Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image

Richard Hutcheson
Computer Science
ECS, Baylor University
Waco, United States
0009-0001-5469-2730

Austin Blanchard
Computer Science
ECS, Baylor University
Waco, United States
0009-0008-3678-6988

Noah Lambaria
Computer Science
ECS, Baylor University
Waco, USA
0009-0009-8154-2664

Jack Hale
Computer Science
ECS, Baylor University
San Francisco, California
0009-0003-7303-0278

David Kozak
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
0000-0002-8846-922X

Amr S. Abdelfattah
Computer Science
ECS, Baylor University
Waco, USA
0000-0001-7702-0059

Tomas Cerny
Systems and Industrial Engineering
University of Arizona
Tucson, Arizona
0000-0002-5882-5502

***Abstract*—Microservices are the mainstream architecture when designing cloud-native systems. The performance and elastic scalability of such systems are the main attraction for many vendors. Recent advancements improving microservice initialization times are related to the ahead-of-time compilation, which produces self-contained executables, significantly reducing load times. Despite recent advancements and various benefits of cloud-native systems, the evolution of such systems might be threatened by a missing system-centered view. Such a view would guide in a better contextual understanding of individual microservices and their dependencies from the holistic system perspective and aid developers in informed decisions to mitigate ripple effects. One way literature has addressed this gap is by performing Software Architecture Reconstruction (SAR), a process essential for understanding, maintaining, and evolving software systems. This paper questions whether instruments used to produce self-contained executables for microservices can be utilized for SAR, producing system-centered views. We propose a methodology for such a process, implement a proof of concept tool, MicroGraal, for the Java Platform, and assess it through a case study involving a third-party microservice system benchmark. We uncovered a system service dependency graph and a context map, comparing the approach and obtained results with source code analysis.**

***Index Terms*—Software Architecture Reconstruction, Microservices, Service Dependency Graph, GraalVM Native Image**

I. Introduction

Using microservice architecture has been the de-facto standard approach in the industry for the last decade to build cloud-native systems. This architecture provides considerable advantages to developers, including *flexibility*, *scalability*, and *facilitated deployment*, as discrete parts of the system can be independently designed, developed, and deployed.

However, developing and evolving microservice systems are also known to be *complex* and *error-prone* [1]. The distributed system nature lacks a holistic perspective [2]. This statement is true both for a static code analysis approach and for the developers contributing to the system, both of which typically analyze only a single microservice at a time. We contend that, in the era of microservices, traditional static code analysis methods are inadequate, as the majority of issues arise from the interactions between distinct microservices within the system and thus remain undetected by these conventional approaches.

To address this challenge, researchers have proposed Software Architecture Reconstruction (SAR) methods specifically tailored for microservices [3, 4, 5, 6, 7]. SAR techniques for microservices facilitate a deeper understanding of the system by generating high-level architectural *views* [8] that concentrate on distinct facets, such as the *service view*, describing the interaction among microservices and the *domain view* which illustrates involved domain data. The SAR process is also a necessary precondition for automated system assessments, such as detecting microservice smells (i.e., cyclic dependencies, etc.) apparent from the system's holistic perspective.

The manual execution of SAR [3, 5] is both time-consuming and susceptible to errors. Consequently, researchers have proposed methodologies that employ static analysis of the source code to address these challenges. However, there are instances where developers may not have access to the source code, such as if there are legal constraints or if the analysis is conducted during deployment. Fortunately, the compiled representation of numerous programming languages, including Java bytecode, retains sufficient information to facilitate static analysis.

A successful SAR approach for microservices should accommodate recent advancements in cloud-native infrastructure, where faster startup times and reduced memory footprints are demanded. Specifically, there is a surge in the popularity of ahead-of-time (AOT) compilation. GraalVM [9] introduced a component: Native Image [10], which is a compiler for Java bytecode that uses a combination of points-to analysis, AOT compilation, and class initialization at build time to create self-contained binaries which start quickly and execute with a lower memory footprint compared to running on the Java Virtual Machine (JVM). These capabilities make it a perfect prospective for cloud environments. Moreover, contrary to the just-in-time (JIT) compilation in a classical JVM, the AOT compilation model of GraalVM Native Image gives the possibility of creating custom static analyzers on top of the compilation pipeline. GraalVM Native Image has been getting a lot of attention in the industry lately: main Java microservice frameworks including Spring [11], Micronaut [12], Quakus [13], and Helidon [14] all invested into developing a first-class support making it easier to compile microservices written in these frameworks into native images.

This paper considers the SAR of microservice systems by utilizing Native Images of particular microservices. By developing an analyzer on the foundation of a widely adopted, industry-ready compiler, we can provide added value to the Native Image community without necessitating the integration of additional tools into their pipelines. On top of that, using Native Image allows us to base the analysis on Java bytecode, covering the case when source code is not available, e.g. for legal reasons. In addition to such advancements, we demonstrate that information extracted from Native Images can be utilized by human experts through an interactive visual perspective for the service view and domain view. These two views combine to provide a holistic view of the system.

In summary, this paper contributes the following:

- It introduces a novel methodology for a static analysis-based SAR for microservice systems using GraalVM Native Image to extract the *service* and *domain views*. Including the analysis in a compiler allows its execution without the need for additional tools. Moreover, the analysis is based purely on Java bytecode, covering the case when source code is not available, e.g. for legal reasons.
- It produces a proof-of-concept tool MicroGraal, targeting SAR for Java-bases microservices, interactively visualizing two architectural views to aid developers in understanding their systems.
- It demonstrates the approach through a case study performed on a well-established microservice benchmark.

This paper is organized as follows. Section II provides background to related notions. Section III provides a related work overview. Section IV details our methodology. Section V provides a case study. Section VI provides discussion. In Section VII, we discuss threads to validity, and, finally, we conclude in Section VIII.

II. Background

This section briefly introduces the SAR process, static analysis, and GraalVM.

