Microservices Architecture Language for Describing Service View

Luka Lelovic[a], Michael Mathews[b], Amr S. Abdelfattah[c] and Tomas Cerny[d]
Baylor University, Computer Science, 1311 S 5th St, Waco, TX 76706, U.S.A.

Keywords: Architecture Reconstruction, Architecture Language, Service View, Intermediate Representation.

Abstract: Microservices Architecture is a growing trend in recent years that has been promoted due to a number of researched advantages. However, as microservice systems grow and evolve, they can become complex and hard to understand. In order to face this problem, techniques to reconstruct, describe and visualize these systems are proposed. Despite this, there are currently no architectural languages actively maintained, adopted, and promoted as the intermediate between the system reconstruction and its corresponding viewpoints. This paper proposes a YAML-based architectural language acting as the intermediate representation for microservice architecture, specifically in the service view architectural perspective. This paper outlines the new language, its basis, example descriptions, and possible architectural visualizations of the descriptions. It also details how it compares to other existing architectural languages in the microservice domain.

1 INTRODUCTION

Microservices Architecture (MSA) is a popular architectural pattern that emphasizes small, independent sets of services. Each of these services has its own codebase and storage system. MSA has been widely promoted and adopted in recent years for its extensive benefits in fault tolerance, loose coupling, and high scalability (Waseem et al., 2021).

A considerable issue arises, however, when trying to maintain a holistic perspective as new independent services are introduced. As complexity grows, it becomes increasingly difficult to understand the entire system. Software Architecture Reconstruction (SAR) attempts to mitigate this issue through either manual, static, or dynamic analysis. After analysis, the system information can be extracted into a proprietary intermediate representation (Walker et al., 2021). In this perspective, system extraction is often tightly connected with a particular purpose, such as the visualization of different system views (service, topological, etc.). There then exists an opportunity to uncouple the system information extraction from the task we target through the form of an architectural language.

While a number of architectural languages exist for various system concerns and architectures, there are none that exist and have seen widespread adoption as the intermediate between system reconstruction and system visualizations. If we introduced an architectural language to describe service views, we could divide focus into reliable information extraction using static or dynamic system analysis from reasoning about the system or its visualization or other reasoning. This situation is sketched in Fig 1.

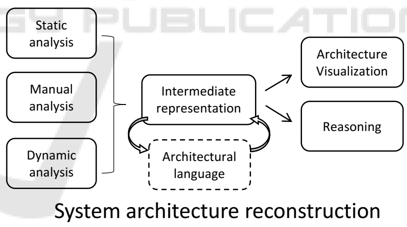

Figure 1: Positioning architectural languages to interface between system analysis and its interpretation.

In this paper, we propose a Microservice Intermediate Representation Language (MIRL) for the Service View perspective. It is a YAML-based description language for intermediate system representation between system reconstruction approaches and consequent visualizations serving human experts. Such language can be used to track service changes across system evolution using version control. It can also be used when planning existing system evolution. In particular, the system service view description can be speculatively extended to let human experts reason about the evolution implication using the planned system visual representation.

The rest of the paper is organized as follows: Sec-

[a] https://orcid.org/0000-0003-0785-7499
[b] https://orcid.org/0000-0002-8457-9894
[c] https://orcid.org/0000-0001-7702-0059
[d] https://orcid.org/0000-0002-5882-5502

Lelovic, L., Mathews, M., Abdelfattah, A. and Cerny, T.
Microservices Architecture Language for Describing Service View.
DOI: 10.5220/0011850200003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 220-227
ISBN: 978-989-758-650-7; ISSN: 2184-5042

tion 2 describes background information. Section 3 describes related work. Section 4 introduces our language. Section 4.2 provides a case study of example systems that were described with the language. Section 5 provides discussion of how our language can act as the intermediate from a static analysis approach, and then utilized in a visualization tool. Section 6 concludes the paper and provides a discussion of future work.

