IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 October 2022, accepted 31 October 2022, date of publication 9 November 2022, date of current version 17 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3221130

TOPICAL REVIEW

Visualizing Microservice Architecture in the Dynamic Perspective: A Systematic Mapping Study

MIA E. GORTNEY[1], PATRICK E. HARRIS[1], TOMAS CERNY[1], ABDULLAH AL MARUF[1], (Member, IEEE), MIROSLAV BURES[2], DAVIDE TAIBI[3,4], AND PAVEL TISNOVSKY[5]

[1]Computer Science Department, Baylor University, Waco, TX 76798, USA
[2]Computer Science Department, Faculty of Electrical Engineering, Czech Technical University in Prague, 166 36 Prague, Czech Republic
[3]M3S Group, University of Oulu, 90570 Oulu, Finland
[4]Cloudsea Group, Tampere University, 33100 Tampere, Finland
[5]Red Hat, 612 00 Brno, Czech Republic

Corresponding author: Tomas Cerny (tomas_cerny@baylor.edu)

This work was supported in part by the National Science Foundation under Grant 1854049, in part by the Red Hat Research https://research.redhat.com, in part by the Ulla Tuominen Foundation (Finland), and in part by the Academy of Finland under Grant 349488-MuFAno.

ABSTRACT As microservices become more popular, more drawbacks become apparent to developers. One issue that many teams face today is the failure to visualize the entire system architecture holistically. Without a full view of the system, the architecture can become convoluted as teams add and subtract from their system without reconciling their changes. One established practice to determine a view on the entire system involves dynamic analysis of microservice interaction and dependencies. In this mapping study, we investigate dynamic analysis as a way to visualize system architecture. Capturing the architectural view with dynamic analysis has the ability to build the system and then show its behavior at run-time. We identify dynamic analysis techniques, the corresponding tools, and the models that these practices can generate. The findings of this study are relevant to developers of decentralized systems looking for a way to visualize their system architecture in a dynamic perspective.

INDEX TERMS Architecture visualization, dynamic analysis, microservices.

I. INTRODUCTION

The industry adoption of microservice architecture has accelerated rapidly.[1] Microservices have solved many of the issues faced by traditional, monolithic systems. The segmentation of a system into small, independently deployable elements aids in the development process and production environment. With microservices, developer teams can work independently in building the system, even utilizing distinct languages and frameworks. In production, services can easily be duplicated to save costs by dynamically managing high workloads or replacing malfunctioning services. Overall, these benefits help to decrease development time while increasing profit and uptime, which is why so many companies, such as Amazon, Netflix, and Spotify, have taken advantage of microservices.

While microservices solved many of the issues faced by monolithic systems, new problems have arisen. One major problem is the system decentralization that is crucial to a microservice framework. Such decentralization makes it difficult to create and maintain a unified view of the system. Without a centralized system view, individual parts could be well maintained with issues that appear when the parts interact. There could be inefficiencies and smells that grow over time as the system evolves [1] and possibly unintentionally degrades. There is also no guarantee that the prescribed system layout matches the actual run-time view of the system [2]. A visualization that could illustrate systems based on

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

[1]https://www.oreilly.com/radar/cloud-adoption-in-2020/

microservice architecture from a high level would make the system more understandable, fault-tolerant as well as easier to manage, extend and adapt to evolving needs.

As with most systems at runtime, even microservices generate event logs; log messages can be coded by developers or instrumented by tools into the system code or binary. Despite the decentralization, in production environments, all generated log data is typically aggregated into a centralized log to facilitate system debugging. Similarly, monitoring of microservices is in place in production, collecting various runtime metrics into a central focal point. This centralized run-time data is an ideal candidate to be used by tools to reconstruct the interactions between services and help uncover the system-centered view of the system architecture. Visualizing the system architecture is one possible approach to verify if the actual architecture of a microservice-based system resembles the prescribed one. For this purpose, different works recently proposed visualizations to represent the system at runtime. However, there are not yet reviews or summaries of the different types of visualization.

Our work aims at providing a comprehensive overview ad a classification of the different approaches proposed to visualize the architecture of a microservice system at runtime.