A. Software Architecture Reconstruction

Software Architecture Reconstruction has been well detailed by O'Brien [15] as "the process by which the architecture of an implemented system is obtained from the existing system". It is meant for evaluating the conformance of the `as-built` to the `as-documented` architecture, reconstructing documentation, and analysis and comprehension of the system architecture. SAR enables modifications of the architecture to satisfy new requirements and eliminate existing software deficiencies.

The reconstruction process aims to uncover particular architectural viewpoints [3, 8] that frame stakeholder concerns about an entity of interest. Such viewpoints govern one or more architectural views comprising a portion of an architecture description [8]. Among the example views for microservices [3, 4] are domain concerns describing the entities of the system along with the data sources; the system's implementation and operation technology aspects; service operators describing the service models that specify microservices, interfaces, and endpoints (i.e., service view realized as service dependency graph); and the operation focusing on service deployment and infrastructure, such as containerization, service discovery, and monitoring.

The construction process has four phases [4]. It first aims to gather the necessary artifacts that serve as information input to the process, which might be relevant to the particular perspective or view of the system. Next, we construct the canonical representation of the perspective. We form an initial intermediate representation and follow to the next phase to combine particular perspectives to reconstruct more holistic architectural details. Finally, this holistic detail is an instrument serving for analysis to provide insight about a system. These questions can relate to various concerns, including domain models, dependencies, interaction, design quality, privacy, or security aspects of the system.

B. Static Analysis

Static analysis is typically performed on program source code to inspect it without execution. The analysis process provides an understanding of the code structure to ensure that the program code and its design follow expected quality or standards. It can be involved throughout software development to detect common errors or poor coding practices. Typically, code quality tools use static analysis to build program intermediate representation, which they use for pattern marching. These patterns typically represent an anti-pattern associated with common design errors or poor coding practices.

Static analysis can operate on source code or even the bytecode or binary. Most commonly, for source code input, we use parsers that produce program graph representations. For example, Abstract Syntax Trees (AST) [16], Control-Flow Graphs (CFG), and Program Dependency Graphs (PDG) [17].

Companies dealing with software security testing wade into the "no source available" pool, where they might be looking for an alternative input to source code. Bytecode is compiled, high-level, machine-independent code that is meant to run on a virtual machine, such as the Java VM or the .NET CLR. For instance, Veracode[1] claims their software bytecode analysis has no lossy intermediate step back to the source code. It is possible to decompile bytecode back to the source code to run source code analysis. However, typically, when we have access to source code, we also have access to the codebase, which contains build, deployment (i.e., docker files), and configuration files not necessarily available in bytecode analysis.

However, some platforms do not compile into a bytecode and instead produce a binary. It can still be analyzed but with difficulty while uncovering less detail about the original code structures. The process is also known as binary analysis.

Most current static analysis approaches remain distant from microservices, as they consider a single program or codebase. On top of that, microservices build on well-established standards encapsulated through components, while static analysis typically looks at the low-level language constructs despite the current higher-level programming practices in various cloud-based or enterprise-based development frameworks. Thus, to properly and holistically perform static analysis for microservice systems, the analysis needs to recognize higher-level programming constructs such as components, endpoints, remote calls, etc.

C. GraalVM

GraalVM is a Java Virtual Machine (JVM) that uses the Graal compiler [18] for just-in-time (JIT) compilation. Graal is a compiler for Java written in Java that was designed to be extensible and maintainable, thus making it easier to develop complex optimizations and to access and comprehend the source code of the compiler using modern, integrated development environments. On top of that, GraalVM also provides the Truffle Framework [19] using which one can create and integrate other language runtimes for GraalVM, such as Python, Javascript, and R [20].

1) Graal Intermediate Representation: Graal Intermediate Representation (IR) [21] is graph-based and models both the control-flow and the data-flow dependencies between nodes. The IR for a given method is generated by parsing its bytecode. It is in static single-assignment (SSA) form [17], i.e., all values have unique static definitions. Using the SSA form speeds up many compiler optimizations, such as constant propagation or dead code elimination. The nodes in the IR are separated into two groups: *fixed* and *floating*. Fixed nodes have a strict ordering based on the control flow of the program. Floating nodes represent values; they float around the fixed nodes and are only loosely coupled to them. This design makes it easier to apply optimizations such as global value numbering [22].

Consider the structure of Graal IR using the example code snippet in Listing 1. The method `abs` computes the absolute value of the parameter `x`. The corresponding Graal IR of the method can be found in Figure 1. The control flow of the program is denoted by the red edges connecting `fixed` nodes and data flow is denoted by the blue edges connecting `floating` nodes. The `if` node is connected with the condition < and has two output branches. Notice the ssa `phi` node merging the two values of `res` coming from different branches. It is associated with the `merge` node and has two input branches corresponding to the two control flow paths that are merged there.

```java
void abs(int x) {
    int res;
    if (x >= 0) {
        res = x;
    } else {
        res = -x;
    }
    return res;
}
```

Listing 1: Code snippet sample illustrated as Graal IR in Fig. 1

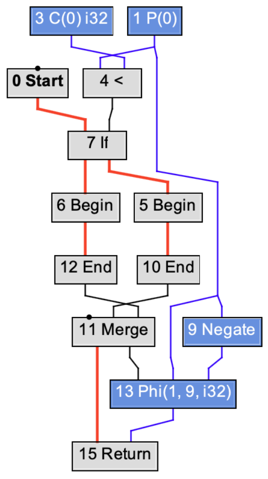

Fig. 1: Visualized Graal IR

We base the analysis on the Graal IR as it is more high-level compared to accessing the bytecode directly, and we can leverage the whole infrastructure of the compiler. However, our methodology is not limited to GraalVM and Graal IR, alternatives such as ASTs could be used as well.

2) GraalVM Native Image (NI) [10]: A Java bytecode compiler that uses a combination of points-to analysis, ahead-of-time (AOT) compilation, and heap snapshotting to compile applications into standalone binaries that have a significantly faster startup time and lower memory footprint compared to running on the JVM. NI works under the *closed-world assumption*, i.e., all application classes must be accessible during the compilation. While restrictive, this assumption allows more aggressive optimizations. Dynamic features of Java, such as reflection, have to be explicitly registered [23].

