RPCover: Recovering gRPC Dependency in Multilingual Projects

Aoyang Fang*, Ruiyu Zhou†, Xiaoying Tang*, Pinjia He†‡
* School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen (CUHK Shenzhen), China
† School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK Shenzhen), China
{aoyangfang, ruiyuzhou}@link.cuhk.edu.cn; {tangxiaoying, hepinjia}@cuhk.edu.cn

***Abstract*—The advent of microservice architecture has led to a significant shift in the development of service-oriented software. In particular, the use of Remote Procedure Call (RPC), a mode of Inter-Process Communication (IPC) prevalent in microservices, has noticeably increased. To figure out the relationships between services and obtain a high-level understanding of service-oriented software, a line of recent work focuses on the dynamic construction of service call graphs, which relies on the preliminary deployment of services and only captures the calling relationships within a specific time frame. Meanwhile, static methods avoid the need for pre-deployment and often provide a more stable and complete graph compared to dynamic techniques. However, research and practical applications of static call graph construction remain relatively unexplored.**

This paper introduces RPCover, a novel gRPC dependency recovery framework that facilitates the interconnection of services across various programming languages using their static gRPC calls. In addition, due to the lack of a multilingual microservice benchmark that uses gRPC, we build the first multilingual benchmark RPCoverBench that contains complex gRPC call relations. RPCover has been evaluated on a single language benchmark (DeathStarBench) and our multilingual benchmark (RPCoverBench). The results show that RPCover effectively recovers 99.33% of the use cases of gRPC calls with less than 200% of the overhead compared with a single-language semantic dependency analyzer.

***Index Terms*—gRPC, dependency recovery, microservice**

I. Introduction

Microservice architecture [1], [2] has become increasingly popular in software development due to its ability to improve agility, scalability, and reliability. However, with the growing usage of microservice, several challenges arise. As the level of microservice system complexity increases, there is a risk of accumulating technical debt [3]. One notable debt is the proliferation of point-to-point connections among services, which can lead to significant costs in terms of system evolution and maintenance [3]. To mitigate such technical debt, many researchers have attempted to perform service-level dependency analysis to reconstruct the architecture of the services.

Prior research in the field of service-level dependency analysis can be broadly classified into two main methodologies: *dynamic analysis* [4]–[13] and *static analysis* [14]–[20]. Dynamic analysis is a technique that executes the system across a range of inputs. Instead of extracting system data from the source code, the analysis is conducted by evaluating generated logs or collected run-time metrics. This approach provides real-time insights into system behaviors under different conditions. On the other hand, static analysis is a method that leverages the code base to reconstruct the system architecture. This is achieved by utilizing the system information embedded within the source code and other associated artifacts, offering a comprehensive understanding of the inherent system structure and potential dependencies. Despite the capabilities of both dynamic and static methods in analyzing services and their dependencies, current works have their own limitations.

Dynamic analysis, which tracks service dependency relationships using real-time metrics such as logs and traces, can yield results that vary significantly when metrics are collected at different times. A study by Luo et al. [11] reveals that a single online service can have up to nine distinct classes of topologically different graphs, which may influence analyzing runtime performance significantly. Additionally, dynamic analysis requires a well-established infrastructure, including available relevant logs and traces, for effective conduct. However, not all systems have this kind of infrastructure in place.

Static analysis can offer a partial solution to the limitations of dynamic analysis in terms of producing consistent results for the same code repository. However, it inherently lacks completeness because it sacrifices the language-independent characteristics of dynamic analysis [14], [15], [18]. This limitation poses challenges to its applicability in multilingual services, rendering it incomplete in this context, as there is no trace indicating the calling chain. While certain efforts have been made to unify static analysis across different programming languages [17], these methods typically require extra efforts to accommodate various programming languages.

In a microservice architecture, gRPC is a commonly used communication mechanism among services. However, existing studies do not adequately illustrate the complete gRPC calling relations among microservices. gRPC requests are described in an intermediate representation (IR) and then compiled into interfaces in multiple languages. It is challenging to extract call graphs among microservices by utilizing conventional single-language semantic analysis, as it cannot trace the call relation to another microservice. Schiewe et al. [17] proposed the use of LAAST (Language-Agnostic Abstract Syntax Tree) to potentially alleviate this issue. However, the extension of LAAST to other programming languages presents a challenge because the abstractions differ between languages.

‡Corresponding author

To tackle these challenges, this paper introduces RPCover as the first framework to automatically identify the dependencies between gRPC (a widely used implementation of RPC) in multilingual microservices. Our approach begins with an *intra-service* analysis, where a single-language semantic dependency analyzer is applied to each microservice to extract dependencies within the service. Subsequently, an *inter-service* analysis is performed using gRPC definitions. This analysis, which involves matching all possible call and implementation sites of gRPC services and requests in the dependency indexes, allows us to establish the gRPC call relations among microservices. Finally, RPCover integrates the inter-service dependency graph into a unified index file, ensuring compatibility with various editors and graph viewers. This enables easy integration and analysis of the output graph and its further visualization.

In addition, we build a microservice benchmark incorporating multiple remote procedure calls (RPC) invocations among these services, written in five distinct programming languages. This benchmark enables us to effectively evaluate the performance and adaptability of our framework. Results indicate that RPCover achieves 99.33% accuracy in the benchmarks, with an overhead less than 200% compared to LSIF language semantic dependency analyzers.

In summary, this paper makes the following contributions:

- **Approach**. We propose RPCover, a gRPC dependency recovery technique based on the SCIP index, which provides easy adoption to different projects and programming languages.
- **Benchmark**. We construct a microservice benchmark consisting of fifteen services across five widely-used backend programming languages, which simulates potential design flaws in production environments and allows performance assessment of software architecture reconstruction.
- **Open source**. Our tool [21] and benchmark [22] have been publicly released.

