaScript/TypeScript, Go, C/C++, Python, Swift, SQL, and Properties. This tool automatically parses microservices' source code and configuration files into uniform Datalog representations. When a developer commits code changes into the code repository, MICROSCOPE automatically identifies the changes and translates the changes into Datalog representations, too. MICROSCOPE then invokes a Datalog solver, namely Souffle [10], to identify the interfaces impacted by code changes. We evaluated MICROSCOPE using real-world code changes from both the open-source community and Ant Group, a leading software vendor. For each commit, MICROSCOPE can identify the impacted interfaces using only 1/4 of the time for building (or compiling) a microservice project. Among 112 examined commits, MICROSCOPE confirmed the code changes influenced 338 (3%) out of 11,458 interfaces used in upstream or downstream applications, with manageable false positive and negative rates (17% and 6%). In contrast, we may miss $4\times$ impacted interfaces even if we remove the support for only one language. These results underscore MICROSCOPE's effectiveness and the importance of addressing the multi-lingual problem in real-world scenarios.

In conclusion, this paper makes the following contributions:

- We introduce a Datalog-based CIA approach that is language-agnostic and adaptable to the evolving needs of different projects.
- We implement the Datalog-based approach as a tool, namely MICROSCOPE, which supports nine languages and has been made publicly available.[1]
- We extensively evaluate our approach in a real and industry setting, demonstrating the efficiency and effectiveness of the tool, MICROSCOPE.

II. MOTIVATION AND OVERVIEW

This section presents the background of microservices, motivates the problem of identifying change impacts, and provides an overview of our approach with an example simplified from a real and severe change-related vulnerability.

A. Necessity of CIA in Industry

Recent reports indicate that code changes are a major cause of incidents, with approximately 50% at China Guangfa Bank [1], 54% at Baidu [2], and 70% at Google [3] being change-related, causing security issues such as Broken Object Level Authorization and Excessive Data Exposure, two OWASP Top 10 API security issues [11]. These statistics underscore the importance of CIA in industrial settings, but we often encounter difficulties managing large, complex systems that constantly change. The difficulties are even amplified within microservices, which feature a high degree of inter-service communication and dependency.

[1]https://github.com/codefuse-ai/CodeFuse-Query.

To ensure software quality after code changes, running the entire test suite after every code change is inefficient, let alone when the changes are frequent. CIA allows us to determine the interfaces impacted by code changes, allowing software engineers to selectively execute tests covering these interfaces. Undoubtedly, such a selective test execution will save considerable time and resources. According to our evaluation, as change-related interfaces only account for 3% of the total interfaces, we can save 97% of testing efforts in terms of reducing the number of interfaces to test and save 73% of testing time on average every time code changes happen.

B. Motivating Example

Microservice is a software architecture that decomposes an application into small and self-contained services. Each microservice is developed, deployed, and maintained independently. In practice, microservices are typically deployed in separate containers, with communication occurring through lightweight mechanisms such as remote procedural calls. Such communications can be viewed as a series of invocations between the caller and callee services, where the caller service relies on the functionality offered by the callee to achieve certain objectives. In this paper, we formulate microservices and service interfaces as follows.

Definition 1: (Microservice) A microservice is denoted by (V, E), where V is a set of functions in the service, and E contains the caller-callee relationship over V.

Definition 2: (Service Interface) An interface of a microservice (V, E) is a function $f \in V$ published to and invoked by clients or other microservices.

Given the distributed and interconnected nature of microservice applications, changes within one service can ripple across the entire system, causing significant consequences as evidenced by a recent report [4]. For example, the one-line code change discussed below leads to the inconsistency between the front end (client) and the back end (server) of Alipay, a mobile app from Ant Group. The inconsistency has caused Alipay to crash and affect over 20 million users.

Basically, this microservice application manages the information of clients and bills from an e-commerce website. Figure 1 shows a few source code files, including four programming or configuration languages, i.e., Java, XML, Javascript, and Typescript. The green and boxed line in Figure 1(a) indicates the one-line code change that adds a phone-number field into the Java class `ClientInfo`.

The code snippets can be segregated into two sections: the back and the front ends. The back end comprises `ClientInfo`, `BillInfo`, `BillFacade`, and `service.xml`. They manage server-side operations, including client data management, bill data management, and system configuration. On the other side, the front end, i.e., the client app, includes `proxy.js` and `genericOrder.ts`. The former facilitates the communication between the front-end and the back-end components. The latter implements client-side business logic. Let us discuss a bit more about what the six files do.

```java
1. public class ClientInfo {
2.     private Integer id;
3.     private String name;
4. +   private String phoneNumber;
5. }
```

(a) ClientInfo.java

```java
6. public class BillInfo {
7.       private ClientInfo clientInfo;
8.       private BillData billData;
9.       ...
10.}
```

(b) BillInfo.java

```java
11. interface BillFacade {
12.    BillInfo load(BillRequest request);
13. }
```

(c) BillFacade.java

```xml
14. <sofa:service
15.    interface="com.store.BillFacade">
16. </sofa:service>
```

(d) service.xml

```javascript
17. { appname:'OrderCenter',
18.    api: {
19.      billFacade:'com.store.BillFacade'
20.    }
21.  }
```

(e) proxy.js

```typescript
22. function queryOrder(orderId:any,
23.                    context:any): any {
24.  return billFacade.load(
25.      {context.userId, orderId}));
26. }
```

(f) order.ts

Fig. 1: Motivating example.

- The Java classes, `ClientInfo` and `BillInfo` in Figures 1(a) and 1(b), encapsulate a customer's profile data (id, name, and phone number) and the billing details.
- The Java interface, `BillFacade` in Figure 1(c), is a public interface that allows clients to retrieve `BillInfo` via a method, namely `load`.
- The configuration file, `service.xml` in Figure 1(d), publishes the interface `com.store.BillFacade`, i.e., the Java interface in (c), to clients such that a client can invoke its methods via a remote procedure call.
- The Javascript, `proxy.js` in Figure 1(e), is a client-side configuration file. Line 19 declares a remote object named `billFacade`, which is of the type `com.store.BillFacade` and can be used in the application named `OrderCenter`.
- The Typescript, `order.ts` in Figure 1(f), implements a front-end function in the application `OrderCenter`. Line 24 invokes a remote method `load` via the remote object `billFacade`.

When upgrading the microservice, developers implement the one-line change in Figure 1(a), which restructures the client information. However, developers fail to recognize that this change will affect the front end, i.e., the function invocation in Figure 1(f), which queries a client's order information but does not provide the phone number as a query condition. Consequently, this inconsistency crashes the microservice app and affects 20 million users.

It is challenging to identify such inconsistencies in microservices manually. Dependencies among functions are often implicitly encoded, e.g., via configuration files, and involve multiple languages. Overlooking any single dependency may cause significant issues, such as in the case of the motivating example. Unfortunately, few existing techniques can automatically analyze such implicit and cross-language change impacts, which is the problem this paper aims to address. In short, we state the problem to address as below.

> Given a code change in microservices, identify the service interfaces (see Definition 2) impacted by the change.