GraalVM Native Image has been lately getting a lot of attention in the industry thanks to the benefits it can bring for Java microservice applications deployed in the cloud. Main Java microservice frameworks, including Spring [11], Micronaut [12], Quakus [13], and Helidon [14] developed first-class support making it easier to compile microservices written in these frameworks into native images. All these frameworks provide support that automatically generate the necessary configuration. Therefore, even Spring applications that are known to use reflection heavily can be compiled via Native Image. On top of that, the AOT compilation model of GraalVM Native Image allows us to integrate our analysis into the compilation pipeline, which would be more complex with a classical JVM.

III. Related Work

Several studies have anticipated System Architecture Reconstruction (SAR) for microservice systems, but they differ

[1]Veracode: https://www.veracode.com, accessed on 05/05/2023.

in their approaches. Some methods leverage the artifacts produced during runtime to perform dynamic analysis, as proposed by Al Maruf et al. [7]. They analyze telemetry data to identify inter-service communication patterns and construct a Service Dependency Graph (SDG), which is then used to detect architectural smells in the system. However, these approaches do not support the context map of the system, and building the SDG from logged traces requires the system to be operational.

Therefore, other approaches use certain artifacts that are generated and available during the development phase to implement hybrid analysis approaches. For instance, Mayer et al. [6] combined static and dynamic analysis techniques to extract the architecture of REST-based microservice systems. They extracted static information using Swagger documentation to generate API descriptions of services and analyzed dynamic data from log files, including incoming and outgoing requests of each service instance.

Other approaches have taken a different direction and focused on manual analysis [3, 5] to build various models to combine. Another alternative was to use static analysis. Cerny et al. [24] and Walker et al. [4] examined the source code of microservice-based systems. However, their method used Java reflection APIs, which limited their ability to only analyze Java-implemented systems. Likely, Das et al. [25] analyzed both source code and bytecode for Java-based systems to construct an SDG, which they then used to detect potential RBAC violations. Although these methods successfully reconstructed the architecture of systems, they are limited to specific machines and languages, which restricts their potential use in heterogeneous microservice systems.

Diving deeper into IR-based approaches, Schiewe et al. [26] proposed a static analysis technique, Relative Static Structure Analyzers (ReSSA), that shows great potential to transform static code analysis practices. Their approach operates with component types instead of low-level programming constructs, constructing and utilizing an IR called Language-Agnostic Abstract-Syntax Tree (LAAST). LAAST representation is constructed from the source code's AST. ReSSA introduced a set of generalized parsers to detect specific component types. These parsers can be system-specific parsers to better cope with platform differences. The study demonstrated a unified identification approach to determine system data entities and endpoints for constructing SDG and the context map from microservice-based systems.

Our proposed approach shares similarities with ReSSA's approach in utilizing an IR, but it has some significant differences. ReSSA's technique requires access to the source code of the system for analysis, which may not always be feasible, particularly for support teams that only have access to the deployable bytecode of the system. In contrast, our approach operates over the Graal IR, which is constructed from the bytecode. Additionally, ReSSA employs LAAST, a proprietary intermediate representation that lacks sufficient support and requires considerable effort to support other languages and fix issues. For example, we are not aware of any visualization tool for ReSSA similar to IGV for Graal IR. As they mentioned that the user is also responsible for handling numerous edge cases in the structure, such as the various possible ways the endpoint URL could be defined in the call. Our proposed approach employs Graal IR, which has the potential for industry usage and support. Graal IR also captures semantic properties of the system like data-flow dependencies between nodes, not just the syntax, which makes it clearer and more promising for extracting information in heterogeneous microservice systems. Finally, ReSSA necessitates several parsers and specifications to extract the required information.

IV. Methodology

For our microservice-aware SAR process, we follow the assumption of "no source" available and use bytecode analysis via Graal IR as the input. Assumptions are made that microservices use best practice design and are developed using well-established frameworks that make use of components and high-level design constructs. These are provided by current frameworks, as they facilitate faster development. These assumptions enable generalization when performing SAR, and can simplify the process to matching components and high-level constructs and their properties.

There are three phases in our process: (1) analyze the Graal IR for every single microservice and detect high-level constructs, (2) using the high-level constructs, reason about the inter-microservice dependencies and build IR for two architectural views (3) visualize the particular architectural perspectives of the system. Figure 2 illustrates the process phases.

In phase 1, to analyze the bytecode of a single microservice via Graal IR, the Graal API is used to parse particular components and high-level constructs that are commonly used across development frameworks. In particular, components like the `entities`, `services`, and `controllers` are recognized and extracted with their details (i.e., `endpoints`, attributes, etc.). It is also necessary to parse `remote REST calls` to other microservices within the system, which are necessary to detect dependencies between microservices. Section IV-A gives more details.

In phase 2, the data extracted from phase 1 are transformed for the creation of the service dependency graph and context map, which represent the service view and the domain view. In the construction of the service dependency graph, REST calls are bound to endpoints via partial signature matching.

In order to provide a more clear representation of the domain view of the system, entities across microservices are combined based on their similarity in names and/or attributes. It is common for microservices to share entity variations across their microservice bounded context. By combining them, we greatly clarify the structure of the system. More details on how the data is merged are given in Section IV-B.