2 BACKGROUND

To understand complex systems built with microservices, we typically perform Software Architecture Reconstruction (SAR). SAR is a reverse engineering process that obtains a representation of software architecture from system artifacts. The aim is to reduce detail and provide a more abstract view for better understanding of certain system aspects. SAR can be performed through many means. It can involve manual, static, or dynamic analysis of the system (see Fig 1) according to the nature of the data, skills of the team, or available tools. Static data exists in many forms, such as source code, configuration files, documentation, code reviews, and the source control commits changes and their messages. The dynamic data is produced during the runtime, which include logs, traces, telemetry data, etc. Moreover, some artifacts share both static and dynamic behaviors, such as system tickets which can be static as requirements or dynamic as production issues.

In order to successfully reconstruct a system, effective system views must be chosen to provide good coverage of system concerns. Work from (Rademacher et al., 2020) classifies system views into the following four:

- *Domain View:* This view covers domain concepts and domain concerns of the system.
- *Technology View:* This view describes the technologies used to implement the microservices.
- *Service View:* This view describes the service models that specify microservices, interfaces, and endpoints.
- *Operation View:* This view focuses on service deployment and infrastructure for service discovery and monitoring.

In relation to architectural languages, these views mentioned above are of primary concern. Since each microservice is self-contained and should have an independent codebase, system polyglots can exist using different languages and conventions (Wiggins, 2017). The disadvantage of this is it results in a more challenging reconstruction process.

Rademacher et al. (Rademacher et al., 2020) suggested performing SAR manually by analyzing various codebase artifacts and existing documentation. Such a process must analyze individual microservices and combine the analyzed results before some benefits like conformance checking, system-centric view perspectives, and others can be provided.

The major challenge in microservices is the decentralized codebase and a missing system-centric overview which could help with system evolvability (Bogner et al., 2021) (i.e., avoid ripple effects, better plan evolution). Constructing a service dependency graph is one the most beneficial perspectives for microservices, as it gives a system-centric perspective (Bogner et al., 2021). While each SAR perspective adds value to human experts, the *service view* fits within the service dependency graphs that describe dependencies between microservices. It shows where microservices call each other (Mayer and Weinreich, 2018). Moreover, (Mayer and Weinreich, 2017) and (Rademacher et al., 2020) identified that this view plays an essential role in understanding how microservice-based systems operate. Therefore, supporting it should be one of the most important goals of SAR.

The *service view* has been recently used by many dynamic analysis tools like Jaeger, or Kiali (Gortney et al., 2022). These tools use distributed tracing, log analysis, and monitoring, from which they extract dependency call graphs that can be aligned with the boundaries of particular microservices. However, the use of dynamic analysis requires system interaction to collect data, and thus the results can only be as complete as good the collected data are. Dynamic analysis often gives a black-box view of the system.

Static analysis techniques can also be employed to generate intermediate representations for the system (i.e., for the service view). These representations serve as an intermediary to technical reasoning or facilitate documentation extraction and can be used to track systems' performance, architecture improvements, and degradation during the evolution process (Rademacher et al., 2020).

As opposed to dynamic analysis, static analysis can be performed on a system before it is deployed. Its techniques are employed which inspect source code to extract the service description and its API specifications. Extracting this representation requires the detection of the endpoints of each service. Some software frameworks help in defining the endpoints, such as Swagger[1]. Once the list of endpoints and calls is collected for each service, the analysis process uses the relative endpoint URL, the HTTP method,

[1] Swagger: https://swagger.io

and parameters to match the calls to endpoints. The result is a service dependency graph representing the system, showing how the services communicate with each other.

3 RELATED WORK

There are numerous description languages ranging in type and purpose. A number of related studies, SAR tools, and languages have been published as promoted solutions for MSA composition, for brevity in this paper length we refer to a related mapping study (Lelovic et al., 2022) on architectural languages in relation to microservice systems.

4 PROPOSED LANGUAGE: MIRL

To describe the service view MIRL uses various building blocks. At a basic level, it enforces a system name, version, and an array of node objects. Within the node objects consist of the following four attributes:

1. *nodeName:* This is the name of the current system node/object being described.
2. *nodeType:* This is the type of the current node (service, kafka, etc). The types of systems can have any range of values depending on what the stakeholder of a particular system defines.
3. *dependencies:* This is an array of objects depicting incoming connections into the current node.
4. *targets:* This is an array of YAML objects depicting outgoing connections into other nodes.

The dependency and target arrays depict endpoints for each microservice in a particular system. The only requirement for dependency and target objects is that a node name is provided.