II. Background

This section provides an overview of the fundamental concepts employed in this paper, including microservice, Software Architecture Reconstruction (SAR), Remote Procedure Call (RPC), dependency terminology, Protocol Buffers, Language Server Index Format (LSIF), and SCIP Code Intelligence Protocol (SCIP).

A. Microservices

Microservices, an architectural style inspired by service-oriented computing, facilitates the design of highly scalable and maintainable software through orchestrating compact services [23]. However, the lack of a definitive standard for determining architectural quality can lead to unintentional technical debt [24], [25]. This includes issues such as excessive inter-service connections, inappropriate incorporation of business logic within the communication layer, and poor source code management across different services [3]. To address these challenges, many researchers are working on reconstructing the software architecture, which will be introduced in the next section.

B. Software Architecture Reconstruction

Given the potential for architectural deterioration due to technical debt [25], proactive software architecture reconstruction becomes crucial for identifying design issues and maintaining organized services. Dynamic and static analysis are the primary methods for this process [26].

1) Dynamic Analysis: Dynamic analysis, independent of programming language, captures intricate software behavior and extracts runtime metrics. However, its reliability depends on tracking the function calling chain within a time frame. A missing calling chain will result in an incomplete dependency graph [4]. Common approaches to reconstruct microservice architecture include log analysis [5], tracing analysis [6]–[11], and monitoring [12], [13].

2) Static Analysis: Static analysis provides a consistent view of software architecture, proving beneficial for preemptive use prior to service deployment. It facilitates early detection of potential issues, thereby ensuring stability and performance. Tools such as Microvision [14] and LAAST employ Abstract Syntax Tree (AST) for architectural reconstruction. While Microvision is language-specific, LAAST [17] aims to unify ASTs across various languages. However, static analysis sacrifices the language-independence characteristic of dynamic analysis, as it necessitates a unique frontend for each programming language to extract relevant information during the analysis process. This requirement imposes additional implementation works.

C. Remote Procedure Call

Remote Procedure Call (RPC), a protocol allowing procedures to execute on remote systems as if they were local, simplifies networking and distributed application development [27]–[29]. gRPC, a high-performance and open-source implementation of RPC, has shaped current RPC standards and efficiently connects services across data centers. Extensive research has been conducted to optimize the RPC process, such as speeding up the RPC process [30], identifying the critical paths of microservices [31], and analyzing the overall structure of microservices [10].

D. Dependency

We classify dependencies into two types: *service-level dependency* and *function-level dependency*. Service-level dependencies exist when one microservice relies on another, often through gRPC calls. Function-level dependencies occur when one function depends on another. If two dependent functions reside in separate services, both service-level and function-level dependencies are present. If they are in the same service, only a function-level dependency exists. For instance, an RPC call represents a service-level dependency, while a regular function call indicates a function-level dependency.

E. Protocol Buffers

Protocol Buffers [32] (Protobuf) is a binary serialization format developed by Google for efficient communication between systems. It uses a schema to define the structure of data and encodes it into a compact binary format, resulting in smaller message sizes. Protobuf finds applications [33] in network communication [34], data storage, and inter-process communication, offering flexibility and compatibility across multiple programming languages.

Protocol Buffers are widely used as an intermediate representation (IR) for describing inter-language structures. For example, gRPC defines its service and request objects in Protocol Buffers and then utilizes Protocol Buffer compiler plugins to translate these definitions into interfaces in multiple languages.

F. Representations of Code Dependencies(LSIF and SCIP)

The Language Server Index Format [35] (LSIF) and SCIP Code Intelligence Protocol (SCIP) [36] are popular language-independent representations of code dependencies. They use a general structure to encode the semantic relationship for code navigation, including definition, reference, and implementation relationships. LSIF encompasses relations between semantic symbols using edge labels like *textDocument/definition*, *textDocument/references*, and *textDocument/implementation*. Fig. 1 provides an example of *textDocument/definition*. The LSIF indexer will translate the input source code into a graph that delineates nodes and edges. Each node represents an identifier, while each edge signifies a relationship between nodes.

Unlike LSIF, SCIP, which is developed by Sourcegraph [37], encodes rich semantic dependency information directly in nodes, thereby eliminating the need for constructing a complex graph. For example, the definition relationship in SCIP is preserved as an attribute of the reference site, rather than a labeled edge connecting the definition node and the reference site. By forgoing graph representation and instead employing Protobuf encoding of symbols and relationships, Sourcegraph asserts that SCIP indexes are, on average, four times smaller when compressed with gzip, compared to their LSIF counterparts. Furthermore, uncompressed LSIF payloads are about five times larger in size [38]. SCIP also maintains compatibility with LSIF by a converter that is capable of generating an LSIF index from SCIP input.

III. Approach

Our research method, shown in Fig. 2, analyzes a collection of microservices across different languages with complex gRPC dependencies. The process begins with an *Intra-Service Analysis*, where SCIP is used to create index files that outline dependencies in each microservice. Subsequently, RPCover carries out an *Inter-Service Analysis* to identify gRPC dependencies between microservices, with the *gRPC Matcher* consolidating these into a unified index file. This unified SCIP index is then transformed into an LSIF index for further analysis. Finally, we employ a graph database to visualize the uncovered dependencies, offering an intuitive understanding of the interrelationships between services.