Finally, in the third phase, the constructed architectural perspective IRs are used for visualizing the constructed architectural perspectives. Our focus is the *service view* (service dependency graph) and the *domain view* (context map formed from individual entities of bounded contexts). The service

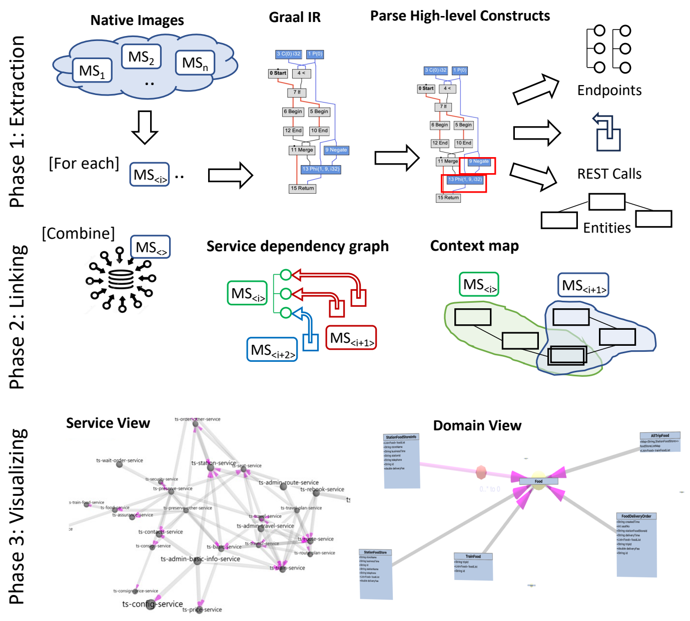

Fig. 2: Illustration of the phases in our methodology where each microservice's Native Image is processed to detect high-level constructs in the Graal IR and then connections across microservices are linked to derive intermediate representations of architectural views, which are then visualized.

dependency graph illustrates communication across microservices. It enables users to obtain detailed insights into the system without examining the code directly. It also illustrates dependencies and potential architectural change propagation. We detail this in Section IV-C.

Our methodology provides a comprehensive solution for analyzing and visualizing complex microservice systems via GraalVM Native Image in order to enable a high-level understanding of a system without delving into the codebase. As a result, informed decision-making and system management can be accomplished at a high-level abstraction of the system.

A proof-of-concept of our methodology has been implemented for practical assessments. The resulting MicroGraal tool allows us to assess the concept feasibility for microservices implemented using the Java Platform. It also serves as a reference implementation for SAR on GraalVM Native Image and will enable replicating our findings in this work. The open-source is shared with the community. The existence of the MicroGraal tool allows us to illustrate examples in each next sub-section that details the phases of our methodology.

A. *Extraction of Data*

To extract the data necessary to derive particular architectural perspectives - the service and domain views - we parse the bytecode of each microservice in the system using Native Image to generate the Graal IR which serves as an input to the analysis. We first assume that the two views are represented by components of development frameworks. Then we analyze each class in the microservice to identify such components and other high-level constructs. This allows our methodology to be reused across different systems.

To aid understanding of the Graal IR, Oracle developed the Ideal Graph Visualizer (IGV)[2] (see Figure 1), which can help with matching nodes necessary to identify high-level construct' patterns in Graal IR. In the following text, we introduce the extraction of important high-level constructs from Graal IR. In particular, we discuss how to extract REST calls, endpoints, and data entities.

1) REST Call Extraction: To extract REST calls, it was necessary to identify their patterns in the Graal IR. The process to identify these patterns involves inspecting the Graal IR in IGV, comparing it with source code examples, and determining which types of nodes contain specific information based on the code structure. Graal IR can exhibit a virtually infinite range of structures due to the unique composition of each method and its contents, as well as the inherently complex mapping from source code to compiler-level representations. This complexity arises from the diverse ways programming constructs can be expressed, the presence of various language features, and the compiler's role in optimizing and transforming the code particularly in the lexical, syntactical, and semantic analysis stages of the compilation process. Consequently, the methodology developed in this research focuses on capturing the most prevalent patterns of REST calls encountered in the research while providing the flexibility for future researchers to extend the approach to additional patterns. Note that while identifying the patterns is a manual process, it is done only during the development of the analysis. The actual analysis execution is automated.

Given the Java specifics, the process begins by retrieving a list of class objects from the microservice's `ImageClassLoader`, which is then filtered to only include class objects belonging to the microservice's base package. From there, every class is broken down into its declared methods. For every method, a `StructuredGraph` containing its IR can be created. Then the IR was traversed to locate the node which contains the `RestTemplate` object commonly used for REST calls.

The patterns we developed assume there is only one `RestTemplate` call in the method. Querying the matched nodes, one can extract details, such as the URL of the HTTP request, the type of HTTP request (`GET`, `PUT`, etc.), the return type of the REST call as well as if it were a collection of that type, and what combination of class and method the call was made in. The details of each REST call are stored in a custom `RestCall` object respectively and appended to the list of REST calls for that given microservice for the Graal analysis system to handle.

2) Endpoint Extraction: Graal's Meta and Compiler APIs provide extensive utilities for extracting endpoints from controllers. An `AnalysisMethod` object, which can be traversed for extracting endpoint data, was used for fetching

[2]IGV: https://docs.oracle.com/en/graalvm/enterprise/20/docs/tools/igv/, accessed on 05/05/2023.

HTTP requests, return types, and parameters. To capture the return type for an endpoint, the Reflection API was adopted as an intermediate step between Graal IR and our custom objects. For other required attributes, several conditions were handled based on Spring's annotations for mapping. For example, the `RequestMapping` annotation required additional implementation to ensure the target endpoint path was accurate. Upon GraalVM compilation, a single CSV file containing each controller's endpoints for a provided microservice is generated.

3) Entity Extraction: To extract the entities from a given microservice, each class, along with the methods and the fields for that class, is considered. The class, fields, and methods are all extracted from their internal representation used by the Graal compiler. In the case of many Java frameworks, the classes, methods, and fields can be checked for annotations which indicate an entity along with its details. This various annotations coming from the Java Persistence API, Spring, Lombok, or other frameworks can be considered along with methods generated which are consistent with an entity class. In some cases, the methods must be checked because annotations have been processed during compilation before Graal's IR analysis begins. Once an entity class has been identified, the entity name as well as each field, including the field name, the type, and all annotations with particular settings, are extracted into an intermediate representation for a given microservice; we use JSON format for this representation.

When the analysis of a microservice is finished and the extraction of information is complete, a CSV file containing all the extracted REST calls and their details is created, and the same is created for endpoints. A third file is also generated - a JSON file with all entities extricated as well as their respective fields and annotations. These files are read by the phase 2 for the creation of domain and service views.