C. *Microscope in a Nutshell*

Our key idea is to provide a language-agnostic layer over programs in different languages, such that we have a uniform representation to ease program analysis. As shown in Figure 2, the first step is to build such a layer using a Datalog-style representation, which we refer to as the code facts and the change facts. We then define Datalog rules to establish relationships among these facts. Finally, a Datalog engine infers service interfaces impacted by changes.

1) Notations: We use a predicate in the form of predicate(x, y, . . .), where black letters in the Times New Roman font, x, and y, are formal parameters, to describe a dataset satisfying the relationship named predicate. Datalog facts are instantiated predicates, e.g., predicate(`a`, `b`, . . .) where red letters in the typewriter font, `a` and `b`, are string or numeric constants, representing a concrete entry in the dataset. For instance, we can use parent(x, y) and grandparent(x, y) to describe the parent-child and grandparent-grandchild relationships between x and y, respectively. The fact, parent(`Alice`, `Bob`), says that `Alice` is a parent of `Bob`.

Datalog rules state how we deduce new facts from known facts. For example, a Datalog rule in the form of grandparent(x, y) :- parent(x, z) parent(z, y) means that if x is a parent of z and z is a parent of y, then x is a grandparent of y. Given all parent-child facts, this rule returns all (x, y) satisfying the grandparent-grandchild relationship. For instance, if we have two facts, parent(`Alice`, `Bob`) and parent(`Bob`, `Charlie`), this rule deduces a new fact grandparent(`Alice`, `Charlie`) to describe their grandparent-grandchild relationship.

2) Code Facts: As the first step, MICROSCOPE translates all source files in Figure 1 into Datalog facts. The code in Figure 1(a) is translated into three facts:

1) field(`Integer`, `id`, `ClientInfo`, `(a):2:2`);
2) field(`String`, `name`, `ClientInfo`, `(a):3:3`);
3) field(`String`, `phoneNumber`, `ClientInfo`, `(a):4:4`).

Each of Facts 1-3 describes a class field, including its type and name, as well as the class where the field is declared. The last argument of these facts is the location where a field is declared, including the source file, the start line number, and the end line number. Similarly, we can translate the Java code in Figures 1(b), 1(c), and 1(d) into the following facts. Facts 4-5 are similar to Facts 1-3. Fact 6 describes a function named `load` in the class `BillFacade`. Its return type is `BillInfo`. We omit the function parameters to ease the explanation. Fact 7 is extracted from the XML file, which states the elements and attributes in the XML file:

4) field(`ClientInfo`, `clientInfo`, `BillInfo`, `(b):7:7`);
5) field(`BillData`, `billData`, `BillInfo`, `(b):8:8`);
6) function(`BillInfo`, `load`, `BillFacade`, `(c):12:12`);
7) xml_attr(`interface`, `BillFacade`, `sofa:service`, `(d):15:15`).

While Facts 1-7 are extracted from the back-end code, the following are from the front end, i.e., (e) and (f) in Figure 1. Fact 8 is from Figure 1(e), describing the key-value pairs in the JS file, i.e., the remote object `billFacade`

TABLE I: Relationship rules for the motivating example.

imp_field(fname, cname)	:-	change(loc) field(_, fname, cname, floc) in(floc, loc)
imp_class(cname)	:-	imp_field(fname, cname)
imp_class(cname)	:-	imp_class(fclass) field(fclass, fname, cname, _)
imp_function(fname, cname)	:-	imp_class(fclass, _) function(fclass, fname, cname, _)
pub_function(fname, cname)	:-	xml_attr(`interface`, cname, `sofa:service`, _) function(_, fname, cname, _)
ref_function(callee, refclass, loc)	:-	call(ref, callee, caller, loc) function(_, caller, app, _) api(ref, refclass, app, _)
cia_result(callee, refclass, loc)	:-	imp_function(callee, refclass) pub_function(callee, refclass) ref_function(callee, refclass, loc)

and its class `BillFacade` that can be used in the front-end app `OrderCenter`. Facts 9-10 are extracted from (f). Fact 9 says that there is a function named `queryOrder` returning a `BillInfo` in the app `OrderCenter`. Fact 10 describes a call expression invoked by the remote object `billFacade`.

8) api(`billFacade`, `BillFacade`, `OrderCenter`, `(e):18:19`);
9) function(`BillInfo`, `queryOrder`, `OrderCenter`, `(f):22:26`);
10) call(`billFacade`, `load`, `queryOrder`, `(f):24:25`).

3) Change Facts: The fact of code changes is straightforward, describing which line in a file is changed, i.e., change(`(a):4:4`) — line 4 in Figure 1(a) is changed. The next section will discuss more details about change facts. Note that we focus on code changes in both source code and configuration files. Changes not in the code are out of the scope of this paper.

4) Relationship Rules: We establish the Datalog rules in Table I to build the relationships among these facts. The rules are general for different microservices and are not changed once defined. They can be divided into four parts: identifying functions impacted by code changes, identifying functions published to clients, identifying functions referenced by clients, and finding the final solution of CIA.

(i) Identifying Impacted Functions. The first four rules in Table I find the fields, classes, and functions impacted by the code change. The first rule identifies a field impacted by code changes. It says that, given a code change like change(`(a):4:4`), if a field is in the same location, the code change impacts the field. In the rules, the underscore symbol is an argument that a rule does not care about, and in(loc_1, loc_2) is an auxiliary predicate that checks if two locations overlap.

The 2nd rule states that a class is impacted by code changes if the class contains an impacted field. The 3rd rule recursively finds classes impacted by code changes. For example, since the class `BillInfo` in Figure 1(b) contains a field of class `ClientInfo`, which contains a change-impacted field, the class `BillInfo` is also impacted by the changes. The 4th rule identifies change-impacted functions, which use an impacted class as the return type. The predicate imp_function(fname, cname) includes the function name, i.e., fname, and the class where the function is declared, i.e., cname.

(ii) Identifying Published Functions. The motivating example uses SOFAStack [12] as the microservice framework, which publishes a function to clients via an XML file like Figure 1(d). The fifth rule in Table I defines the published functions. It states that a published function is in a Java class declared in the XML attribute `interface` of the XML element named `sofa:service`. Due to Fact 7, this rule yields that all functions in the class `BillFacade` are published and can be invoked by clients. Since Fact 6 indicates that the function `load` is in the class `BillFacade`, it is a published function.

(iii) Identifying Referenced Functions. As discussed, Figure 1(e) defines a remote object, `billFacade` of class `BillFacade`. The client app `OrderCenter` can use it to invoke a remote function. Figure 1(f) is a function in the client app that uses the remote object, `billFacade`, and invokes the remote function named `load`. As such, we identify functions referenced by the client app via the sixth rule. The left-hand side of this rule returns a function (i.e., callee), the class where it is declared (i.e., refclass), and the location where the function is invoked (i.e., loc). The right-hand side states that such a function should be invoked by a remote object declared in Figure 1(e).

(iv) Putting the Facts and Rules Together. Finally, we use the final rule in Table I to get a set of functions that are impacted by code changes, published to the external world, and, meanwhile, used by clients. The result includes the function name, the class where the function is declared, and the location where the function is invoked.