We specifically investigate five aspects: 1) What dynamic analysis methods can be used to visualize a microservice-based architecture; 2) What are the available dynamic analysis tools that provide an architectural visualization; 3) What are the features and properties of these tools; and 4) what information can be extracted when visualizing the software architecture. Understanding these aspects represents a crucial challenge for software engineering. The results of our work can indeed inform researchers and practitioners on the existing microservice architectural visualization tools and their characteristics so that they can make informed decisions on what of them to adopt when starting the development of a new system.

We address our goal with a *systematic mapping analysis* of the literature [3]. Through this process, we identify and classify the existing literature on the visualization of microservices-based architecture using tracing data. Starting from a total amount of 562 resources identified by querying four, we ended up with the analysis of 20 primary papers that contributed to addressing the research goals of the study. We report the results achieved when considering our four research angles, concluding with a set of recommendations and lessons learned that may drive future researchers and practitioners interested in visualizations of the microservices-based architecture at runtime. Last but not least, we highlighted the gaps in current practices and tools and suggested steps that can be taken to create more enhanced and robust visualizations.

The remainder of this paper is organized as follows. Section II gives a general background on microservices, software architecture reconstruction (SAR), and dynamic analysis. Section III summarizes similar mapping studies that have been performed in this field. Section IV describes how the authors collected the relevant papers on visualizing architecture. Section V answers the research questions listed in section IV. Section VI discusses challenges in this field of research and potential gaps to be filled in the future. Section VII discusses our results and begins to form final conclusions. Section VIII details any threats to the validity of our study and what steps were taken to mitigate those threats. Finally, we conclude the paper in section IX with a general summary of our contributions.

II. BACKGROUND

A microservice-based system is characterized by a makeup of distinct containers, also commonly called services. Each container is an independent part with a small but specific responsibility to play in the overall system. Services can communicate with each other using lightweight protocols, but good design principles dictate that they should depend on as few other services as possible in order to maintain low coupling. Containers also are self-contained environments meaning they do not require a full operating system to run, so one physical machine can support many containers with little overhead. Similarly, connected services can also be deployed across multiple physical machines. Microservices take advantage of three-tiered architecture with service endpoints at the top, business logic in the middle, and data persistence at the bottom.

Over time, large microservice-based systems tend to have services added, modified, moved, or removed, making the system more difficult to maintain and introducing possible regression and technical debt. These changes are often performed by numerous developer teams, each working on a specialized part of a system that can contain thousands or tens of thousands of different containers. As the system grows more complex, the architecture drifts from its intended view in a process called software architecture erosion [4]. As the name suggests, this process can erode, or degrade, the performance of the system by introducing various inefficiencies or anti-patterns. To mitigate the effects of software architecture erosion, it is necessary to visualize or reconstruct software architecture in order to capture an entire, holistic view of the system.

To reconstruct software architecture, both static and dynamic analysis [5] can be used. Static analysis takes the code base and attempts to reconstruct the architecture using the system data recorded in the source code and other artifacts. As opposed to dynamic analysis, it does not execute the system. Dynamic analysis is performed by executing the system on sufficient inputs without extracting system data from the source code but rather by considering generated logs or collected run-time metrics. To collect sufficient run-time data, the inputs on which the system executes might be in the form of tests that intend to cover all possible system interactions; or for the software architecture reconstruction process, many approach resort to using production traffic. The produced run-time data include event logs, and traces,

to identify the interactions among different services, considering how the system operates.

In dynamic analysis, data from system artifacts produced at run-time are aggregated and analyzed to reveal physical or logical connections between system parts. Cerny et al. [6] reviewed the benefits and drawbacks of dynamic analysis for reconstructing software architecture. Dynamic analysis relies on traffic from the system at run-time instead of inspecting code, meaning that dynamic analysis can be language-independent and analyze a system that consists of many different code bases. Additionally, dynamic system analysis can capture complex behaviors and performance metrics that cannot be extracted from static code.

On the other hand, Cerny et al. [6] conclude dynamic analysis is only as reliable as the data produced by the system in a given timeframe being analyzed. If a specific call contained in the code is never executed, it would be lacking from a reconstructed architecture.