B. Transformation of Data

Using the high-level structures, constructs, components, and their contextual details extracted using GraalVM Native Image, we next pay attention to the relationships between these extracts. The aim of this consideration is to form particular system views such as the domain and the service view. In particular, we construct the context map and service dependency graphs which serve as illustrations of these views. For the transformation of the data process, users must supply a JSON list of microservices, which for every microservice includes the microservice name, base package, and base directory. As discussed in further sections, our utilized microservice benchmark was manually input into our JSON file. Note that the necessity to manually supply this JSON file is purely an artifact of our current prototype. This step could be automated in the future, for example by interacting with the build system to fetch the data.

1) Service Dependency Graph: Extracting REST calls from each microservice includes the URL, HTTP method, destination microservice (if present in the URL), and if there is a body parameter. Endpoint parsing will have subsequent information such as the HTTP method, the URL, and parameter types. Therefore the linking can be achieved using the URL, HTTP method, microservice name (if it can be parsed from the REST Call), and if there is a body parameter to match the REST call to the endpoint in the microservice. One thing to note is that the URL is matched from a REST call to an endpoint without the use of path parameters. This is because of the limitation of finding the names of the path parameters from the REST Calls. Therefore, the matching of URLs only match the hard-coded parts of the URL and deletes the path variables. The URL is a complete match only on the hard-coded sections of the string.

After the linking process completes, the service dependency graph is complete and stored as a JSON with all the microservices listed as nodes, and all REST call/endpoint pairs placed within their respective links for the visualizer to read.

2) Context Map: In order to transform the extracted data entities into a more comprehensive perspective, bounded context data models are formed and used to derive a system context map. Extracted entities are associated according to relationships to form data models within each microservice. In addition, inter-service entity models are merged based on entity similarities.

The process considers every microservice we point to via the configuration file that phase 2 requires. In the case of Java, each microservice is packaged as a Java Archive that has a well-defined structure of compiled classes libraries, and resources, which needs to be accessed when converting to the Graal IR and analyzing it (i.e., `/BOOT-INF` folder).

The entities from every microservice are extracted and collected to serve as references for possible parts of a relationship between entities. Every microservice is iterated over again, extracting the fields from every entity within a microservice. These fields are compared against the list of entities extracted previously to check for a type match, which indicates a relationship between the two entities. If there is a match, the multiplicity of the relationship is recorded based on the number of fields and if those fields are a collection. Furthermore, the relationships extracted are one-sided (A to B is different from B to A) and thus have to be combined to form a full representation of the multiplicities of that relationship.

The following step is to merge data models across microservices to create a context map for the entire system. Since multiple microservices may operate in the same domain, some bounded contexts may contain the same entities. Additionally, different bounded contexts may have different purposes for the entities they share, which means they may retain different fields from each other. To accomplish this, the following merging rules are applied to the entities and their fields:

- Entities are merged by determining if they have the same or similar names. We utilize the WordNet project [27] to detect similarities in names.
- Fields that have the same data type and the same or similar names in the merged entity are merged.
- Non-matching fields from both entities can be appended to the merged entity.

Finally, the list of merged entities and their relationships is used to create a JSON schema to represent an intermediate representation of the context map.

C. *Visualization of Data*

The architectural view IRs of derived views would not serve much purpose unless constructed information becomes easily accessible. For this reason, we consider interactive visualizations that provide capabilities for reconstructing a service dependency graph and context map.

1) Service Dependency Graph: With the formed service dependency graph intermediate representation, it is possible to approach visualization. A collection of nodes, node labels, and directed links represent this graph. The nodes represent individual microservices, and the labels above each node display the name of the microservice.

The users may interact and select each node and open a window that displays information about the microservice, such as that microservice's dependents and dependencies. The dependents of, for example, microservice A, are any microservices that make calls to microservice A. The dependencies of A would be any microservice that A makes outbound calls to. Microservices that make no calls and receive no calls to other microservices have no links and thus float around the graph.

The directed links are a visual representation that show the direction of the REST call from source to target. When the user clicks on a link, a window pops up that lists the source microservice, the target microservice, and all the REST calls that occur from source to target.

The application also allows users to grab nodes and drag them around the three-dimensional (3D) space, as well as pan and rotate the camera to better orient themselves and view the graph from different perspectives.

A snapshot of service dependency graph reconstruction is visible in Figure 3.

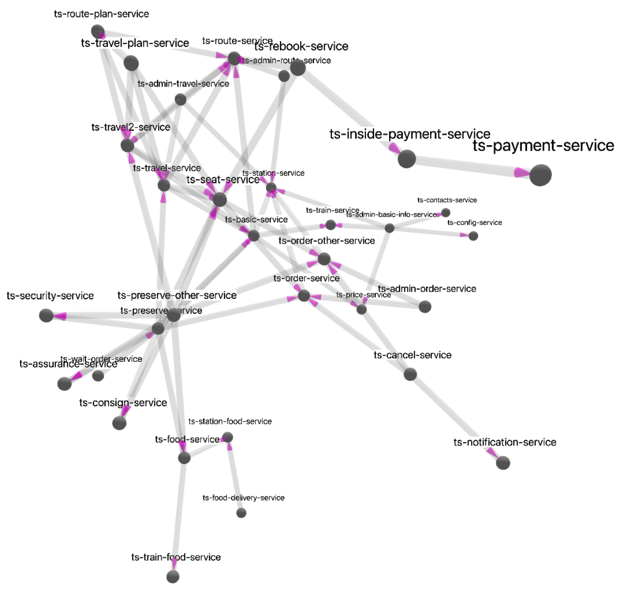

Fig. 3: Service Dependency Graph for Train-ticket benchmark

2) Context and Sub-Context Maps: Based on the analysis of data entities, the JSON-based context map is visualized in a web-accessible (react-force-graph-3d and three.js) format. Once processed, all context map entities are displayed as `CSS3DObject` nodes in a 3D, customized with each entity's associated fields and types. Links between nodes are contextual and hoverable with the cursor, revealing the multiplicities between different entities.

Our interactive visualization allows for creating sub-context maps by selecting certain microservices. This assists practitioners in understanding how a microservice's direct neighbors and data model overlap with dependencies. The this feature is beneficial for use cases where a provided system contains a significant number of entities and microservices. An example of a sub-context map is depicted within Figure 4.