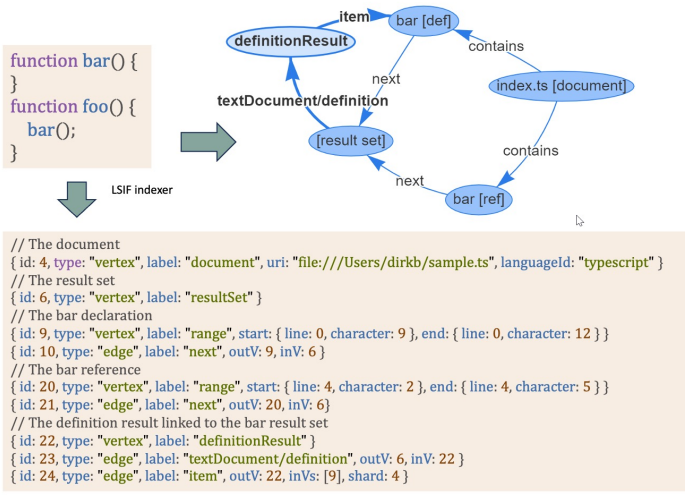

Fig. 1. Example of `textDocument/definition` in LSIF [39]. The reference site of the function `bar()` in `foo()` is represented as a node labeled `bar [ref]` in the LSIF index. It processes a `[result set]` node which maintains an `textDocument/implementation` edge to its definition node labeled `bar [def]`.

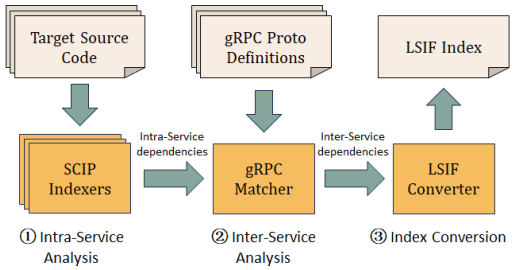

Fig. 2. Workflow of RPCover

A. Intra-Service Analysis

Within each microservice, gRPC Protocol Buffer compiler plugins transform the gRPC service and request definitions into stubs, as illustrated in Fig. 3 and Fig. 4. To outline dependencies across different microservices, our first step is to identify the invocation of a generated gRPC stub within a particular microservice. To accomplish this, we index each microservice to map out the corresponding call relationships.

There are two popular formats for semantic analysis. The LSIF, which heavily relies on edges to expose relationships, can add complexity and hamper performance when used for direct analysis. Given this, the SCIP is more apt for such analysis tasks for the following reasons:

- It holds extensive information, such as source location and reference relationships, in each symbol object, which signifies a declaration of a `variable`, `function`, or `type` in the source code.

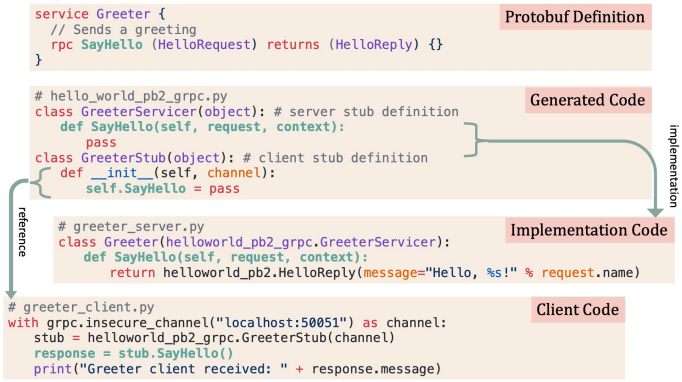

Fig. 3. Connection of Single Language Repository, $Conn$(client, generatedCode, reference) and $Conn$(server, generatedCode, implementation)

- It applies URI-based symbol name encoding for quick and accurate queries across different source files or even different projects.

Due to these benefits, we choose the SCIP and its corresponding indexers as the Intermediate Representation (IR) for the analysis process.

As depicted in Step 1 of Fig. 2, RPCover utilizes the appropriate SCIP indexer to analyze each microservice [36], establishing function-level dependencies within the service. To enhance clarity and comprehension of these indexes, we use the following formula to illustrate the structure of a single entry stored in the index.

$$Conn(node1, node2, relationship)$$

The variables $Conn$, $node1$, and $node2$ represent a connection and two different code stubs, respectively. The variable $relationship$ denotes the relationship between $node1$ and $node2$. For example, as shown in Fig. 3, *Conn(client, generatedCode, reference)* and *Conn(server, generatedCode, implementation)* represent relationships in Python using the provided client and server stubs. The file generated by the Protocol Buffer compiler includes both the server stub (which requires further developer implementation) and the client stub (which facilitates service invocation by the developer). The server is implemented in `greet_server.py`, while the corresponding service is invoked via the client stub in `greet_client.py`. After the indexing phase, we get two distinct relationships: the *implementation* relationship between server stub and its definition site, and the *reference* relationship between client stub and its call site. Through these connections between the generated code and its call or implementation sites from the SCIP index file, we can identify whether a microservice implements or calls a certain gRPC service, forming a foundation for analyzing dependencies between microservices.

B. Inter-Service Analysis

After the intra-service analysis, we obtain the gRPC server implementations and request calls in each microservice. However, the call relationships across microservices remain unclear. The gRPC request calls are still not associated with their implementations from one microservice to another. To fully uncover the gRPC dependencies, we incorporate an inter-service analysis. This step aims to extract cross-service call relationships, thereby connecting the separate call graphs of microservices together via these gRPC call relationships.