This paper reviews research works applying dynamic analysis to provide a holistic system view of a microservice system, evaluates practical tools to do so, and determines what gaps exist in these practices.

III. RELATED WORK

Research into visualizing software architecture has mainly been conducted with a focus on static analysis and examination of source code. Some research into traceability as a practice has been done but not extensively.

Mattila et al. conducted a study in [7] on 83 articles from 2010 to 2015. They focused on using static analysis to capture metrics within the source code, such as coupling, code complexity, and cohesion. These metrics can then be visualized. This study found that the three main themes of visualization are interaction, methods used, and tools. These themes are then most related to graph and tree visualizations.

Salameh et al. found in [8], composed of 29 studies between 2002 and 2014, that there were several different concepts to base visualizations on: graph-based, notation-based, matrix-based, or metaphor-based. Graph-based visualizations are denoted by nodes and links that capture the structural relationships of the software. The notation-based method uses notations such as Unified Modeling Language to illustrate the inner workings of the software. Matrix-based visualizations use matrices to display detailed dependencies between two modules, and metaphor-based uses an analogy to better illustrate an object-oriented system and its evolution.

While static analysis is most commonly used in software visualization, research has been done on identifying traceability between source code and design artifacts. Javed and Zdun discovered in [9], which is composed of 11 studies between 1999 and 2013, that there are different techniques for identifying traceability in source code. One of these techniques is model-based traceability. Model-based traceability focuses on providing links between pieces of source code and the component in the model they represent. When the source code changes, the model changes. A similar technique was uncovered by Aung et al. in [10], which is made up of 33 studies up to the year 2019. In this study, they identify traceability as an integral part of software change impact analysis, also called CIA. To recover traces to be used in a model, there are four different approaches: information recovery, heuristic-based, machine learning, and deep learning. Traceability based on information recovery is most similar to dynamic analysis, which is covered extensively in this study. Our study will be the first to cover the research on visualizing architecture through dynamic analysis as a singular topic.

IV. MAPPING STUDY METHOD

In this study, we used a structured procedure to collect and synthesize the research works on visualizing architecture from a dynamic perspective. We followed the guidelines presented by Petersen et al. [3]. Our complete mapping study document can be found below,[2] detailing our filtration and mapping process.

In the first phase of our mapping study, we defined a set of research questions to acquire an extensive understanding of our topic. Next, we identified the search terms for querying across different indexing sites. Once all papers were gathered, we manually filtered through out-of-scope papers by using the exclusion criteria and reading through the titles and abstracts. Finally, we rigorously analyzed the filtered list of papers to answer the research questions.

The research questions we examined in this mapping study are as follows:

- RQ1 What dynamic analysis practice can be used to visualize software architecture involving microservices?
- RQ2 What dynamic analysis tools currently exist?
- RQ3 What are the tools' features and properties?
- RQ4 What information can be extracted when visualizing architecture and how can that information be represented?

We used four major indexing sites and portals (indexers), namely IEEE Xplore, ACM Digital Library (DL), ScienceDirect, and SpringerLink. We tailored our search queries to look for papers related to visualizing architecture and the dynamic perspective. We divided our search query into three parts. In the first part of our query, we included the search term "visual" along with the related terms "dashboard", "graph", "view", and "model". In the second part, we used terms related to dynamic perspective including "dynamic", "run-time", and "runtime". For the last part, we added the term "microservice" to refine our results to only distributed systems. We avoided using terms such as "tracing", "process mining", and "logging" as these terms are specific methods for gathering information to visualize microservice-based systems. Instead, we wanted to keep our query broad in order to encompass all techniques for data extraction and visualization. The full search query is presented in Listing 1

[2]https://bit.ly/3Rz0lsC

TABLE 1. Related work research questions.