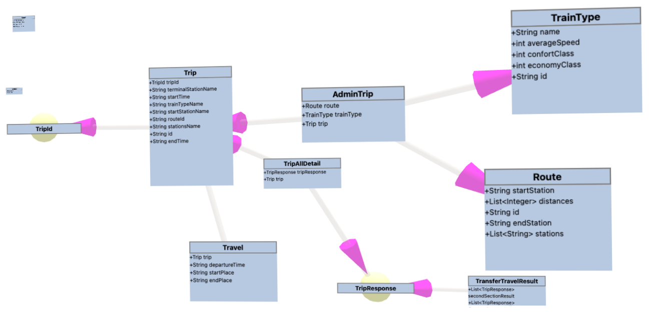

Fig. 4: Sub-Context map for Train-ticket benchmark

Included Microservices: ts-travel-service, ts-travel2-service, ts-travel-plan-service, ts-common.

V. Case Study

To assess the reliability of our SAR process, we have implemented a prototype tool MicroGraal[3] following our methodology. This prototype was prioritized towards components using the Java Spring Platform, which is well-adopted in the industry. In addition, we utilize a fork of GraalVM Native Image[4] as the base for our analyzer because the AOT nature of Native Image allows us to access the whole compiled application, extract the Graal IR out of the relevant methods, and perform static analysis on top of them. For the 3D visualizations that can be seen in the figures in this paper, we developed Graal Microservice Visualization Platform (MVP)[5].

A well-established community benchmark called train-ticket [28] was used for the assessment. Train-ticket is comprised of microservices that were devised by different frameworks and languages, such as Java, NodeJS, Python, and Go. Based on our static analysis method, we only assessed Java Spring Boot microservices, and all other microservices were excluded from

[3]Prototype: https://github.com/cloudhubs/graal-prophet-utils, accessed on 11/01/2024.

[4]Graal Fork: https://github.com/cloudhubs/graal, accessed on 11/01/2024.

[5]Graal MVP: https://github.com/cloudhubs/graal_mvp, accessed on 11/1/2024.

our analysis. As a result, 42 Java microservices from train-ticket v1.0.0[6] were considered in our analysis.

The analysis of one train-ticket microservice using MicroGraal takes 15 seconds and has 850 MB RSS on average when running on a MacBook Pro 2018 with 2.9 GHz 6-Core Intel Core i9 processor and 32 GB of main memory with macOS Sonoma 14.2.1. We report averages as the analysis runs of individual microservices follow the same pattern and there were no outliers.

A. *Reliability of GraalVM*

In our study, we ensured the reliability of our approach by conducting a manual analysis of the train-ticket benchmark. This manual analysis process involved three authors, where two authors extracted the data and one validated the data by examining the source code. Additionally, we used a tool from a related work by Walker et al.[4] that implemented source code-based analysis to achieve automated SAR. We ran this tool on the same version of the benchmark that we examined in this case study and compared its results with ours in the following subsections. We created the service dependency graph and context map for the testing system, taking into account several special cases specific to the Java Spring Framework. Our generated dataset[7] contains information on the extraction of entities, where we also performed a statistical evaluation of the given components in the train-ticket system.

B. *Service Dependency Graph*

Our prototype was executed to produce the service dependency graph from the benchmark by analyzing REST calls, endpoints, and their connections. The outcomes of the manual process, the proposed method, and the source code-based tool are presented in Table I. This table provides a comparison of the number of calls and endpoints and how they collaborate in constructing the service dependency graph.

TABLE I: Service Dependency Graph Data Analysis

Numbers/Approaches	**Manual**	**MicroGraal**	**Rel. Tool [4]**
REST Calls	146	146	146
Endpoints	261	261	261
Request Pairs in SDG	142	123	114
Links in SDG	90	82	82

Our analysis depicts that our prototype successfully extracted all REST calls and endpoints from the system. It is worth noting, 4 of the REST calls have URLs that point to nonexistent endpoints in the system. In terms of the number of calls participating in the service dependency graph, our prototype missed 19 pairs of calls and endpoints that were expected to be present in the service dependency graph. However, these 19 pairs did not impact the number of links between microservices. Our prototype was able to extract 82 out of the 90 links in the service dependency graph, as each link can contain multiple calls between the involved microservices.

A more thorough investigation into the 19 missed pairs reveals that 11 of them were not captured because their URLs were assigned to a variable within a conditional branch (i.e., if-else), which our prototype did not cover. Additionally, the remaining 8 pairs were not formed because their calls contained the body parameter in a pattern that our prototype was not designed to handle. Moreover, the 8 missing service dependency graph links were a result of the previously mentioned missed pairs.

Comparing the results with those of a source-code-based tool, we found that we were able to extract all the endpoints and calls, and both approaches identified the same number of 82 links in the service dependency graph. However, our method showed improvement in the number of matched call/endpoints pairs in the service dependency graph, as it was able to extract 9 more pairs than the source-code approach.

C. *Context Map*

In the constructed context map analysis, our prototype was utilized to generate a context map from the benchmark by extracting entities and their relationships with each other. The outcomes of the manual process, the proposed approach, and the source code-based tool are presented in Table II. This table provides a comparison of the entities and relationships in both the bounded context and in the context map after the merge process is executed.

The analysis of the context map data indicates that our prototype was able to construct a complete context map in terms of the number of entities and relationships, except for one entity that was missed across all microservices. However, this entity did not contain any relationships and was identified as merged in the manual analysis, so it did not impact the resulting context map.

Upon inspecting the context map data, we found that the `ts-common` and `ts-delivery-service` utility packages were incompletely parsed due to compatibility issues between the provided JAR and native image. As a result, it was not possible to retrieve entity fields within these microservices and they were omitted. However, the entity names remained available and were used to identify links between them and other microservices' entities in the system. The missed entity was the `Delivery` entity, which was not identified. However, since `ts-delivery-service` did not participate in any relationships, it did not affect the analysis. Moreover, although the `VerifyResult` entity was extracted, it was extracted from the `ts-rebook-service` microservice project, where the Maven file was configured to compile its file, and not from the microservice where it was defined, `ts-common`. Using this data, we were able to construct a holistic context map that includes entity attributes, entity names, and multiplicities between entities.