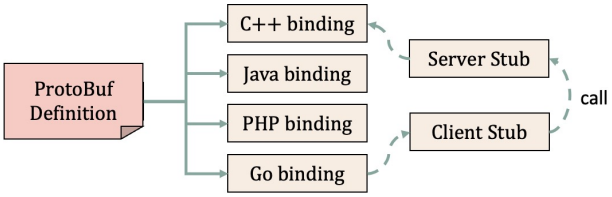

Fig. 4. Relationships between Protobuf Definition File and Language Bindings

For instance, as illustrated in Fig. 4, the Go binding functions might be used on the client side, and the C++ binding functions on the server side. This setup allows the Go client to invoke the C++ server. By applying the notation described earlier, we can express the relationship between the Go client and the C++ server. Our method would then establish the connection $Conn(generatedCode_{go}, generatedCode_{c++}, relationship)$.

For instance, as illustrated in Fig. 4, a microservice written in Go may call another microservice written in C++ via the generated Go bindings and client stubs. By the intra-service analysis, we have already established the following relationship within each microservice:

$$Conn(userCallSite_{go}, clientStub_{go}, reference) \quad (1)$$

$$Conn(userImplSite_{c++}, serverStub_{c++}, reference) \quad (2)$$

Inter-service analysis would establish the connection

$$Conn(clientStub_{go}, serverStub_{c++}, reference)$$

to associate the connection (1) with (2) and then form a complete dependency path from the Go microservice to the C++ microservice.

To construct the connection, it is necessary to characterize the stubs and the gRPC definitions, thereby constructing a relationship between them. We define an `index type` object to represent stubs in different languages. This object encapsulates the names of a declared type and its methods, along with code reference information. These `index type` instances are created in the *gRPC Matcher* when reading SCIP index files from SCIP indexers, as depicted in Step 2 of Fig. 2. Upon gathering the `index type` objects for all possible definitions in the source code of microservices, we use a *matcher* to identify possible associations between any pair of stubs and gRPC definition. Once two stubs from different microservices are associated with the same gRPC service definition, we can then conclude the connection between the two stubs as the example shown above. Algorithm 1 illustrates the process of building these associations, while Algorithm 2 shows an example of a fuzzy matcher. After establishing

Algorithm 1: Dependency recovery between Protobuf and SCIP index

Data: gRPC service Protobuf definition s_{proto}, index type t_{index}, and a matcher mt
Result: Updated index class t'_{index}, or MISMATCH

if $mt.MatchService(s_{proto}, t_{index})$ **then**
  $M \leftarrow$ A map that maps the method to its matched gRPC request definition
  for r_{proto} *in* s_{proto}.*Requests* **do**
    $m_{index} \leftarrow mt$.*FindAndMatchMethod*(r_{proto}, t_{index})
    if m_{index} *is nil* **then**
      return MISMATCH
    end
    $M[m_{scip}] \leftarrow r_{proto}$
  end
  $t' \leftarrow t_{index}$.*AddRelation*(s_{proto})
  for m_{index}, r_{proto} *in* M **do**
    m_{index}.*AddRelation*(r_{proto})
    $t'_{index} \leftarrow t'_{index}$.*UpdateMethod*($m_{index}$)
  end
  return t'_{index}
else
  return MISMATCH
end

Algorithm 2: Example fuzzy matcher

Data: gRPC service Protobuf definition s_{proto}, gRPC request Protobuf definition r_{proto}, and index type from SCIP t_{scip}
Result: $True$ or $False$ of whether the names in index follows the naming convention

def *convertName (n)*:
  $p \leftarrow$ Remove non-alphanumeric characters and convert string n to lowercase
  return p
def *FindAndMatchMethod* (r_{proto}, t_{scip}):
  for m_{scip} *in* t_{scip}.*Methods* **do**
    $n_r \leftarrow$ *convertName*(r_{proto}.*Name*)
    $n_m \leftarrow$ *convertName*(m_{scip}.*Name*)
    if n_r == n_m **then**
      return m_{scip}
    end
  end
  return nil
def *MatchService* (s_{proto}, t_{scip}):
  $n_s \leftarrow$ *convertName*(s_{proto}.*Name*)
  $n_t \leftarrow$ *convertName*(t_{scip}.*Name*)
  if n_t.*HasSubString*(n_s) **then**
    return *true*
  end
  return *false*

the connection, the gRPC Matcher unifies the SCIP index files from multiple microservices into one and embeds the newly constructed connections into the unified SCIP index. Nevertheless, the name normalization strategy might result in occasional false positives. For instance, RPCover will establish dependencies irrespective of the specific conditions required in actual client-server interactions. To address this limitation, a uniform interface is also provided for users to implement their own matchers. By defining customized matching algorithms, one can possibly reduce false positives from the default fuzzy matcher in a certain usage scenario.

The `index type` abstraction allows us to overlook language-specific details during dependency recovery. By identifying associations between gRPC stubs and definitions, and combining this with the results of intra-service dependencies from the previous step, RPCover forms the directional dependency from reference to implementation across microservices, thus revealing inter-microservice dependencies in the entire system. Furthermore, high-level features derived from the Protocol Buffer Compiler toolchains can be seamlessly integrated with RPCover. This includes functionalities such as the publisher-subscriber patterns, primarily because RPCover establishes relatively low-level connections between the Protobuf Definitions and client, and server stubs.

C. Index Conversion and Visualization

After following the steps mentioned above to establish all the relationships, we obtain a unified SCIP index file with the gRPC dependency information across microservices encoded. During this particular stage, the *LSIF converter* will convert the SCIP file into an LSIF file. The *LSIF converter* will discard redundant information to alleviate the burden on disk storage. This is necessary as the file consolidates all the preceding projects, and without this optimization, its size would become excessively large.