III. Approach

MICROSCOPE supports nine programming or configuration languages to address the multi-lingual problem in microservices. However, we cannot discuss all of them due to the page limits. Instead, we discuss the design using a widely used microservice framework, SOFAStack, and assume both the back-end (server) and the front-end (client) of a microservice application are written in Java and XML. In this section, after introducing the basic concepts in SOFAStack (§III-A), we discuss the technical design following the workflow in Figure 2, including the extraction of Datalog facts (§III-B) as well as the definition of Datalog rules for change impact analysis (§III-C).

A. SOFAStack for Microservices

SOFAStack [12] extends Spring Boot [13] and provides a flexible framework for developing microservices in different languages, such as in the motivating example discussed. On the server side, an XML file declares implicit dependencies

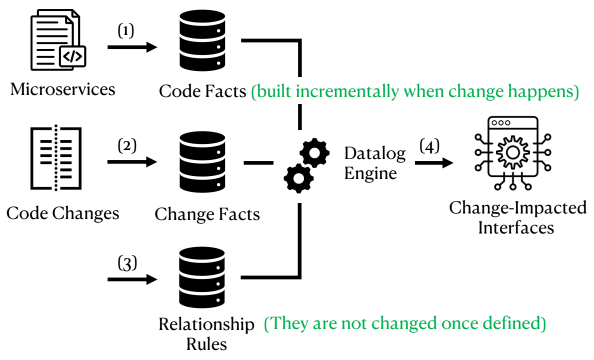
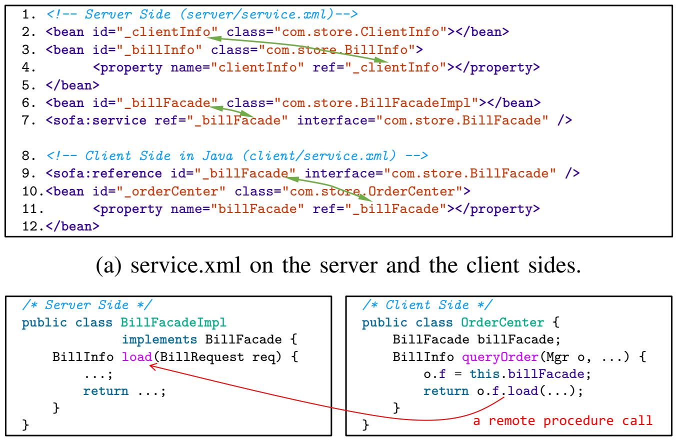

Microservices
(1)
Code Facts (built incrementally when change happens)
Code Changes
(2)
Change Facts
Datalog Engine
(4)
Change-Impacted Interfaces
(3)
Relationship Rules (They are not changed once defined)

Fig. 2: Workflow of MICROSCOPE.



```
1. <!-- Server Side (server/service.xml)-->
2. <bean id="_clientInfo" class="com.store.ClientInfo"></bean>
3. <bean id="_billInfo" class="com.store.BillInfo">
4.        <property name="clientInfo" ref="_clientInfo"></property>
5. </bean>
6. <bean id="_billFacade" class="com.store.BillFacadeImpl"></bean>
7. <sofa:service ref="_billFacade" interface="com.store.BillFacade" />

8. <!-- Client Side in Java (client/service.xml) -->
9. <sofa:reference id="_billFacade" interface="com.store.BillFacade" />
10.<bean id="_orderCenter" class="com.store.OrderCenter">
11.       <property name="billFacade" ref="_billFacade"></property>
12.</bean>
```

(a) service.xml on the server and the client sides.

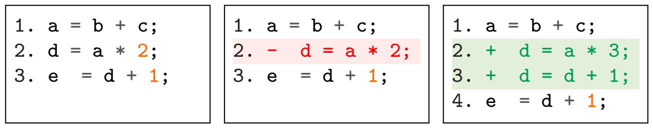
```
/* Server Side */
public class BillFacadeImpl
             implements BillFacade {
    BillInfo load(BillRequest req) {
        ...;
        return ...;
    }
}
```

```
/* Client Side */
public class OrderCenter {
    BillFacade billFacade;
    BillInfo queryOrder(Mgr o, ...) {
        o.f = this.billFacade;
        return o.f.load(...);
    }
}
```

a remote procedure call

(b) Additional Java source code.

Fig. 3: Illustration of SOFAStack.



```
1. a = b + c;
2. d = a * 2;
3. e  = d + 1;
```

(a) Original code

```
1. a = b + c;
2. -  d = a * 2;
3. e  = d + 1;
```

(b) Code deletion

```
1. a = b + c;
2. +  d = a * 3;
3. +  d = d + 1;
4. e  = d + 1;
```

(c) Code addition

Fig. 4: Dealing with code changes.

in the Java code and publishes functions that a remote client can invoke. As shown in Figure 3(a), the server-side XML file often includes a list of beans and SOFA services to publish. For each bean, the underlying SOFA framework creates a Java object of the specified Java class at runtime. We can also declare dependencies among these beans. For instance, the class `BillInfo` has a field named `clientInfo`. Line 4 in the XML file states that this field is initialized with the object whose bean ID is `_clientInfo`. That is, in Line 4, the attribute `name` specifies the field name; the attribute `ref` specifies the ID of the bean to initialize the field.

Line 6 declares a bean of Java class `BillFacadeImpl`, which, as shown in Figure 3(b), implements the Java interface `BillFacade`. The Java class implements the function `load`. Line 7 in the XML file declares that all functions in the Java interface `BillFacade` are published to the clients. The object of Java class `BillFacadeImpl` will be used to invoke these functions from a remote client because the attribute `ref` refers to the bean ID defined at Line 6.

We assume that the client side is also implemented in Java and XML. The XML element `sofa:reference` at Line 9 states that it uses a remote object, namely `_billFacade` due to the attribute `id`, of the class `BillFacade` to invoke remote functions. As shown by Lines 10-12 in Figure 3(a) and the Java code in Figure 3(b), a client Java class, named `OrderCenter`, is initialized by assigning the remote object `_billFacade` to its field `billFacade`. The underlying SOFA framework automatically performs this initialization. As such, in the function `queryOrder`, we can invoke the remote function `load` using the remote object.

B. Extracting Facts

We extract facts from code changes as well as code in Java and XML. At a high level, facts from code changes directly reflect where the changes occur; facts from code are extracted during the code-parsing procedure via syntax-directed translation [14] combined with lightweight dependency analysis.

1) Workflow Revisit: Code changes may not simply add a line of code as the motivating example. However, any code changes can be decomposed into code deletion and code addition. For example, Figure 4 shows that a code change that replaces the second line in (a) with lines 2-3 in (c) can be decomposed into two steps: code deletion, which removes the second line as shown in (b), and code addition, which adds two new lines as shown in (c). MICROSCOPE separately deals with code deletion and code addition in two steps, i.e., performing the workflow in Figure 2 twice.

First, MICROSCOPE finds service interfaces impacted by the code deletion. Use the example in Figure 4 as an example. In this step, we specify change(`(a):2:2`), which says that line 2 in the original code will be changed, and input the code facts extracted from the original code. In practice, if we have stored the code facts in a database, we do not re-extract facts from the original code. This step then outputs all interfaces impacted by the deletion of line 2.