By comparing our results with the source code-based tool, we observed improvements in all the extracted data. Specifi-

[6]Train-ticket: https://github.com/FudanSELab/train-ticket/tree/v1.0.0, accessed on 11/1/2024.

[7]Dataset: https://zenodo.org/record/7902018#.ZFXrVy2B1YI, accessed on 05/05/2023.

cally, the source code-based tool failed to construct a complete context map. It missed 4 relationships and 9 entities in the bounded context, while our prototype only missed 1 entity.

TABLE II: Context Map Data Analysis

Numbers/Approaches	Manual	MicroGraal	Rel. Tool [4]
Entity Bounded Context	117	116	108
Relation Bounded Context	43	43	39
Entity Context Map	84	84	76
Relation Context Map	24	24	20

When including our extraction of `ts-common`, our approach to extracting entities correctly extracts 116 of 117 entities and 43 of 43 relationships before the combination of entities as well as 84 of 84 entities and 24 of 24 relationships after the combination of entities. When using microservices that produce JARs that can be parsed by MicroGraal, our tool is 100% accurate when compared to manual analysis. Our approach identifies more entities and relationships both before and after the condensation of entities than the related source code analysis tool listed.

VI. Discussion

Static analysis tools are commonly used by developers to assess the quality of their code and system design at an early stage. However, when it comes to microservice systems, these tools are still in their infancy and there is currently a market gap that needs to be addressed. Developing static analysis tools and visualizations for microservice systems makes it easier to identify inappropriate patterns in their design. Finding and fixing such issues leads to an improvement of the overall quality of such systems.

To ensure effective static analysis, it is crucial for any tool to establish a robust intermediate representation of the system at the outset, regardless of the type of system being analyzed. This serves as a foundational step in achieving the goal of uncovering various architectural views through the Static Analysis Review process.

However, it is important to note that static analysis alone cannot provide a complete system-centered perspective. It is essential to consider other perspectives as well, such as the system runtime, dynamic analysis, infrastructure details, team organization, and the development process. Only by taking all these perspectives into account can we obtain a comprehensive understanding of the system and achieve our quality goals.

In the context of microservice systems, current tools prioritize dynamic analysis, such as OpenTracing, due to the ease of dealing with system polyglots. However, the management of system tracings introduces complexity and can only provide a black box perspective of the system. Furthermore, dynamic analysis requires user interaction or comprehensive testing to generate traces, and this does not come free of charge. Not many companies will prioritize visible (functional) system additions over invisible (quality) solutions, which could introduce technical debt and lead to architectural degradation.

Relying solely on dynamic analysis necessitates the use of infrastructure resources to test new versions of microservices. However, these tests may be limited in scope and rely on outdated test suites, leading to the inefficient use of time and resources with significant energy footprints. Requiring dynamic analysis tests for every new commit to a microservice can lead to prolonged periods between the introduction of errors and their identification. This inefficiency hampers the development process and can significantly impede the ability to address issues quickly and effectively.

Static analysis offers advantages to cope with the previously mentioned setbacks because it does not require a running system to provide insights into the system's architecture, changes, and qualities. This makes it a closer and more accessible tool for developers, offering early feedback on code changes. By placing a single microservice in the context of other connected neighboring microservices, static analysis allows developers to reason about change propagation, impact, and implications. This approach can help mitigate ripple effects and identify potential issues early in the development cycle. Moreover, static analysis enables direct comparison to previous versions of the system, a capability that dynamic analysis does not possess. Overall, static analysis offers a powerful tool that can aid in managing complex microservice architectures and promoting high-quality software development practices.

Our work is foundational to the microservice-aware SAR as it demonstrates that static analysis can uncover reliable architectural views. The combination with GraalVM is a wise decision as it goes beyond the Java Platform. The integration of other language runtimes becomes possible with Truffle [19], which could potentially address the current greatest weakness of static analysis: its limited applicability to monoglot systems. Still, there is a long path as components across other frameworks would need to be recognized. It must also be seen in the context of low-level virtual machines [26], which are unsuitable for SAR as they operate with low-level constructs. Microservices use high-level constructs and framework components; thus, architectural views should reflect both these since developers are familiar with them in their code.

VII. Threats to Validity

1) Internal Validity: In our approach, the JAR files for microservices must be compatible with Native Image to parse entities. In the train ticket, the `ts-common` microservice had to be extracted manually to access all information within it. However, this can be mitigated by extracting the classes or fields present within other entities and extrapolating this to find the `ts-common` entities.

Our approach is limited to the identification of HTTP calls made with the `RestTemplate` class. Other ways HTTP requests are made could be parsed, such as to `HttpClient` or `HttpURLConnection` libraries. The REST call extraction can be improved by implementing detection of the pattern that we currently miss in train-ticket, which is when there is conditional branching that determines what URL or its part is to be passed or appended into the `RestTemplate` HTTP request. Though in train-ticket, this pattern does not result in missed links between microservices; in other systems, if this pattern was encountered, links could be missed if there are no other REST calls present in the source microservice to the same target microservice to still create the link.

There are limitations to what we are able to extract from the REST calls. We are not able to determine the REST call body parameters types because we were unable to retrieve that information from the Graal IR. For path parameters, it is complicated to trace through the intricate structures of the IR to identify what may be path parameters included in the URLs of REST calls. Because we do not identify if there are path parameters we do not find their types. For these calculations, the act of traversing an AST would be beneficial because these high-level values are more easily identifiable compared to Graal IR. The inability to extract REST call body parameter types could be a result of our lacking expertise and knowledge of traversing Graal IR and using the Graal API to obtain information rather than it being actually impossible to obtain. However, an overload of REST endpoints is typically not supported by containers.

The patterns our approach currently addresses are catered to that of the method structure, its code content, and the ways of making HTTP requests in train-ticket. A future improvement would address other common ways methods and calls may be executed and structured that may not be in train-ticket so that the same HTTP request information can still be extracted.