We utilize Memgraph to enhance the comprehension of dependencies. This platform visually illustrates the connections among various microservices. As shown in Fig. 5, it presents all the gRPC dependencies among the source files. In our benchmark, we set up 15 services each implementing a unique gRPC service, resulting in 15 distinct dependency clusters. Taking the `Go_A.proto` cluster as an example, its gRPC server establishes links with definitions and references distributed in 19 source files. These include one source file containing the implementation of `Go_A` service (`server.go`) and 18 generated source files from `Go_A.proto`, as illustrated in Table I.

IV. Benchmark

In this section, we introduce the limitations of the current benchmarks and show how we design and implement our benchmark.

Existing Benchmark: For our research, we investigated well-established microservice testbeds developed by external researchers, namely DeathStarBench [40] and TrainTicket [41]. DeathStarBench includes five end-to-end services, four for cloud systems, and one for cloud-edge systems running on drone swarms. TrainTicket consists of 41 Java-based microservices. These testbeds were chosen due to their widespread usage and their ability to represent diverse systems utilizing various components. The design methodologies employed by these testbeds offered valuable insights for related studies.

As the existing benchmark testbeds, DeathStarBench [40] and TrainTicket [41], primarily focus on one or two pro-

TABLE I
SEARCH RESULT OF Go_A'S RELATIONSHIPS

Index	File Name	Index	File Name
1	Go_A/cmd/server.go	11	Go_A/proto/Go_A_grpc.pb.go
2	Cpp_A/protos/Go_A.grpc.pb.h	12	Go_B/proto/Go_A_grpc.pb.go
3	Cpp_A/protos/Go_A.grpc.pb.cc	13	Go_C/proto/Go_A_grpc.pb.go
4	Cpp_B/protos/Go_A.grpc.pb.h	14	Java_A/src/main/benchmark/Go_AGrpc.java
5	Cpp_B/protos/Go_A.grpc.pb.cc	15	Java_B/src/main/benchmark/Go_AGrpc.java
6	Cpp_C/protos/Go_A.grpc.pb.h	16	Java_C/src/main/benchmark/Go_AGrpc.java
7	Cpp_C/protos/Go_A.grpc.pb.cc	17	Python_A/protos/Go_A_pb2_grpc.py
8	Ts_A/protos/Go_A.ts	18	Python_B/protos/Go_A_pb2_grpc.py
9	Ts_B/protos/Go_A.ts	19	Python_C/protos/Go_A_pb2_grpc.py
10	Ts_C/protos/Go_A.ts		

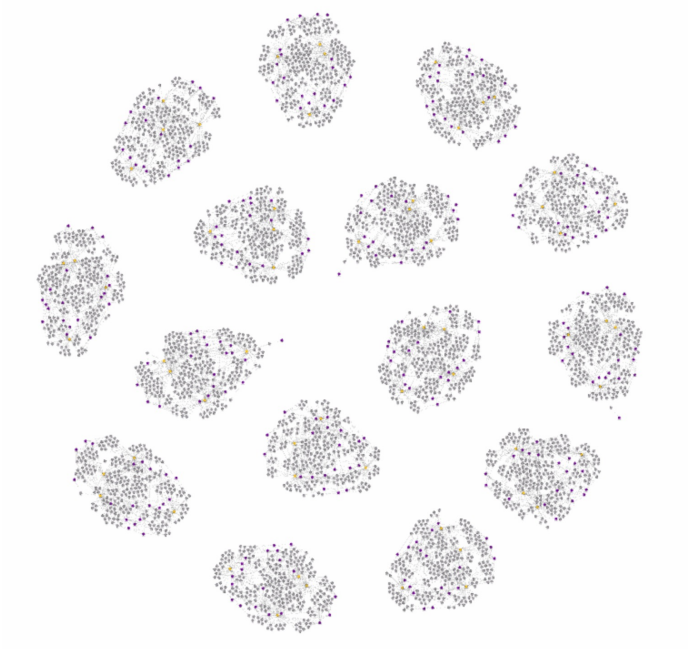

Fig. 5. Overview of benchmark visualization. Each cluster represents a gRPC Protocol Buffer definition and its related source files which have a reference between them.

gramming languages. Additionally, only DeathStarBench incorporates gRPC in its services. Hence, we have identified the necessity of developing our own benchmark to thoroughly evaluate our performance.

RPCoverBench: We want to benchmark RPCover in the following scenarios:

- Direct gRPC dependencies among microservices written in the same programming language
- Direct gRPC dependencies among microservices written in different programming languages
- Long dependency chains involving multiple microservices.

Moreover, we have considered the typical behavior of a service. Specifically, a service often functions as either a single client, a single server, or both.

In light of these requirements, we built a benchmark consisting of fifteen services across five programming languages to evaluate our approach in establishing interdependent multilingual services using gRPC. We selected the five most widely used backend programming languages (i.e., C++, Java, Python, Go, TypeScript) for the implementation of microservices. Additionally, we implemented three microservices for each language, as this is the minimum number necessary to create potential indirect dependencies within the microservices of the same language, with each service potentially acting as a client, server, or both.

This benchmark, which includes both typical and exceptional scenarios, enabled us to assess the efficiency and effectiveness of our approach. To be more intuitive, Fig. 6 represents the layout of our simplified benchmark, with each dashed square denoting a specific language's service implementation. Each square comprises two or three types of circles, with each circle representing an individual service.

- The green circle symbolizes the pure gRPC client, which solely invokes gRPC servers of other services.
- The red circle represents the pure gRPC server, which exclusively functions as a gRPC server, awaiting client requests.
- The yellow circle encompasses both scenarios, as it serves as both a client and a server simultaneously.

Each programming language provides three services, each of which comprises three functions. Each function calls several other functions, which can either belong to the same service or originate from external services. This is denoted in the *Dependent Services Function ID* column. The function IDs follow the format *[language]-[service name]-[function number]*. For instance, in Python service A, function `Python-A-1` would call `Python-A-2`, `Python-B-1`, and `Python-C-1`. For detailed information about the services, please refer to the benchmark repository [22].