Second, MICROSCOPE finds service interfaces impacted by the newly added code. Use the example in Figure 4 as an example. In this step, we specify change(`(c):2:3`), which says that lines 2-3 in (c) are changed and input the code facts extracted from the new code in (c). In practice, we do not re-extract code facts from all files but only re-extract code facts from files containing additional code. This step then outputs all interfaces impacted by the newly added code.

Next, we discuss the Datalog facts extracted from the code and the changes in detail.

2) Facts from Code Changes: We elaborate a bit more on the facts extracted from code changes. Previously, a code-change fact is specified in the form of change(loc), where loc is a string in the form "path:startline:endline", which reflects what lines of code are changed. We predefined an auxiliary predicate in(loc_1, loc_2) to check if the two locations overlap.

TABLE II: Translating XML into Datalog facts by syntax-directed translation.

ID	XML Grammar	Action for Translating XML to Datalog Facts
1	xml := elmt_0 elmt_1 ...	xml.facts = elmt_0.facts ∘ elmt_1.facts ∘ · · ·
2	elmt := <name attr_0 attr_1 .../>	elmt.id = id(); attr_i.parent = elmt.id; elmt.facts = xml_elmt(elmt.id, name, elmt.parent, elmt.loc) ∘ attr_0.facts ∘ attr_1.facts ∘ · · ·
3	elmt := <name attr_0 attr_1 ...> elmt_0 elmt_1 ... </name>	elmt.id = id(); attr_i.parent = elmt.id; elmt_i.parent = elmt.id; elmt.facts = xml_elmt(elmt.id, name, elmt.parent, elmt.loc) ∘ attr_0.facts ∘ attr_1.facts ∘ · · · ∘ elmt_0.facts ∘ elmt_1.facts ∘ · · ·
4	attr := key = value	attr.facts = xml_attr(key, value, attr.parent, attr.loc)

To specify code changes more precisely, the location string can be easily extended to include both line and column numbers. In what follows, all Datalog facts are ended with a location, which may be omitted to keep the text clean.

3) Facts from XML Files: Facts extracted from an XML file describe XML elements, attributes, and their containment relationship. Table II formally defines how an XML file is translated into Datalog facts. The second column is the XML grammar, and the third defines the actions to take when a production rule in the grammar is applied for parsing. The grammar contains three non-terminal symbols: xml, elmt, and attr. The non-terminal, xml, represents the whole XML file. The non-terminals, elmt and attr, mean XML elements and XML attributes being parsed. The first production rule in the grammar states that an XML file is a list of XML elements. The second and the third state that an element has a name and may contain attributes and sub-elements. The fourth production states that each attribute is a key-value pair.

To translate an XML file into Datalog facts, we associate each non-terminal symbol with some fields. Each non-terminal has a field, e.g., xml.facts, representing the string of Datalog facts generated after using the corresponding production rules for parsing. For example, after the first production rule is used to parse the XML file, we have already parsed each elmt_i, and elmt_i.facts have been generated. As such, the facts we generate for the whole XML file, i.e., xml.facts, is the concatenation of all elmt_i.facts (see the 2nd row and 3rd column of Table II).

Each non-terminal, elmt or attr, is associated with a field, e.g., elmt.parent, initialized as 0. Each non-terminal, elmt, is also associated with a field, e.g., elmt.id, to distinguish different XML elements with the same name. As shown in Table II, whenever the second or the third production rules are applied to parse an XML element, and after the parser visits the element's name, we create a unique and positive ID for the XML element and assign the ID to the attributes' and subelements' parent field, meaning that their parent is the current XML element. After the parser parses all XML attributes and subelements, the facts of an XML element are generated, including the fact xml_elmt, which contains the ID, name, parent, and the element's location, and the facts of its attributes and subelements.

The final row of Table II explains how we translate an XML attribute into the fact xml_attr: after the final production rule is applied, we create the fact xml_attr for the XML attribute being parsed, including the key and value of the attribute, its parent XML element, and the attribute's location.

Example 1 (XML Facts): The client's XML configuration file in Figure 3 can be translated into the three xml_elmt facts, where '...' means an omitted location. The third arguments of the first two facts are `0` because they do not belong to any parent XML elements. The third argument of the third fact is `2` because it is in the XML element of Fact 2.

1) xml_elmt(`1`, `sofa:reference`, `0`, . . .)
2) xml_elmt(`2`, `bean`, `0`, . . .)
3) xml_elmt(`3`, `property`, `2`, . . .)

The facts for attributes are like xml_attr(`id`, `billFacade`, `1`, . . .). The first two arguments form a key-value pair, and the third is `1` because it belongs to the element of Fact 1. □

We can then use the following Datalog rules, where underscore means a don't-care argument, to define beans and properties in the XML files. Each bean includes its ID, class name, and location. Each property includes an ID, its name, the bean it references, its parent bean, and its location.

```
1. bean(bean_id, class_name, bean_loc) :-
2.     xml_elmt(elmt_id, bean, _, bean_loc)
3.     xml_attr(id, bean_id, elmt_id, _)
4.     xml_attr(class, class_name, elmt_id, _)
5. prop(prop_id, prop_name, ref_bean_id, bean_id, prop_loc) :-
6.     xml_elmt(prop_id, property, bean_id, prop_loc)
7.     xml_attr(name, prop_name, prop_id, _)
8.     xml_attr(ref, ref_bean_id, prop_id, _)
```

4) Facts from Java Files: We follow a similar syntax-directed translation method to translate Java code into Datalog facts, so that the translation procedure can be integrated into Java parsers. Due to the space limit, we omit the formal description, which is similar to that for XML. Similarly, we translate Java into facts for Java classes, Java interfaces, fields, functions, expressions, call statements, and their containment relationship. These facts use Predicates 1-6 in Table III. For practical purposes, these predicates differ from those in the motivating example in two aspects. First, each predicate is assigned a unique ID. Second, the predicate function uses a mangled function name [15] so that we can distinguish functions with the same name but different parameter types (see Fact 3 in Example 2).

The second group of facts uses Predicate 7-8 in Table III to record the inheritance relationships among Java classes and interfaces. The following rules define indirect inheritance.

```
1. inherits(id, pid) :- implments(id, pid)
2. inherits(id, pid) :- extends(id, pid)
3. inherits(id, pid) :- extends(id, xid) inherits(xid, pid)
```

TABLE III: Facts generated from Java code.