Listing 1: Basic MIRL Example.

```yaml
---
systemName: MIRL-Basic-Example
systemVersion: 0.0.1
nodes:
- nodeName: basic-service-1
  nodeType: service
  dependencies: []
  targets:
  - nodeName: basic-service-2
- nodeName: basic-service-2
  nodeType: service
  dependencies:
  - nodeName: basic-service-1
  targets: []
...
```

A basic MIRL example can be seen in Listing 1. This shows a general example/template of the minimum requirements in our language. In the basic example template, the basic-service-1 has no incoming dependency endpoints, but makes an outgoing connection to basic-service-2. The basic-service-2 node then describes this dependency to basic-service-1. This would ideally be visualized in a workflow perspective as an arrow between basic-service-1 to basic-service-2.

The core language constructs are extensible. It is possible to capture other attributes such as the endpoint types and execution steps of a particular node. MIRL allows endpoints to be described either at a high-level or fine granularity depending on the architect's needs. This can be seen with Listing 2 where each dependency and target node is only given a name and associated request array describing the type of request the node is making on the current, and the step of execution in the flow of the system. By contrast, Listing 3 describes more in the endpoints and their requests (e.g. arguments, return types, etc). Overall, this structure provides the capability for orchestrating microservices and their choreography, as well as describing the workflows between them.

MIRL as well requires a *systemName* and *systemVersion*. This provides the capability for version control to tracks changes as the system evolves. Comparing across multiple MIRL system versions can allow service changes to be tracked by human experts utilizing the language alongside various corresponding visual tools.

Despite being a language for microservices, the language also supports other types of node objects within the system. A list of node types is specified in a separate JSON file as a list. These types can be extended as well to fit the architect's needs of node types. These types are used to generate the node images from the Python Diagrams library. The node types provided are: [*customer*, *srcSink*, *archive*, *kafka*, *service*, *bucket*, *pipeline*, *proxy*, *writer*, *database*, *config*, and *API*]. These types were derived based on assessment of industry project architectures and feedback from practitioners involved in microservice system development.

MIRL is provided with a JSON schema for validation. This schema can validate both YAML and JSON descriptions. The main requirements/concepts of MIRL nodes and their relationships can be seen in Figure 2.

Table 1: Comparison of Features with MIRL and Other Languages.

Language	Notation	Language Agnostic	Intermediate Solution	Visual Component
Jolie	OL			
Medley	JavaScript			
Thrift	Thrift			
Silvera	Python			
BPMN	XML	✓		
TOSCA	YAML	✓		
MicroART	Java		✓	✓
MIRL	YAML	✓	✓	✓

4.1 Feature Comparison with Other Languages

When looking at a broader context, we based MIRL's structure on common service composition approaches found in existing description languages, and the perspectives within the example systems that we describe in Section 4.2. We compare to similar languages for microservice description from related work by (Lelovic et al., 2022), and breakdown language capabilities and features in Table 1 in acting as intermediate solutions based on the 4 criteria:

- **Notation:** The notation of the language is important to note as it outlines what format visualizers would have to parse.
- **Language Agnostic:** This feature ties into the notation, where architectural languages that utilize language's with compilers (Java, Jolie, etc) are platform-dependent and cannot be easily parsed.
- **Intermediate Solution:** This feature outlines whether a language is initially proposed as an intermediate between reconstruction and corresponding visualizations. Currently MicroART and MIRL are the only two found for this.
- **Visual Component:** This feature describes whether a language is coupled with its own visualization tool or generator for building a visualization from the language.

From Table 1, it can be seen that MIRL is the only language that was found to be language agnostic, composing an intermediate solution, and being coupled with a visualization component.

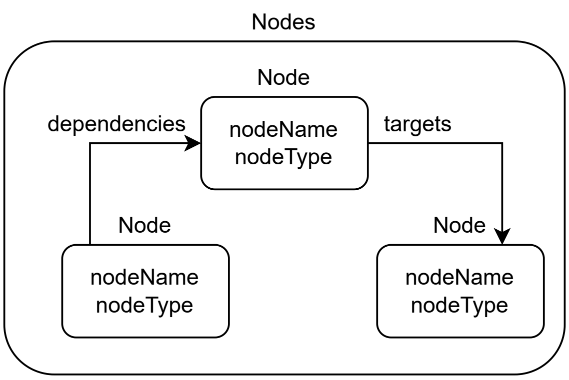

Figure 2: Relationship Structure in MIRL.