V. EVALUATION

In this section we aim to answer the following research questions:

- **RQ1:** What is the coverage of the RPCover in finding the gRPC dependency?
- **RQ2:** How effective does the RPCover compared with the existing LSIF-based semantic dependency analyzers in terms of the execution time?

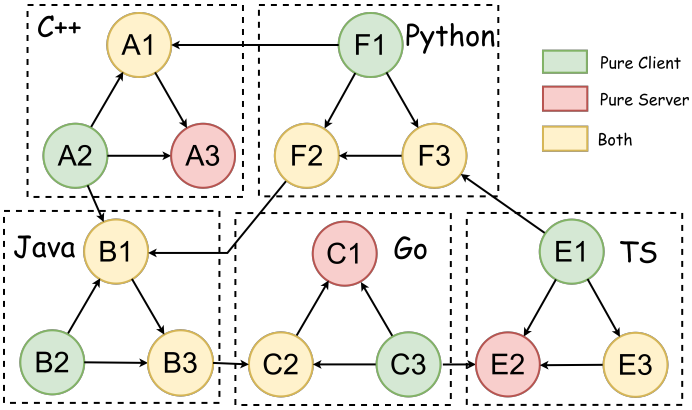

Fig. 6. Simplified Benchmark Layout

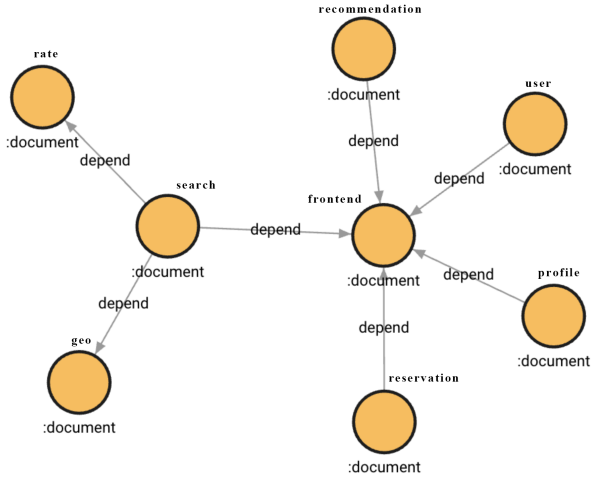

Fig. 7. SAR of DeathStarBench's hotel reservation services cluster

We ran the RPCover to analyze RPCoverbenchmark and the hotelReservation test from the DeathStarBench on a machine with 13th Gen Intel Core i5-13400; 126GB RAM; 1T SSD, running Ubuntu 22.04.

A. Result Analysis: coverage

In evaluating our approach, we consider both *precision* and *recall*. We manually count the gRPC calls in DeathStarBench [40] and our proposed benchmark RPCoverBench. As outlined in Table II, DeathStarBench comprises 7 gRPC calls, while RPCoverBench includes 142 gRPC calls. Using our method, RPCover identifies 7 function-level dependencies in DeathStarBench. Of these, 7 are correct, yielding a precision of 100%. This gives a recall of 100%. In the RPCoverBench evaluation, RPCover correctly identifies 142 function-level dependencies out of a total of 142 with 1 false-positive, resulting in a precision of 99.30%. The recall in this case is 100%.

In examining the misrecognized gRPC calls, we found that false positives arise when a struct shares the same name and function name as a service definition in a Protocol Buffer file. This leads RPCover to erroneously identify the struct as an implemented service, even though it is not. This case is unusual because the likelihood of writing a struct, which shares the same name and method names but has no relationship with the Protocol Buffer definition, is quite low.

To demonstrate our visualization results, we select the *Hotel Reservation* service from the DeathStarBench as an example. The outcome of the Software Architecture Reconstruction (SAR) is depicted in Fig. 7. The *Hotel Reservation* microservice cluster consists of seven sub-services: *user*, *profile*, *recommendation*, *reservation*, *search*, *rate*, *geo*, and *frontend*. Utilizing visualization platforms such as Memgraph or Neo4j, we can uncover the relationships between services within the cluster. For instance, the *search* service interacts with both the *rate* and *geo* services, whereas the *frontend* service makes calls to its related services.

B. Result Analysis: effectiveness

To answer RQ2, we evaluate the time and memory expenditure of RPCover across two benchmarks, taking into consideration both the programming language and the number of code lines.

Taking the single-language LSIF indexing with open-source indexers as the baseline, we conducted 20 runs of RPCover on each benchmark (i.e., DeathStarBench and RPCoverBench) and calculated the average cost. As presented in Table III and IV, the time overhead introduced by RPCover was found to be below 150%. The performance is primarily determined by the base SCIP indexer that is utilized, as the gRPC matching cost only accounts for 3-5% of the indexing task. Specifically, RPCover brings overhead only in C++ projects. The main reason for this large overhead is that C++ SCIP indexer does more work than the LSIF indexer. The C++ SCIP indexer analyses the source code in a more precise way with each translation unit considered, whereas the C++ LSIF indexer omits some of the external header files. In other languages, we even gain a performance boost using RPCover. The memory overhead brought by RPCover is less than 100%. Note that the baseline time cost and memory usage for the Java project is not applicable since we could not find a usable Java LSIF indexer. Currently, there are only two LSIF indexers for Java: `lsif-java` [42] developed by Sourcegraph has been deprecated, and `lsif-java` [43] developed by Microsoft can only run on Windows.