Reference	Research Questions
[7]	RQ1: What is the focus of software visualization research? RQ2: What is the maturity of the software visualization research field?
[8]	RQ1: What kinds of visualization techniques are mostly used in software evolution? RQ2: What is the main target of SEV? RQ3: What are the most used sources of information in SEV?
[9]	RQ1: What is the state-of-the-art in traceability approaches and tools between software architecture and the source code? RQ2: What information is available for traceability from architectural artifacts to more low-level artifacts (like the source code) and vice versa? RQ3: What empirical evidence has been reported in the field of traceability between software architecture and the source code? RQ4: In how far are the reported traceability relationships useful in software architecture understanding? RQ5: What are the reported benefits and liabilities of traceability approaches between software architecture and the source code? RQ6: What are the reported issues, barriers, and challenges of traceability between software architecture and the source code?
[10]	RQ1: What approaches have been adopted to recover traceability links between artifacts to support CIA RQ2: Which change impact sets have been covered? RQ3: How have studies adopted transitive tracing approaches to recover traceability links between artifacts?

and returned a total of 562 results across the indexing sites as shown in Table 2.

```
(" visual∗"OR "dashboard" OR "graph∗" OR "view" OR "model")
AND ("dynamic" OR "runtime" OR "run−time") AND "microservice"
```

Listing 1. Search query for the indexers.

To vet works based on their relevance, we used the following inclusion criteria:

1. Papers that involved discussion of process mining or log analysis
2. Papers that involved discussion of distributed tracing
3. Papers that reviewed existing dynamic analysis tools
4. Papers that introduced new dynamic analysis tools

We also filtered out non-relevant papers using the following exclusion criteria:

1. Papers that were in non-English languages
2. Papers without available full-text
3. Papers published before 2010
4. Papers that were not peer-reviewed
5. Papers that were duplicate results

Using the exclusion criteria to filter through papers first, the final result was 75 papers. After reading through all titles and abstracts of the ones remaining, we narrowed the set of papers down to 35. Then, we went through the related work section of the remaining papers to include relevant studies that the search query omitted. We included 5 papers through snowballing. The final step of the filtering process was to read fully through all remaining papers. This step resulted in 20 papers for the final total.

An overview of our search and filtering process is given in Figure 1. The number of papers returned by search queries for different indexers can be seen in Table 2. The complete list of filtered results is listed in Table 3 along with the papers we found by exploring related work sections. Once we narrowed down the relevant works to 20 papers, we studied them to discover dynamic analysis techniques to visualize architecture effectively along with tools to accomplish these practices.

Our full-text study process included matching the texts to our research questions and our inclusion criteria. If the text did not answer the research questions or match the inclusion criteria, it was discarded.

TABLE 2. Search query results for various index sites.

Indexer	Search Results	Filtered	Referenced	Total Relevant
ACM DL	15	3	1	4
IEEE Xplore	91	5	2	7
SpringerLink	276	4	-	4
ScienceDirect	180	3	-	3
Others	-	-	2	2
Total	562	15	5	20

To answer our first research question, we focused on identifying if a study focused on a specific dynamic analysis practice and how it was using that practice to visualize architecture. This research question coincided with our process mining and tracing inclusion criteria points because we knew both were defined dynamic analysis techniques. To answer our second and third research questions, we looked for specific tools that accomplished visualization through dynamic analysis. We made a distinction between new tools and old tools that would need an extension with our inclusion criteria. To answer our fourth research question, we searched for specific graphs or models that could successfully capture the information collected to provide visualization. We kept our study open to models or graphs that would need to be built upon as well.

To identify the dynamic tools available (RQ2), two authors looked individually for tools in the selected papers. In case of disagreements, a third author helped to discuss and to make a decision, obtaining a total of 8 tools. Moreover, we decided to look beyond the scope of research into what tools are being used in the industry. We used a Google search with the query ‘‘microservices visualization tool’’ to see what was available in the grey literature. Google returned 167 results. After the application of our inclusion and exclusion criteria, we narrowed down the list to 5 new tools. Similarly, the results were screened by two authors, with the help of a third author, to solve disagreements. We finally selected a total of 13 tools (8 from the selected publications and 5 from Google). Each visualization tool and its features can be seen in Table 6.

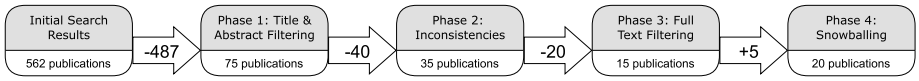

FIGURE 1. SLR process at each step.

TABLE 3. Final list of the 20 primary studies identified by search process.