ID	Predicates	Description
1	class(id, name, loc)	Java class with a unique id, the class name, and where it is defined
2	interface(id, name, loc)	Java interface with a unique id, the interface name, and where it is defined
3	field(id, cid, name, pid, loc)	A field assigned a unique id; cid indicates the Java class or interface, i.e., type, of the field; pid is the Java class where the field is declared
4	function(id, cid, name, pid, loc)	A function assigned a unique id, cid indicates the return type of the function, and name is the mangled function name
5	expr(id, cid, expr, pid, loc)	An expression that is either a variable or in the form of o.f.g...., which accesses a Java object's field; it is of the type identified by cid and is defined or used in a function with pid as the unique ID
6	call(id, eid, fid, pid, loc)	Using a Java expression whose unique ID is eid to invoke a function whose unique ID is fid; pid indicates the caller function where the call occurs
7	extends(id, pid, loc)	Java class with id extends the Java class with pid
8	implements(id, pid, loc)	Java class with id implements the Java interface with pid
9	dep(id$_1$, id$_2$)	An expression or a class field with id$_1$ data-depends on the other with id$_2$
10	alias(id$_1$, id$_2$)	An expression or a class field with id$_1$ data-is an alias of the other with id$_2$

Additionally, we use a lightweight must alias analysis [16], which is also based on Datalog, to build alias and data dependency facts among variables or expressions (Predicates 9-10 in Table III). The Datalog rules for the alias analysis are omitted as they are not our contribution and can be found in the previous work [16]. Since the alias analysis is also Datalog-based, they can seamlessly work with MICROSCOPE. As explained in §III-A, SOFAStack's XML files may introduce extra aliases if two bean properties reference the same bean. These extra alias rules can be found below, where Lines 2-3 and Lines 6-7 find the properties that reference the same bean ID, ref_bean_id. Other lines find the corresponding class member fields in the Java source files.

```
1.  alias(field_id_1, field_id_2)  :-
2.      bean(bean_id_1, class_name_1, _)
3.      prop(_, prop_name_1, ref_bean_id, bean_id_1, _)
4.      class(class_id_1, class_name_1, _)
5.      field(field_id_1, _, prop_name_1, class_id_1, _)
6.      bean(bean_id_2, class_name_2, _)
7.      prop(_, prop_name_2, ref_bean_id, bean_id_2, _)
8.      class(class_id_2, class_name_2, _)
9.      field(field_id_2, _, prop_name_2, class_id_2, _)
```

The data dependency facts include direct def-use relations and indirect data dependencies hidden behind pointer aliases. Although a must-alias analysis helps improve MICROSCOPE's precision, we acknowledge that its unsoundness may lead to false positives or negatives in the change-impact analysis. Nevertheless, our experiments (see §V-C) show that MICROSCOPE reports only a moderate number of false positives and false negatives, which are manageable in practice.

Example 2 (Java Facts): Consider the client's Java code in Figure 3(b). We will have the following facts, where '...' means an omitted argument, such as the locations.

```
1) class(1, OrderCenter, ...)
2) field(2, ..., billFacade, 1, ...)
3) function(3, ..., queryOrder(Mgr), 1, ...)
4) expr(4, ..., o, 3, ...)
5) expr(5, ..., o.f, 3, ...) /*defined in the assignment*/
6) expr(6, ..., o.f, 3, ...) /*used in the call*/
7) expr(7, ..., this.billFacade, 3, ...)
8) call(8, 6, ..., 3, ...)
```

Facts 1-3 describe the class, as well as the fields and functions of the class. The function uses a mangled name that includes the parameter types. Facts 4-7 describe the variables or expressions defined or used in the function. We have two predicates for the expression `o.f` as it is defined and used in different locations. Fact 8 describes the call, which uses the expression (ID=6, i.e., `o.f`) to invoke the callee named `load`. In addition, via a lightweight pointer analysis, the following predicates are held to be true: dep(5, 7); dep(6, 7); alias(5, 7); alias(5, 6); alias(6, 7). We do not need to enumerate these dependency/alias facts before the change impact analysis but query the underlying pointer analysis on demand. □

C. Defining Relationship Rules

Datalog rules define the relationships among the facts and allow us to reason functions impacted by code changes, published to clients, and referenced by clients. While different microservice frameworks, which may not use XML like Figure 3, may require us to define different relationship rules, we argue that Datalog provides an easy manner to describe relationships and, once defined, the relationships are not changed due to their generality for a microservice framework.

1) Identifying Published Functions: SOFAStack publishes functions via XML elements named `sofa:service` as shown in Figure 3(a). The complete Datalog rule for identifying published functions is listed below.

```
1.  pub_function(func_id, func_name, class_id, func_loc)  :-
2.      function(func_id, _, func_name, class_id, func_loc)
3.      class(class_id, class_name, _)
4.      bean(bean_id, class_name, _)
5.      xml_elmt(elmt_id, sofa:service, _, _)
6.      xml_attr(ref, bean_id, elmt_id, _)
```

Lines 2-3 state that a published function is defined in a Java class named class_name. Line 4 states that the class is declared as a bean in the configuration file. Lines 5-6 state that the bean is referenced by an XML element named `sofa:service`.

2) Identifying Referenced Functions: SOFAStack allows a client to use a remote function that is declared in an XML element named `sofa:reference`, as exemplified in Figure 3(a). The rule for identifying referenced functions is listed below.

```
1.   ref_function(func_id, func_name, pid, func_loc)  :-
2.       function(func_id, _, func_name, pid, func_loc)
3.       inherits(pid, interface_id)
4.       interface(interface_id, interface_name, _)
5.       xml_elmt(elmt_id, sofa:reference, _, _)
6.       xml_attr(id, remote_obj, elmt_id, _)
7.       xml_attr(interface, interface_name, elmt_id, _)
8.       bean(bead_id, class_name, _)
9.       prop(prop_id, prop_name, remote_obj, bead_id, _)
10.      field_name = prop_name
11.      class(class_id, class_name, _)
12.      field(field_id, _, field_name, class_id, _)
13.      alias(eid, field_id)
14.      call(_, eid, func_id, _, _)
```

The rule can be understood in three parts: Lines 2-7, 8-10, and 11-14. Line 2 says that a referenced function is a function. Line 3 says that the Java class where the function is defined inherits a Java interface. Line 4 finds this Java interface, namely interface_name. Lines 5-7 state that the Java interface and its corresponding remote object are declared in an XML element named `sofa:reference`.

Lines 8-10 state that the remote object, remote_obj, is assigned to a field, field_name, of a Java class, class_name. Consider Figure 3. This is to find the bean and the property XML elements on the client side, where remote_obj = `_billFacade`, field_name = `billFacade`, class_name = `com.store.OrderCenter`.

Line 11 finds the Java class according to the class name. Line 12 finds the field that receives the remote object in the Java class. Line 13 finds the field's aliases, i.e., aliases of the remote object, which invoke the remote function at Line 14.

3) Identifying Impacted Functions: We then identify functions impacted by code changes in a few steps. First, we identify Java classes and functions impacted by code changes. We say code changes impact a Java class if one of its fields or parent classes is impacted by code changes (Lines 1-4). Similarly, a function is impacted by code changes if one variable or expression it defines or uses is impacted by code changes (Lines 5-6).

```
1.   imp_class(id, name, loc)  :-
2.       class(id, name, loc) imp_field(_, _, _, id, _)
3.   imp_class(id, name, loc)  :-
4.       class(id, name, loc) inherits(id, pid) imp_class(pid, _, _)
5.   imp_function(fid, fname, pid, floc)  :-
6.       function(fid, _, fname, pid, floc) imp_expr(_, _, _, id, _)
```