4.2 Sample Demonstrations

For a demonstration we describe two existing MSA-based systems in MIRL. The first system is the architecture pipeline model for Red Hat Insights[2]. The second is an existing microservice benchmark TrainTicket (Zhou et al., 2018) while using the service view architecture reconstructed by static analysis provided in (Walker et al., 2021)[3]. From both of these descriptions, a visual model was then generated by parsing our language and utilizing the Python Diagrams library and is shown to demonstrate the final outcome.

4.2.1 Red Hat Pipeline

The Red Hat pipeline, seen in Figure 3, is a manually modeled diagram by the system architect. It sketches the flow of microservices within Red Hat's system architecture. The description in MIRL was performed manually based on the model and its captured services.

The *service view* of the Red Hat pipeline is represented through the nodes within the dependency and target arrays. These nodes describe the microservices, interfaces, and endpoints which model the interaction between the nodes. Listing 2 shows a particular node in the Red Hat description. In the SHA-extractor, the targets array shows one outgoing connection. The SHA-extractor node has an array of requests extended with it. This requests array is not enforced by the language. The request in the array describes an outgoing POST request to this node that is being made as the first step in the system execution.

The *topology* of the Red Hat pipeline is shown as a singular network through the array of nodes that binds

[2]https://github.com/RedHatInsights/insights-results-smart-proxy/blob/master/docs/Overall_architecture.png

[3]http://demo1986600.mockable.io/train

the entire system together. Any node in the system must be described in this array as a YAML object. The topology is coupled with the service view of dependencies.

Figure 4 shows the Red Hat pipeline *visualized* based on the service view that is described in the language. While the provided visualization is a different perspective from the initial model in Figure 3, the information provided within the model remains one-to-one. It is important to note as well that the Python Diagrams library limited our visualization to certain node images. In Figure 4, there are nodes with given images that are not the same as the ones from the initial pipeline model. One thing that could be extended in the future for the Red Hat description and its visualization is describing the clustering of nodes and layers, such as the 3Scale layer.

4.2.2 Feedback from the Architect

After describing the Red Hat pipeline in MIRL and generating a corresponding visualization, the results were given to a Red Hat architect for feedback. The architect provided a positive reception of the language and found its structure to be potentially reusable for several purposes. One area the architect noted for improvement and optional extension was providing additional information on the messaging formats between nodes and additional information on the nodes themselves. Visualization tools could then provide this additional information when nodes are clicked on. This would be similar to the functionality provided here.[4] MIRL fully supports optional attributes such as this, but does not standardize any in its schema. The schema could be improved in the future to provide this.

Listing 2: Red Hat Pipeline (snippet) in MIRL.

```
...
- nodeName: SHA-extractor
  nodeType: pipeline
  dependencies:
  - nodeName:
      Kafka-topic-platform.upload.buckit
    requests:
    - requestType: POST
      executionStep: "#1"
  - nodeName: S3-bucket
    requests:
    - executionStep: "#2"
  targets:
  - nodeName: Kafka-topic-archive-results
    requests:
    - requestType: POST
      executionStep: "#17"
...
```

[4]https://redhatinsights.github.io/insights-results-smart-proxy/overall-architecture.html

4.2.3 Train Ticket System

The second system analyzed and described was the benchmark train ticket system. While the description of the Red Hat pipeline was minimally extended outside of the requirements that the language enforces, the train ticket system demonstrates MIRL's extensibility in a greater amount. Given the example train ticket YAML file that was reconstructed, we wrote a Python script to parse the language into MIRL. This increased the accuracy of our description and reduced the manual effort required in contrast to the Red Hat pipeline.

The *service view* of the Train Ticket System is represented largely the same as the Red Hat pipeline. In Listing 3, the order other service is shown, with one node of the many nodes in its dependencies array given, and the only node in its targets array described as well. The incoming and outgoing connections differ from the Red Hat pipeline by the different attributes displayed in the requests array. While the type of request is still given, other attributes such as the arguments, return types, function names, and request paths are also given. These attributes are entirely given by the train ticket system and not enforced by MIRL's schema. Other attributes as well such as the length and width of the connections are given, which could be translated into visualization software.