Ref	Year	Title
[11]	2022	Microservices-based systems visualization: student research abstract
[4]	2021	A method for monitoring the coupling evolution of microservice-based architectures
[12]	2021	Automated Analysis of Distributed Tracing: Challenges and Research Directions
[13]	2020	Graph-based root cause analysis for service-oriented and microservice architectures
[14]	2021	Offline Trace Generation for Microservice Observability
[15]	2017	MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems
[16]	2017	Towards Recovering the Software Architecture of Microservice-Based Systems
[17]	2020	Graph-based trace analysis for microservice architecture understanding and problem diagnosis
[18]	2020	Architectural runtime models for integrating runtime observations and component-based models
[19]	2016	Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications
[20]	2022	Code Smell Prioritization with Business Process Mining and Static Code Analysis: A Case Study
[21]	2021	Enjoy your observability: an industrial survey of microservice tracing and analysis
[22]	2021	Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis
[23]	2019	Graph-based and scenario-driven microservice analysis, retrieval, and testing
[24]	2018	Using Service Dependency Graph to Analyze and Test Microservices
[25]	2018	Execution-Based Model Profiling
[26]	2017	Supporting Microservice Evolution
[27]	2010	Dapper, a large-scale distributed systems tracing infrastructure
[28]	2021	μViz: Visualization of Microservices
[29]	2021	Temporal Relations Extraction and Analysis of Log Events for Micro-service Framework

After the identification of the list of tools, we manually extracted the information to answer RQs 3 and 4.

V. RESULTS

Out of 562 papers returned by the search, there was a small number of relevant works available. The majority of the papers considered visualizing architecture theoretically, not practically. There was also a big focus on detecting anomalies, which is an advantage of visualizing architecture but not the main focus. Only 20 papers were considered for the final analysis (Table 3). This section presents the findings of the study by answering the research questions in separate subsections.

A. RQ1: CURRENT DYNAMIC ANALYSIS TECHNIQUES

To attain an architectural run-time view, one must use dynamic analysis. This section describes three different dynamic analysis techniques: process mining, distributed tracing, and monitoring. A brief description will be given as well as any advantages or disadvantages that arose during our research. Table 4 gives an overview of the dynamic analysis techniques and their corresponding references.

1) PROCESS MINING (LOG ANALYSIS)

One of the most common practices we found is process mining, also referred to as log analysis. It is a procedure that involves reading through event logs as the system is running and extracting information about the system. Mazak et. al. [25] defined three techniques in process mining: the discovery technique, the conformance checking technique, and the enhancement technique. The discovery technique is the process of extracting a process model directly from log data. The conformance-checking technique is the process of connecting an event log with an existing process model. The enhancement technique is the process of changing or extending a process model [25].

The procedure of analyzing log messages is relatively simple. Zuo et al. [29] describes the iterative partitioning log mining method (IPLoM). This method processes log messages vertically and aligns each term along a potential subset. Another method is looking for calls to other services. Finding and parsing these messages allows information to be extracted and then visualized into different models. One common visualization of process mining is the business process diagram, which will be discussed more in Section V-D [20]. Since many decentralized systems already use centralized

logging platforms, i.e., Splunk,[3] process mining tools are easy to set up. Additionally, since process mining orders log messages by time [25], a linear flow of the system from endpoint to endpoint can be derived (more often, however, using tracing), as well as bottlenecks within the flow. A drawback to this procedure is it can only find relations between services based on the event logs. Many services may not log every call made, meaning resulting visualizations may be incomplete. Along the same lines, many logs may not be detailed enough or contain information that is too low-level to provide more than a basic understanding of the functioning of the overall system [26]. As a result, a process mining approach can only be as good as the quality and quantity of the event logs.

2) DISTRIBUTED TRACING

Another common practice in dynamic analysis that can be identified in the current approaches is distributed tracing. Distributed tracing involves labeling a request with a unique identifier and using this identifier to follow the request as it moves throughout a system [12]. Since the request maintains its unique identifier in header information, it can be tracked as it moves through multiple services across many machines, like in a decentralized system.

Distributed tracing follows a five-step procedure of tracing and analysis. The five steps are logging, collection, preprocessing, storage, and analysis. The messages involving the trace or correlation ID are recorded and collected. They are preprocessed and then sent to storage where they are analyzed [21]. These traces can be used to visualize architecture dynamically as they capture system workflow and record services and dependencies while the system is running [12].