The rules above depend on imp_field() and imp_expr(), which specify a class field or an expression impacted by code changes. The rules for imp_expr() are listed below, which are put into four cases. The first case states that if an expression is defined or used at a location where the changes occur, the expression is impacted by code changes. The second case states that if an expression is of a type (i.e., a Java class) impacted by code changes, it is also impacted. The third and the fourth state that if an expression depends on another expression or a field impacted, it is also impacted.

```
1.   imp_expr(id, cid, expr, pid, loc)  :-
2.       expr(id, cid, expr, pid, loc) change(cloc) in(loc, cloc)
3.   imp_expr(id, cid, expr, pid, loc)  :-
4.       expr(id, cid, expr, pid, loc) imp_class(cid, _, _)
5.   imp_expr(id, cid, expr, pid, loc)  :-
6.       imp_expr(eid, _, _, _, _) dep(id, eid)
7.   imp_expr(id, cid, expr, pid, loc)  :-
8.       imp_field(fid, _, _, _, _) dep(id, fid)
```

The rules that define imp_field() are listed below. Lines 1-2 state that a field is impacted if it is declared at the location where code changes occur. Lines 3-4 state that a field is impacted if it is of a type, i.e., a Java class, impacted by code changes. Lines 5-6 state that a field is impacted if it depends on an expression impacted.

```
1.   imp_field(id, cid, name, pid, loc)  :-
2.       field(id, cid, name, pid, loc) change(cloc) in(loc, cloc)
3.   imp_field(id, cid, name, pid, loc)  :-
4.       field(id, cid, name, pid, loc) imp_class(cid, _, _)
5.   imp_field(id, cid, name, pid, loc)  :-
6.       imp_expr(eid, _, _, _, _) dep(id, eid)
7.   imp_field(id, cid, name, pid, loc)  :-
8.       field(id, cid, name, pid, loc) class(pid, pname, _)
9.       bean(bid, pname, _) imp_prop(_, name, _, bid, _)
```

Lines 7-9 above are a bit different. Recall the XML configuration file in Figure 3(a) — SOFAStack allows us to initialize a class field using the XML element named `property`. Thus, Lines 7-9 above state that code changes impact a class field if the corresponding XML element named `property` is impacted by code changes, denoted by the predicate imp_prop() and defined below. The idea is similar: an XML element named `property` is impacted by code changes if the changes happen there (Lines 1-3 below) or the property element references an impacted bean (Lines 4-6 below).

```
1.   imp_prop(prop_id, prop_name, ref_bean_id, bean_id, loc)  :-
2.       prop(prop_id, prop_name, ref_bean_id, bean_id, loc)
3.       change(cloc) in(cloc, prop_loc)
4.   imp_prop(prop_id, prop_name, ref_bean_id, bean_id, loc)  :-
5.       prop(prop_id, prop_name, ref_bean_id, bean_id, loc)
6.       imp_bean(ref_bean_id, _, _)
```

The above rules then depend on the predicate imp_bean(), representing the beans impacted by code changes. As listed below, a bean is impacted by code changes if the changes happen at the location where the bean is declared (Lines 1-2), the bean references another impacted bean (Lines 3-5), or the bean references a class impacted by code changes (Lines 6-7).

```
1.   imp_bean(bid, cname, loc)  :-
2.       bean(bid, cname, loc) change(cloc) in(cloc, loc)
3.   imp_bean(bid, cname, loc)  :-
4.       bean(bid, cname, loc) prop(_, _, ref_bean_id, bid, _)
5.       imp_bean(ref_bean_id, _, _)
6.   imp_bean(bid, cname, loc)  :-
7.       bean(bid, cname, loc) imp_class(_, cname, _)
```

4) Summary: All the above rules are finally reduced to facts extracted from code changes and source code. Thus, by combining all code and change facts with the above rules, a

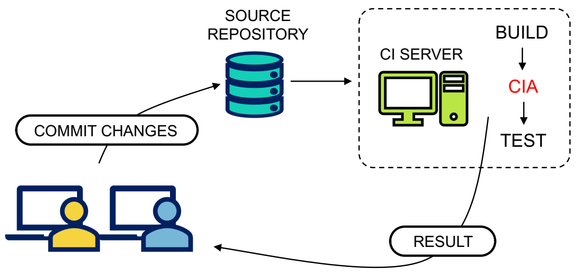


Fig. 5: MICROSCOPE working with CI.

Datalog engine can then reason functions (i.e., microservice interfaces) impacted by code changes, published to clients, and referenced by clients.

D. Extending to Other Languages and Frameworks

Recall that we translate both Java and XML into basic elements in their languages and the containment relationship among the elements. Such code elements (e.g., classes, functions, expressions, etc) and containment relationships are shared by common programming languages and microservice frameworks such as Spring Boot and Apache Dubbo, to name a few. Thus, we argue that our Datalog-based approach discussed above can be easily extended to other languages. Due to the space limits, we discuss a subset of them. Datalog rules for nine common programming and configuration languages can be found in our artifact.

IV. IMPLEMENTATION

We implement our approach as a tool named MICROSCOPE to support CIA. The tool is now open-sourced and publicly available. In MICROSCOPE, the components of extracting facts from the source code are implemented as plugins of each language parser. With the facts extracted from the source code, we use Souffle [10], a state-of-the-art Datalog engine, to infer impacted microservice interfaces. In what follows, we discuss a few important details of the implementation.

Continuous Integration. MICROSCOPE is now working as a critical phase in the continuous integration (CI) system, as illustrated in Figure 5. In the CI system, MICROSCOPE works after developers commit an update of the code to help identify impacted service interfaces. Without MICROSCOPE, the CI system is unaware of what service interfaces are affected by code changes and, thus, has to invoke regression testing to run all test cases. By contrast, MICROSCOPE allows the CI system to identify interfaces impacted by the code changes, thus running only a small subset of test cases and speeding up the CI procedure.

Incremental Building. Due to the frequent code changes in the industry, it is not efficient and also not necessary to re-extract the facts from all source code or configuration files. Instead, due to the integration of MICROSCOPE and the CI system, MICROSCOPE benefits from the incremental strategy of common build systems, e.g., Maven and Gradle, which only rebuild code containing changes. As such, MICROSCOPE also only re-extracts facts from files containing changes and, thus, is highly efficient even in the face of frequent changes in industrial settings.

TABLE IV: Microservice applications for evaluation.

Apps	KLoC	#F	#I	Java	XML	JS	TS	Others
Monitor	127	1,220	703	✔	✔	✔		✔
Ledger	104	927	428	✔	✔			✔
Authorization	296	2,535	613	✔	✔	✔		✔
TradeCenter	160	1,364	216	✔	✔	✔		✔
ClearingCenter	604	4,501	4,036	✔	✔	✔	✔	✔
SmartEngine	366	3,440	725	✔	✔			✔
PriceCenter	72	668	776	✔	✔	✔		✔
TransferCenter	233	2,176	430	✔	✔	✔		✔
RejectPayment	102	975	243	✔	✔	✔		✔
Regression	90	818	1,288	✔	✔	✔	✔	✔
TCC-Transaction	26	438	233	✔	✔	✔		✔
Incubator-Seata	287	2,042	80	✔	✔		✔	✔
Shenyu	300	2,930	1,420	✔	✔			✔
OpenSPG	94	1,019	267	✔	✔			✔

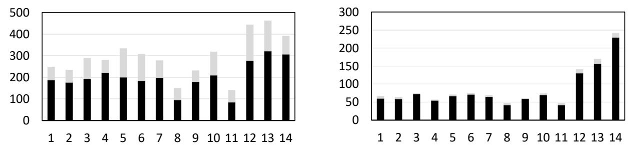
V. EVALUATION

As shown in Figure 5, MICROSCOPE helps identify impacted service interfaces after developers commit an update of the code into the repository. As such, we only need to focus on impacted interfaces in regression testing. In particular, we evaluate the effectiveness and efficiency of MICROSCOPE by investigating the following three research questions:

- **RQ1:** How efficient is MICROSCOPE in extracting the relational representation from multilingual microservices?
- **RQ2:** What is the overhead of MICROSCOPE in identifying impacted microservice interfaces for a commit?
- **RQ3:** How effective is MICROSCOPE in identifying impacted microservice interfaces?