The topology is similar to the Red Hat pipeline. The same array of nodes is required. One notable extension of the train ticket system is that the shape of the node is provided. The order other service is given a node shape of a box. Like the length and width attributes within the dependencies and targets array, the node shape could also be used within visualization software to display how the individual node looks. This further shows how the MIRL language can act as the intermediate between the software reconstruction and the visualization of the system. This also demonstrates how the topology is necessarily represented alongside the service view to maintain a complete view of the system.

Figure 5 shows the train ticket system fragment *visualized* based on the service views described in the language. Since everything in the benchmark system is a service, the nodes all contain the same node image. The visualization shows a notable difference from the Red Hat pipeline in terms of the dependencies and connections between nodes. Due to multiple requests, a particular service in the system makes to outgoing services, there are often multiple connec-

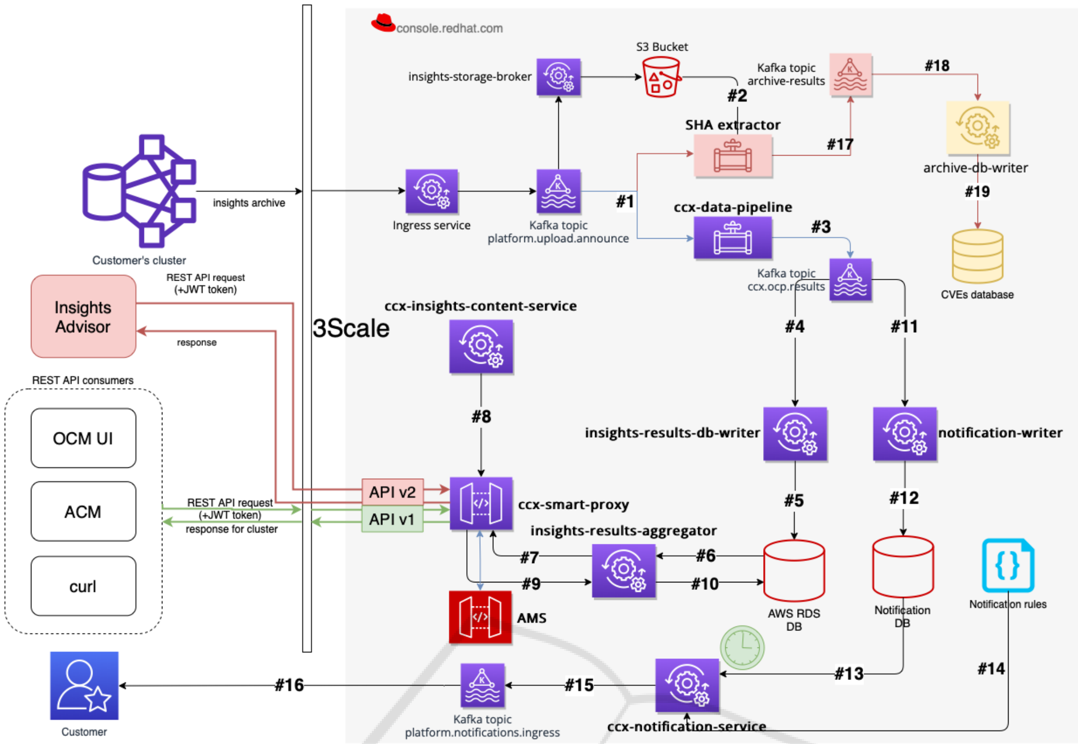

Figure 3: Red Hat Pipeline[2].