One issue that distributed tracing has encountered over the years is the lack of a standard protocol for collecting and sending data to tracing tools. To fill this gap, the OpenTracing[4] and OpenCensus[5] standards were created. OpenTracing is a vendor-neutral API that allows the sending of telemetry data to a tool like Jaeger[6] or Zipkin.[7] OpenCensus is a set of language-specific libraries that developers use to instrument their code for sending to tracing tools.[4] These standards combined to form OpenTelemetry,[8] which allows for a platform that developers can use to format their information to send to an observability tool. Once the information is sent to the tool, metrics, and visualizations can be extracted from the data.

Distributed tracing is a completely dynamic process in that it does not record data when the system is not running. This caveat is what puts tracing at a disadvantage. When visualizing services and the connections between them, distributed tracing only focuses on the components currently in use. To combat this disadvantage, tracing information would have to be stored continuously, and the system would need to be built upon over time. Processing in this way can quickly become a problem due to the sheer volume of tracing data [12], [17]. While this disadvantage exists, it does not outweigh the many advantages of distributed tracing. Keeping track of system traces when visualizing architecture allows for an extension of functionality. Traces can be used to identify high coupling, cyclic dependencies, and other inherent problems [12], [22]. Anomaly detection through tracing can be essential to developers as they are building their system, which makes tracing an integral part of system architecture reconstruction [30].

[3]https://www.splunk.com/
[4]https://opentracing.io
[5]https://opencensus.io
[6]https://www.jaegertracing.io/
[7]https://zipkin.io
[8]https://opentelemetry.io/docs/concepts/what-is-opentelemetry/

TABLE 4. Dynamic analysis techniques and corresponding references.

Dynamic Analysis Practice	References
Process Mining (Log Analysis)	[20], [25], [26], [29]
Distributed Tracing	[12], [17], [21], [22], [30][4]
Monitoring	[4], [13], [18], [19]

3) MONITORING

The last principal dynamic analysis practice that can be identified is monitoring. Monitoring is a detection process that can be used in two different ways. The first way is scanning for system metrics and dependencies as the system is running, similar to tracing [13]. The second way is detecting changes in the system and updating the system model accordingly.

The monitoring practice can capture multiple kinds of metrics in order to fully understand the system. Numerical metrics, such as the number of bytes sent between containers; categorical metrics, like names and labels; and ontological metrics, which represent the hierarchy of similar concepts, can all be captured using monitoring [13]. These metrics can be used to visualize connections between services and the activity that happens between them. They can also be used to detect future problems in the system, such as high coupling, and allow developers to fix those issues before they continue developing further [4].

System evolution is the most common use of the monitoring practice. Heinrich [18], [19] describes the iObserve approach, which involves the Monitor, Analyze, Plan, and Execute process (MAPE). By following this process, the system can automatically update the model as changes are made [19]. This feature can pose a problem if the developers cannot reconcile the new and old models. So, iObserve proposes a metamodel that can keep track of the old model and compare it to the new one [18]. Tracking changes at runtime is something that is unique to the monitoring approach and sets it apart from the other dynamic practices. Building a system architecture is a great start to visualizing architecture but monitoring changes and updating accordingly is also an important part of creating the system view.

B. RQ2: ARCHITECTURE VISUALIZATION TOOLS USING DYNAMIC ANALYSIS

Visualizing architecture from the dynamic perspective involves employing different practices, including log analy-

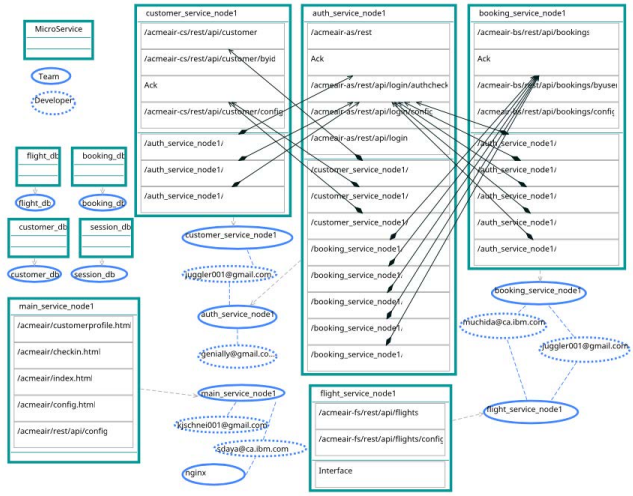

FIGURE 2. Logical architecture generated by microart [15].