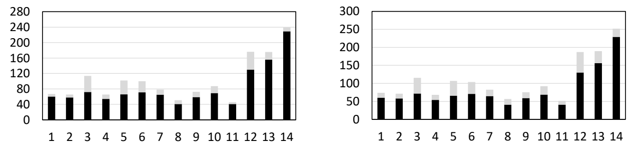
We aim to evaluate MICROSCOPE's performance in real industry settings, thus using ten most frequently updated core applications from Ant Group, a Global 500 company, as well as four open-sourced applications [17]–[20]. The details of these applications are listed in Table IV. Their sizes range from 26 to 600 KLoC (Kilo Lines of Code), containing up to 4,501 source code files (#F) and hundreds to thousands of interfaces (#I) exposed to the external world. On average, each application involves at least four languages.

For RQ1, we translate all applications' source files into our cross-language representation and record the time cost for evaluation. Note that each experiment is conducted twenty times, and the average time cost is reported. To assess the change impact analysis, i.e., RQ2 and RQ3, we select the main development branch of each application and extract the code changes from the latest eight commits, yielding a total of 112 commits or code changes. Of the 112 code changes, 67 are considered significant as they involve semantic changes to Java code or configuration files. These significant changes are then fed to the evaluation of RQ2 and RQ3, which evaluate the efficiency and effectiveness in identifying impacted microservice interfaces.



(a) Clean build (b) Incremental build

Fig. 6: Time overhead of extracting facts (RQ1). The X-axis lists the microservice applications. The Y-axis is time in seconds. ■: build time; ▨: extraction time.



(a) Reasoning overhead (b) Total overhead

Fig. 7: Time overhead of change impact analysis (RQ2). The X-axis lists the microservice applications. The Y-axis is time in seconds. ■: time of incremental build; ▨: overhead of (a) Datalog reasoning and (b) fact extraction + Datalog reasoning.

In our experiments, we failed to find any existing technique supporting so many programming or configuration languages. In other words, this is the first static CIA for multilingual microservices. Thus, we cannot compare it to other similar tools. To show the efficiency of our approach (RQ1 and RQ2), we compare the time cost of CIA with the building time of each application, which we believe is sufficient to show MICROSCOPE's efficiency. To show the effectiveness (RQ3), we discuss the false positives and negatives reported by MICROSCOPE. Also, to show the necessity of the language-agnostic solution, we compare the false positive and false negative rates to the solution where we remove the support for some languages. Putting all experiment results together, we can make the conclusion below.

By spending a very short time, i.e., about 38% of the time for a common incremental build (RQ1 and RQ2), MICROSCOPE can precisely identify change-impacted interfaces and reduce 97% of regression test cases and save 73% of testing time (RQ3).

All experiments are conducted on a laptop with a 6-core Intel i7-9750H CPU @ 2.60GHz and 16GB of RAM.

A. RQ1: Efficiency of Fact Extraction

As shown in Figure 2, the first step of MICROSCOPE is to extract facts from the code and the code changes, which are implemented as a part of the code parser. Figure 6(a) displays the overhead of MICROSCOPE compared to a clean build. A clean build means that the code is built from scratch. Thus, we re-extract all facts from the code. As shown in the figure, extracting the facts exhibits a moderate, i.e., 46% on average, time overhead compared to a clean build. In other words, extracting code and code-change facts takes less than half the time for a clean build.

In practice, we often do not build a software project from scratch but leverage a build system's capability of incremental building. That means we only rebuild files containing changes. MICROSCOPE, as a part of the parser, only re-extracts the facts from the files with changes. Figure 6(b) displays the time overhead of fact extraction in an incremental build. The overhead ranges from 3% to 15%, 8% on average, which is quite low and often imperceptible. For the largest project with over half a million lines of code, i.e., ClearingCenter, MICROSCOPE exhibits an average overhead of 8%, demonstrating that the scalability of MICROSCOPE is graceful and MICROSCOPE is practical even for large codebases.

Answer to RQ1: MICROSCOPE can extract code facts with an average overhead of 8% in a common incremental build, showing its graceful scalability for CIA.

B. RQ2: Efficiency of CIA

As shown in Figure 2, after extracting facts from the code and the code changes, we use a Datalog engine to reason the impacted interfaces. The time overhead of reasoning change impacts over an incremental build — which is more common in practice than a clean build — is shown in Figure 7(a). As plotted, the reasoning procedure exhibits 11% to 55% overhead, 23% on average, over an incremental build.

Figure 7(b) shows the time cost of the whole CIA procedure, including the fact extraction time discussed in RQ1 and the reasoning time. The evaluation results show that the total time cost of a change impact analysis usually takes 23% to 62% (38% on average) of the build time, demonstrating a moderate and manageable overhead in practice.

Answer to RQ2: On average, MICROSCOPE can identify change-impacted interfaces by 38% of the incremental build time, showing a moderate time overhead.

C. RQ3: Effectiveness of CIA

As shown in Table V, among the 112 examined commits (or code changes), MICROSCOPE finds that 338 out of 11,458 microservice interfaces in either upstream or downstream applications are impacted, with 59 false positives and 20 false negatives. On average, the false positive rate is 17%, and the false negative rate is 6%. However, given a solution that does not address the multilingual problem, the false negative rate may significantly be increased. For instance, as shown in Table V, even if we remove the support for only one language, say XML, we will miss about 4× impacted interfaces, demonstrating the necessity and value of our language-agnostic approach.

Consider that the impacted interfaces only account for 3% of the total. Compared to the situation where we do not have a CIA for multilingual microservices and have to test all interfaces, our CIA lets the subsequent testing procedure focus

TABLE V: The number of all interfaces (#AI), impacted interfaces (#II), false positives (#FP) and negatives (#FN), as well as the impacted interfaces (#II'), the false positives (#FP') and negatives (#FN') if we do not support XML.

Apps	#AI	#II	#FP	#FN	#II'	#FP'	#FN'
1	703	50	7	9	41	9	20
2	428	0	0	0	0	0	0
3	613	9	1	0	9	1	0
4	216	0	0	0	0	0	0
5	4,036	0	0	0	0	0	0
6	725	60	10	4	54	10	10
7	776	117	18	0	97	18	20
8	430	0	0	0	0	0	0
9	243	17	2	1	11	2	7
10	1,288	50	16	4	42	16	12
11	233	5	0	1	6	1	1
12	80	3	0	0	2	0	1
13	1,420	18	3	1	14	4	6
14	267	9	2	0	6	2	3
Total	11,458	338	59	**20**	282	63	**80**
%	-	-	17%	**6%**	-	22%	**27%**

on a small number of impacted interfaces, reducing 97% of interfaces to test. As for how much testing time we can save, it may differ a lot in different applications because the test suite may contain different numbers of test cases for each interface. According to our statistics, for the fourteen microservices, our CIA lets us save 73% of testing time on average every time a code change is committed to the code repository.