In addition, to further illustrate the performance impact brought by RPCover, we selected large-scale projects that employ gRPC and applied RPCover to them. Due to the lack of large open-source projects with microservice dependencies, we selected `grpc` and `grpc-go` with extensive Protocol Buffer dependencies (i.e., for example, grpc has 277 Protocol Buffer definition files), which require RPCover to match and construct dependencies between numerous symbols. The results show that the overhead brought by RPCover is also acceptable. The time overhead was approximately 175% for `grpc` and 22%

TABLE II
RQ1: RESULT

Benchmark Cluster	True Positive	True Negative	False Positive	False Negative	Precision	Recall
DeathStarBench	7	0	0	0	100%	100%
RPCoverBench	142	0	1	0	99.30%	100%
Overall	149	0	1	0	99.33%	100%

TABLE III
RQ2: TIME COST OVERHEAD

Repository	Language	Service	Lines of Code	Baseline Time(s)	Eval Time(s)	Overhead
DeathStarBench	Go	hotelReservation	4,503	4.53	3.46	-23.62%
RPCoverBench	Typescript	A	2492	3.3	1.98	-40.0%
		B	2486	3.26	2.003	-38.56%
		C	2475	3.86	2.009	-47.97%
	Python	A	1942	5.21	3.683	-29.32%
		B	1915	3.83	3.649	-4.73%
		C	1912	3.77	3.623	-3.9%
	Go	A	3795	5.56	3.484	-37.33%
		B	3786	5.01	3.425	-31.64%
		C	3656	4.14	3.416	-17.48%
	Java	A	8296	-	26.332	-
		B	8330	-	26.098	-
		C	8373	-	25.819	-
	C++	A	13953	23.55	47.684	102.48%
		B	13930	23.2	47.496	104.72%
		C	13922	23.5	47.382	101.62%
grpc	C++	-	934,936	1110.53	3062.09	175.73%
grpc-go	Go	-	85,890	15.39	11.90	22.68%

for `grpc-go`. Moreover, RPCover reduced memory usage to about 30% and 97% of the baseline on `grpc` and `grpc-go`, respectively.

VI. THREATS TO VALIDITY

A. Internal Validity

For internal validity, a suitable benchmark for our work does not currently exist. In particular, we need a multilingual benchmark where all services utilize gRPC. Consequently, we have to design and implement a multilingual benchmark evaluation framework that incorporates complex service call sequences. While this benchmark may not perfectly reflect real-world scenarios, as actual large-scale microservice systems could involve thousands of services with intricate dependencies, it is designed to emulate a variety of situations for comprehensive testing.

In our implementation, we can not completely eliminate the possibility of a false-positive when a user-defined type shares the same names as the service and method names in a gRPC definition. This occurrence is rare in practice, as it requires an exact match for both the type name and method names. However, even in these rare instances, our tool RPCover remains robust - it will not miss any gRPC calls that it should identify. Additionally, we provide an interface enabling users to define their own matching logic, catering to the needs of those using their own Protocol Buffer compiler plugins.

B. External Validity

Our approach has been implemented and evaluated for compatibility with the gRPC framework. However, it carries the potential for extending to other protocols that also utilize Protocol Buffers. Furthermore, in theory, our approach can be generalized to encompass all 11 programming languages supported by gRPC.

RPCover leverages the existing SCIP indexers to build the index for a specific repository. As a result, our approach's performance and reliability hinges on these external tools. It is worth noting that the tools employed in RPCover may contain certain known or unknown bugs. While we have addressed several bugs in our version, the possibility of yet-undiscovered issues remains.

VII. RELATED WORK

Multiple approaches aim to perform static analysis to recover code dependencies.

Microvision [14] leverages the Abstract Syntax Tree (AST) of the code to parse the codebase and identify endpoints. This enables Microvision to infer the complete structure of microservices. However, their approach only takes into account the same programming language. In contrast, our

TABLE IV
RQ2: MEMORY COST OVERHEAD

Repository	Language	Service	Lines of Code	Baseline Peak Memory(MB)	Eval Peak Memory(MB)	Overhead
DeathStarBench	Go	hotelReservation	4,503	521.6	546	4.68%
RPCoverBench	C++	A	13,953	1195	181.8	-84.79%
		B	13,930	1195	179.0	-85.02%
		C	13,922	1208	182.0	-84.93%
	Go	A	3,795	484	541.2	11.81%
		B	3,786	487	542.95	11.49%
		C	3,656	516	545.85	5.78%
	Java	A	8,296	-	1010.55	-
		B	8,330	-	1018.95	-
		C	8,373	-	1016.15	-
	Python	A	1,942	124	184.95	49.15%
		B	1,915	123	184.65	50.12%
		C	1,912	122	185.6	52.13%
	TypeScript	A	2,492	224	223.4	0.37%
		B	2,486	220	222.95	1.34%
		C	2,475	218	224.3	2.89%
grpc	C++	-	934,936	2,794	812	-70.93%
grpc-go	Go	-	85,890	1485	1444	-2.76%

evaluation considers five programming languages, and it is easy to extend, allowing for a more comprehensive analysis. LAAST [17] has the objective of converting language-specific code representations into a Language-Agnostic Abstract Syntax Tree (LAAST) to enable cross-language analysis of service repositories. However, the implementation of LAAST for multiple programming languages can be resource-intensive and expensive, posing a significant challenge in terms of practical adoption and scalability. In contrast, our approach offers ease of use and scalability through parallel execution and analysis.

Bushong et al. [18] utilize source code analysis to identify entities and bounded contexts within services, facilitating a better understanding of the system's architecture. However, their method is specifically designed for the Springboot framework. Granchelli et al. [19] utilize a combination of static and dynamic analysis, incorporating source code and Dockerfiles, to reconstruct the architecture of a system. On a similar note, Ibrahim et al. [20] leverage Dockerfiles to construct attack graphs and identify security vulnerabilities in container images. However, in our approach, we specifically concentrate on analyzing the calling dependencies within the code repository.