When matching endpoints to REST calls in a system, it only matches the hard-coded part of the URI. Therefore, any unique information that separates two endpoints by path parameters could compromise the matching of a REST Call to an endpoint. This is a rare scenario as the endpoints would also have to have the same HTTP Method, and both have a body present. Another scenario similar to this is to have two endpoints, one with a path parameter in the middle of a URI and one that is hard-coded such that when the path parameter is filtered out it is the same as the URI. For example, if there is a URI with this structure `/A/{B}/C` and one with this structure `/A/C` the matching will cut out the parameter "B" therefore matching the same URI. Assuming they have the same HTTP method and have a body is taken into consideration as well.

The destination microservices are found by checking if the microservice name is in the hostname or in the URI path. However, in some cases that are not train-ticket, this might not work. Therefore, in more general approaches this might have to be disabled.

2) External Validity: Our proof-of-concept has been evaluated on a single system benchmark and naturally might be biased to constructs utilized in this benchmark. We aimed to avoid this bias by targeting general constructs used in the Java Spring Framework. Thus, new systems might require adjustments in pattern matching. The intent of the case study was to demonstrate the feasibility of microservice-aware SAR on Native Images rather than to deliver a production-level solution. Custom extensions to the process to detect new patterns, new situations, and new frameworks are expected from community adoption. This is supported by open-sourcing our proof-of-concept.

To enable the reproducibility of this work, its assessment, and comparison with alternative works, we share our dataset of detected endpoints, REST calls, and entities in our case study. This can serve the scientific community in further advancements beyond the goals of this work.

An issue that our visualizer has pertains to older hardware, which can encounter performance issues if the provided microservices are excessive in size. As a result of these large microservices, it can be difficult to interact with the full context map and service dependency graph.

VIII. Conclusion

With the prevalence of microservice systems, it is important to abstract them with important information accessible in a system-centric view to understand the system, identify issues, and maintain the system. Because of the size of most microservice systems, an automated and scalable solution is needed.

This paper elaborates on microservice-aware SAR methodology utilizing recent advancements in improving microservice initialization times. GraalVM Native Image has been used to demonstrate the feasibility of the process and reliability of produced results when compared to a manual review of source code. Our proof-of-concept tool MicroGraal demonstrates that Native Images can be analyzed to produce two important architectural views for microservice systems, the service and domain views. In a case study using a third-party system, we demonstrated promising results to build service dependency graphs and context maps that can assist practitioners in making informed decisions and act as the foundation for advancements in static analysis for microservice systems.

Future work will target an analysis of polyglot systems and a broader analysis of other system benchmarks.

Acknowledgements

This work has been supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This material is also based upon work supported by the National Science Foundation under Grant No. 2245287.

References

[1] M. Kleehaus, O. Uludag, P. Schäfer, and F. Matthes, *MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments*, 06 2018, pp. 148–162.

[2] G. Granchelli, M. Cardarelli, P. Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," 04 2017, pp. 298–302.

[3] F. Rademacher, S. Sachweh, and A. Zündorf, “A modeling method for systematic architecture reconstruction of microservice-based software systems,” in *Enterprise, Business-Process and Information Systems Modeling*, S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic, Eds. Cham: Springer International Publishing, 2020, pp. 311–326.

[4] A. Walker, I. Laird, and T. Cerny, “On automatic software architecture reconstruction of microservice applications,” in *Information Science and Applications*. Singapore: Springer Singapore, 2021, pp. 223–234.

[5] N. Alshuqayran, N. Ali, and R. Evans, “Towards micro service architecture recovery: An empirical study,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 47–4709.

[6] 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.

[7] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, “Using microservice telemetry data for system dynamic analysis,” in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 29–38.

[8] “Software, systems and enterprise — Architecture description,” International Organization for Standardization, Geneva, CH, Standard, Nov. 2022.

[9] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, “One VM to rule them all,” in *Proceedings of Onward!* ACM Press, 2013.

[10] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, “Initialize once, start fast: Application initialization at build time,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, p. 184, 2019.

[11] VMWare, “Spring,” 2023. [Online]. Available: https://spring.io/

[12] Micronaut foundation, “Micronaut,” 2023. [Online]. Available: https://micronaut.io

[13] RedHat, “Quarkus,” 2023. [Online]. Available: https://quarkus.io

[14] Oracle, “Helidon,” 2023. [Online]. Available: https://helidon.io/

[15] L. O’Brien, C. Stoermer, and C. Verhoef, “Software architecture reconstruction: Practice needs and current approaches,” 2002.

[16] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, “Self-optimizing AST interpreters,” in *Proceedings of the Dynamic Languages Symposium*. ACM Press, 2012, pp. 73–82.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” vol. 13, no. 4, pp. 451–490, 1991.

[18] “Graal project,” 2013. [Online]. Available: http://openjdk.java.net/projects/graal

[19] C. Wimmer and T. Würthinger, “Truffle: A self-optimizing runtime system,” in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH ’12. New York, NY, USA: Association for Computing Machinery, 2012.

[20] “Truffle language implementations,” https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/Languages/, 2023, accessed: 2023-04-26.

[21] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, “Graal IR: An extensible declarative intermediate representation,” in *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[22] C. Click and K. D. Cooper, “Combining analyses, combining optimizations,” *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, p. 181–196, mar 1995. [Online]. Available: https://doi.org/10.1145/201059.201061

[23] D. Landman, A. Serebrenik, and J. J. Vinju, “Challenges for static analysis of Java reflection: Literature review and empirical study,” in *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 2017, pp. 507–518.

[24] 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)*, 2022, pp. 49–58.

[25] D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas, “On automated rbac assessment by constructing a centralized perspective for microservice mesh,” *PeerJ Computer Science*, vol. 7, p. e376, 2021.

[26] 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.

[27] F. Christiane and K. Brown, “Wordnet and wordnets,” in *Encyclopedia of Language and Linguistics*. Oxford: Elsevier., 2005, pp. 665–670.

[28] 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, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991