Answer to RQ3: MICROSCOPE can effectively identify change-impacted interfaces, allowing the subsequent testing procedure to focus only on 3% out of all interfaces, saving 73% of testing time.

D. Discussion on False Positives and Negatives

False positives mean we misreport service interfaces unaffected by code changes. Regression testing then has to test interfaces that have been misreported to be impacted by changes. Testing more interfaces does not affect the testing effectiveness but the efficiency. We have shown that our approach can reduce the testing efforts significantly. Thus, false positives are less important in our application scenario and are mainly attributed to the imprecision of the underlying alias and dependence analysis. As discussed before, MICROSCOPE applies existing alias and dependence analysis, thus inheriting their imprecision.

False negatives mean that we may miss change-impacted interfaces such that regression testing will also not test them. This situation, in theory, may degrade the effectiveness of regression testing, leaking bugs into the product. However, we observed few such issues in practice. Currently, false negatives are mainly attributed to engineering issues, e.g., currently, MICROSCOPE does not support anonymous Java classes. Figure 8 shows an example where Line 2 creates an anonymous Java class that overrides and implicitly invokes the

```
1. boolean updateInfo(Operator operator) {
2.   var result = template.execute(Result.class, new Callback() {
3.         @Override
4.         public Result executeService() { ... }
5.   });
6.   throw new Exception(result.getCode());
7. }
```

Fig. 8: The cause of false negatives.

function `executeService`. Our Datalog rules currently do not capture such implicit calls. This problem can be addressed by integrating more advanced call graph analysis, which is left as our future work.

VI. RELATED WORK

Program analysis in enhancing microservices has gained significant recognition in recent years [21]–[25]. Datalog-based program analysis has also increased interest over the past decade, with many noteworthy examples [26]–[32]. Despite a lot of progress, the capability of handling multiple languages is still limited. This underscores the need for our multi-lingual solution, where not only do the Datalog rules provide the capability of reasoning change impacts, but the Datalog representation also provides an abstraction layer over diverse languages.

Traditional CIAs utilize program slicing to identify program locations potentially influenced by code changes [7], [33]. They typically work on graphical representations of the program, such as program or system dependency graph [34], [35], determining the impacted program locations through specific graph reachability problems. In contrast, MICROSCOPE maintains only the syntactical constructs in its relational representation and uses Datalog solvers to derive semantic properties, providing a more adaptable and comprehensive approach to change impact analysis.

VII. CONCLUSION

This paper introduces a language-agnostic change impact analysis via Datalog, namely MICROSCOPE, to pinpoint microservice interfaces exposed to the external world and impacted by code changes. Through our evaluation of a leading software company, we conclude that MICROSCOPE can efficiently and precisely identify change-impacted interfaces, with the potential of saving testing efforts.

REFERENCES

[1] N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, and D. Pei, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'21. ACM, 2021, pp. 527–539.

[2] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, and Z. Zang, "Rapid and robust impact assessment of software changes in large internet-based services," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT'15. ACM, 2015, pp. 2:1–2:13.

[3] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[4] Y. Wu, B. Chai, Y. Li, B. Liu, J. Li, Y. Yang, and W. Jiang, "An empirical study on change-induced incidents of online service systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP'23. ACM, 2023, pp. 234–245.

[5] R. S. Arnold, *Software Change Impact Analysis*. Washington, DC, USA: IEEE Computer Society Press, 1996.

[6] S. Lehnert, "A taxonomy for software change impact analysis," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, ser. IWPSE-EVOL'11. ACM, 2011, pp. 41–50.

[7] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE'11. ACM, 2011, pp. 746–755.

[8] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'14. ACM, 2014, pp. 343–348.

[9] H. Cai and D. Thain, "DistIA: a cost-effective dynamic impact analysis for distributed programs," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'16. ACM, 2016, pp. 344–355.

[10] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, ser. CAV'16. Springer, 2016, pp. 422–430.

[11] https://owasp.org/API-Security/editions/2023/en/0x11-t10/, 2023.

[12] https://www.sofastack.tech/en/, 2023.

[13] https://spring.io/projects/spring-boot, 2023.

[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Addison Wesley, 2007.

[15] https://en.wikipedia.org/wiki/Name_mangling, 2024.

[16] G. Balatsouras, K. Ferles, G. Kastrinis, and Y. Smaragdakis, "A datalog model of must-alias analysis," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP'17. ACM, 2017, pp. 7–12.

[17] https://github.com/changmingxie/tcc-transaction, 2023.

[18] https://github.com/apache/incubator-seata, 2023.

[19] https://github.com/apache/shenyu, 2023.

[20] https://github.com/openspg/openspg, 2023.

[21] T. Yang, J. Shen, Y. Su, X. Ling, Y. Yang, and M. R. Lyu, "AID: efficient prediction of aggregated intensity of dependency in large-scale cloud systems," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'21. IEEE, 2021, pp. 653–665.

[22] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, ser. ICSE'23. IEEE, 2023, pp. 1750–1762.

[23] S. Zhang, P. Jin, Z. Lin, Y. Sun, B. Zhang, S. Xia, Z. Li, Z. Zhong, M. Ma, W. Jin *et al.*, "Robust failure diagnosis of microservice system through multimodal data," *IEEE Transactions on Services Computing*, vol. 16, no. 6, pp. 3851–3864, 2023.

[24] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial SOA applications: a case study at alibaba," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'20. ACM, 2020, pp. 1477–1486.

[25] Z. Zhong, J. Liu, D. Wu, P. Di, Y. Sui, and A. X. Liu, "Field-based static taint analysis for industrial microservices," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP'22. IEEE, 2022, pp. 149–150.

[26] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA'09. ACM, 2009, pp. 243–262.

[27] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *Proceedings of the 30th European Conference on Object-Oriented Programming*, ser. ECOOP'16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:25.

[28] Y. Yuan, Z. Liu, and S. Wang, "Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software," in *Proceedings of the 32nd USENIX Security Symposium*, ser. Security'23. USENIX, 2023, pp. 2009–2026.

[29] T. Szabó, S. Erdweg, and G. Bergmann, "Incremental whole-program analysis in datalog with lattices," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI'21. ACM, 2021, pp. 1–15.

[30] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'14. ACM, 2014, pp. 239––248.

[31] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing lattice-based program analyses in datalog," *Proceedings of ACM Programing Language*, vol. 2, no. OOPSLA, pp. 139:1–139:29, 2018.

[32] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC'16. ACM, 2016, pp. 196–206.

[33] F. Angerer, A. Grimmer, H. Prahofer, and P. Grunbacher, "Configuration-Aware Change Impact Analysis (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'15. IEEE, 2015, pp. 385–395.

[34] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[35] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.