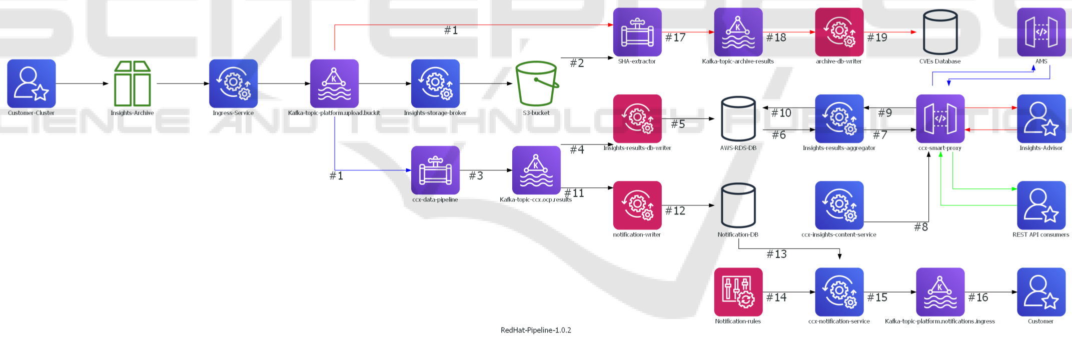

Figure 4: Red Hat Description Visualized.

tions between nodes. While one advantage of this may be that it shows the detailed coupling between nodes, a downside is the readability of the visualization.

5 DISCUSSION

This section discusses the language's ability to act as an intermediary. In particular, we demonstrate how MIRL acts as the intermediate between static analysis and a 3rd-party visualization tool. This was the aim we presented in Figure 1.

5.1 Static Analysis Approach

The initial retrieval of the Train Ticket system was taken from an existing reconstruction output of the benchmark system performed through static analysis by (Walker et al., 2021). The output of this was then parsed and translated into a MIRL description following the MIRL schema through a Python script. It should be noted that the reconstruction method, however, could be modified to create output in MIRL rather than its custom metadata. However, this opens MIRL usage to any other approach, whether using static or dynamic analysis or manual intervention. For instance, we could perform event trace analysis from

Listing 3: Train Ticket System in MIRL.

```
...
- nodeName: ts-user-service
  nodeType: service
  nodeShape: box
  dependencies:
  - nodeName: ts-admin-user-service
    requests:
    - type: DELETE
      argument: "[@PathVariable ...]"
      msReturn: ResponseEntity<Response>
      endpointFunction: ...
      path: "/UserController/deleteUserById"
    ...
  targets:
  - nodeName: ts-auth-service
    requests:
    - type: POST
      argument: "[@RequestBody ...]"
      msReturn: HttpEntity<Response>
      endpointFunction: ...
      path: "/AuthController/createDefaultUser"
    length: 200
    width: 1
...
```

a microservice system and extract service dependencies as commonly done by other researchers or tools like Jaeger or Kiali. The great benefit of an intermediary language is that we can extract the current system version service view and speculate on system evolution to assess possibilities before we perform the actual implementation.

5.2 3 Rd-Party Visualization

To demonstrate the other end of the intermediary, we show the use of a third-party tool that we can feed with MIRL. A 3-dimensional visualization tool has been identified aiming at constructing visualizations of microservice systems [5]. This tool imports a JSON format of the system data. The system described in MIRL can be converted to such a JSON and adopted to visualize the system described by the intermediate language. Figure 6 shows the Red hat pipeline visualized within this tool. The tool can provide visualization in both a 3-dimensional perspective and a 2-dimensional perspective. This demonstrates the potential applicability of our language to outside visualization tools as an imported format.

5.3 System Evolution

There exists another promising exploration in observing the evolution of a system through system version-