VIII. CONCLUSION

In this paper, we introduce RPCover, a novel approach for static service dependency recovery based on source code analysis. By leveraging the concept of analysis on top of a cross-language code indexing representation, RPCover enables the construction of a multi-lingual (including Protocol Buffer and other programming languages) and efficient dependency recovery tool for gRPC. This approach can easily be integrated into existing tools for further analysis, depending on the actual demand. We have also proposed a benchmark that includes five programming languages, with each language encompassing three services, which can be utilized to test RPCover's efficiency.

In our evaluation, we used our own proposed benchmark along with another benchmark, DeathStarBench. RPCover achieves an accuracy of 99.33% in identifying service relationships, while maintaining an overhead of less than 200% of the existing LSIF indexers across all benchmarks.

ACKNOWLEDGMENTS

This paper was supported by the National Natural Science Foundation of China (No. 62102340), and Shenzhen Science and Technology Program.

REFERENCES

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

[2] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[3] S. S. de Toledo, A. Martini, A. Przybyszewska, and D. I. Sjøberg, "Architectural technical debt in microservices: a case study in a large company," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 78–87.

[4] M. E. Gortney, P. E. Harris, T. Cerny, A. Al Maruf, M. Bures, D. Taibi, and P. Tisnovsky, "Visualizing microservice architecture in the dynamic perspective: A systematic mapping study," *IEEE Access*, 2022.

[5] Y. Zuo, X. Zhu, J. Qin, and W. Yao, "Temporal relations extraction and analysis of log events for micro-service framework," in *2021 40th Chinese Control Conference (CCC)*. IEEE, 2021, pp. 3391–3396.

[6] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, pp. 1–15, 2021.

[7] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, pp. 1–28, 2022.

[8] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," in *Information Science and Applications: Proceedings of ICISA 2020*. Springer, 2021, pp. 223–234.

[9] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1387–1397.

[10] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[11] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3901–3914, 2022.

[12] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *Journal of Systems and Software*, vol. 159, p. 110432, 2020.

[13] R. Heinrich, "Architectural run-time models for performance and privacy analysis in dynamic cloud applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 4, pp. 13–22, 2016.

[14] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microvision: Static analysis-based approach to visualizing microservices in augmented reality," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 49–58.

[15] ——, "Microservice architecture reconstruction and visualization techniques: A review," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 39–48.

[16] B. Mayer and R. Weinreich, "An approach to extract the architecture of microservice-based software systems," in *2018 IEEE symposium on service-oriented system engineering (SOSE)*. IEEE, 2018, pp. 21–30.

[17] M. Schiewe, J. Curtis, V. Bushong, and T. Cerny, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, vol. 10, pp. 30 743–30 761, 2022.

[18] V. Bushong, D. Das, A. Al Maruf, and T. Cerny, "Using static analysis to address microservice architecture reconstruction," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1199–1201.

[19] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *2017 IEEE International conference on software architecture workshops (ICSAW)*. IEEE, 2017, pp. 46–53.

[20] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," in *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, 2019, pp. 1235–1242.

[21] A. Fang and R. Zhou. (2023) protoc-gen-scip. [Online]. Available: https://github.com/CUHK-SE-Group/protoc-gen-scip

[22] ——. (2023) Rpcoverbenchmark. [Online]. Available: https://github.com/CUHK-SE-Group/RPCoverBenchmark

[23] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[24] H. Vural, M. Koyuncu, and S. Guney, "A systematic literature review on microservices," in *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part VI 17*. Springer, 2017, pp. 203–217.

[25] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[26] D. Guamán, J. Pérez, J. Diaz, and C. E. Cuesta, "Towards a reference process for software architecture reconstruction," *IET Software*, vol. 14, no. 6, pp. 592–606, 2020.

[27] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[28] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5, pp. 102–113, 1989.

[29] B. J. Nelson, *Remote procedure call*. Carnegie Mellon University, 1981.

[30] J. Chen, Y. Wu, S. Lin, Y. Xu, X. Kong, T. Anderson, M. Lentz, X. Yang, and D. Zhuo, "Remote procedure call as a managed system service," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 141–159.

[31] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "{CRISP}: Critical path analysis of {Large-Scale} microservice architectures," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 655–672.

[32] P. B. Team. (2023) Protocal buffers documentation. [Online]. Available: https://protobuf.dev/overview/

[33] ——. (2023) Who uses protocol buffers. [Online]. Available: https://protobuf.dev/overview/#who-uses

[34] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*. IEEE, 2016, pp. 261–265.

[35] S. Team. (2023) Lsif community. [Online]. Available: https://lsif.dev/

[36] ——. (2023) Scip code intelligence protocol. [Online]. Available: https://github.com/sourcegraph/scip

[37] ——. (2023) Sourcegraph. [Online]. Available: https://github.com/sourcegraph/sourcegraph

[38] Ólafur Páll Geirsson. (2022) Scip - a better code indexing format than lsif. [Online]. Available: https://about.sourcegraph.com/blog/announcing-scip

[39] M. Team. (2023) Defines a common protocol for language servers. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsif/0.6.0/specification/

[40] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[41] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 323–324.

[42] S. Team. (2023) Scip code intelligence protocol (lsif) generator for java. [Online]. Available: https://github.com/sourcegraph/scip-java

[43] jdneo. (2023) Language server indexing format implementation for java. [Online]. Available: https://github.com/microsoft/lsif-java