[5]https://github.com/patrickeharris/ArchitectureVisualizationPOC

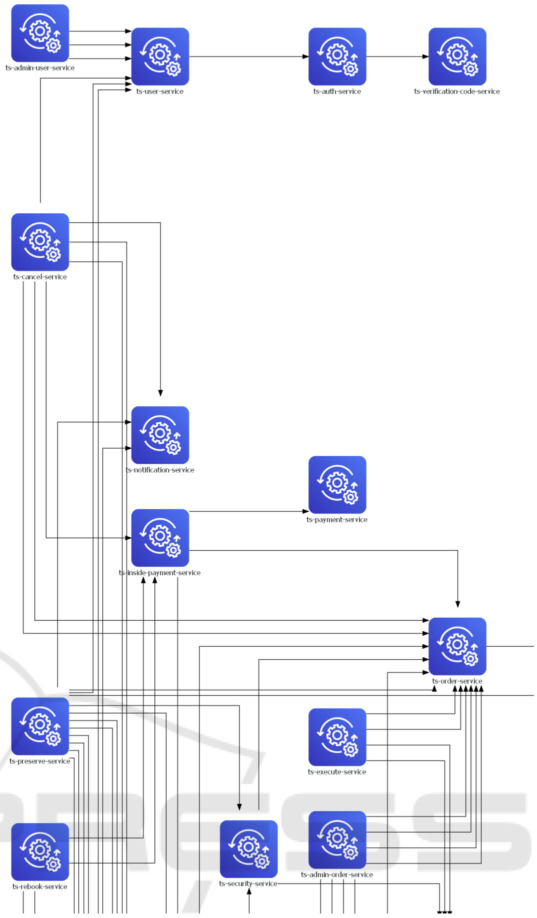

Figure 5: Train Ticket System Visualized (fraction of the whole system).

ing provided in MIRL. This evolution of the system includes adding a new service, a new version, and moving endpoints between services. Therefore, representing the service view intermediary helps to highlight these dependencies in an early stage.

Through this avenue, the analysis and the tracking of changes support detecting the impact on the architecture degradation over time. Therefore, MIRL supports the system to be automatically updated, tracked, and shown in a visualization or provenance tracking tool. Thus it would become easier to keep going with the rapid evolution pace of the system development.

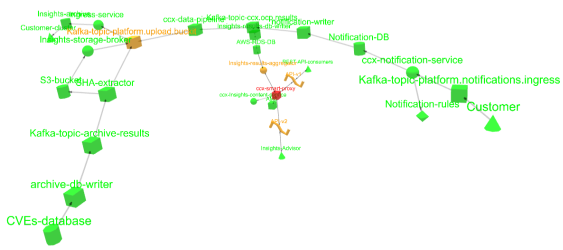

Figure 6: Red Hat Pipeline Visualized with 3rd Party.

6 CONCLUSION

In this paper, we presented a new Architectural Language for the Service View perspective (MIRL) which can act as an intermediary step to interface between various system analysis approaches and consequent reasoning or visualization to aid human experts. In addition, it can serve to better understand the evolutionary changes of a system as it can be easily tracked in version controls.

When compared to other languages that could act as potential intermediates, such as BPMN, TOSCA, and the MicroART-DSL, these languages were found to either have a number of complex requirements resulting in parsing hurdles or be proposed for a different domain or understanding of a system. Compared to other languages, MIRL is lightweight and allows considerable extensibility for the architect to examine and the visualization tool to easily parse.

There are several planned areas of future work. The first is to conform a currently maintained visualization tool(s) to accept MIRL as an import. The second is to develop or extend an existing reconstruction technique to generate system descriptions conforming to the language schema. Ultimately this would allow architects to automatically retrieve a description and corresponding visualization of their microservice systems. A final area of future work would be adding more optional descriptive attributes for nodes in the MIRL schema, such as clusters, layers, and node or request descriptions.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

REFERENCES

Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2021). Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):1–39.

Gortney, M. E., Harris, P. E., Cerny, T., Maruf, A. A., Bures, M., Taibi, D., and Tisnovsky, P. (2022). Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. *IEEE Access*, pages 1–1.

Lelovic, L., Mathews, M., Elsayed, A., Cerny, T., Frajtak, K., Tisnovsky, P., and Taibi, D. (2022). Architectural languages in the microservice era: A systematic mapping study. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, RACS '22, page 39–46, New York, NY, USA. Association for Computing Machinery.

Mayer, B. and Weinreich, R. (2017). A dashboard for microservice monitoring and management. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 66–69.

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

Rademacher, F., Sachweh, S., and Zündorf, A. (2020). A modeling method for systematic architecture reconstruction of microservice-based software systems. In Nurcan, S., Reinhartz-Berger, I., Soffer, P., and Zdravkovic, J., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326, Cham. Springer International Publishing.

Walker, A., Laird, I., and Cerny, T. (2021). On automatic software architecture reconstruction of microservice applications. In Kim, H., Kim, K. J., and Park, S., editors, *Information Science and Applications*, pages 223–234, Singapore. Springer Singapore.

Waseem, M., Liang, P., Shahin, M., Di Salle, A., and Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061.

Wiggins, A. (2017). The twelve-factor app. (Accessed on 10/02/2021).

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018). Benchmarking microservice systems for software engineering research. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, 2018*, pages 323–324. ACM.