sis, distributed tracing, and monitoring. In a practical sense, tools are needed to carry out these procedures and accomplish a true visualization of the architecture. In this section, examinations of some outstanding tools will be detailed.

1) MICROART

It is an architecture recovery tool that is capable of recovering both the physical and logical architectures of the system. The physical architecture is the overall abstraction, while the logical architecture is related to service discovery [15]. MicroArt divides its initial visualization process into three phases which are extraction, abstraction, and presentation.

The extraction phase is divided into static and dynamic analysis. The system name, its developers, and the service descriptors are retrieved with static analysis, while dynamic analysis extracts the communication logs and the container information [16]. The dynamic analysis is carried out by a monitoring tool that collects the information and a log analysis tool that filters it. The abstraction phase focuses on regrouping and filtering the information gathered in the extraction phase to map to MicroArt's domain-specific language (DSL) [16]. The presentation phase involves taking the components defined in the DSL and visualizing an architectural model. After modeling the initial architecture, MicroArt uses service discovery resolution to further refine the model and showcase the system's behavior in the logical architecture [16]. MicroArt accomplishes this by utilizing tracing to understand the system's actions at run-time. An example of the architecture generated by MicroArt can be seen in Figure 2.

MicroArt uses all three techniques discussed in Section V-A to capture a system's architecture. In the future, the developers plan to extend this tool by supporting the use of other tools in its processes. They have also expressed they will continue to test it on open-source platforms to further strengthen its capabilities [15].

2) DAPPER

It is a tracing program developed by researchers at Google. It was designed with low overhead, application-level transparency, and scalability in mind to provide ubiquitous deployment and continuous monitoring [27]. The approach Dapper uses provides a black-box view of the system without the need for additional annotations or application-level modifications.

Dapper organizes trace information into spans, trees, and annotations. A single span exists for every remote procedure call, and each span records annotations provided by Dapper's instrumentation library to implement a detailed trace. Support is available for the addition of custom annotations to record data that may be more specific to a certain system. Dapper implements its instrumentation by recording the context of callbacks in common libraries [27]. Span data is then written to local log files, pulled from each host by Dapper daemons, and written to a Dapper Bigtable repository [27]. The causal and temporal relations between spans are then plotted on a graph to illustrate the connections among services. The data is filterable by certain time windows and clusters to provide a fine level of control, and each trace is visible along with how long each trace took. A diagram of Dapper's process collection procedure can be seen in Figure 3.

Dapper is very low-level, which can be useful in determining exactly what procedure calls are bottlenecks in the system, but a higher-level abstraction would be better to have an understanding of the system as a whole. There are also cases where Dapper is unable to follow the control path directly, meaning manual trace parsing may be necessary [27]. However, Dapper's approach in combining multiple traces into a single repository is unique in the field and useful in troubleshooting a system with low overhead as an added benefit.

3) JAEGER AND ZIPKIN

Both are popular open-source distributed tracing systems. They are designed for mid-to-small-sized distributed systems [17]. The framework for both Jaeger and Zipkin is meant to be reusable and capture traces on a smaller scale [14]. The system stores the original trace information and visualizes a single trace at a time [17]. Jaeger creates a tree-like structure similar to the model produced by Dapper, while Zipkin implements its own data model [14]. An example of Jaeger's trace and span comparison graph can be seen in Figure 4. Both of these tools have been used several times in combination with other tools to visualize system architecture, but on their own, the volume of traces needed is too much for the system [12]. While both of these tools are great for tracing on a small scale, their functionality is not exactly what we are looking for in our study of visualizing architecture.

4) KIALI

It is a visualization platform for Istio service meshes. Multiple types of topology graphs to visualize the system are produced

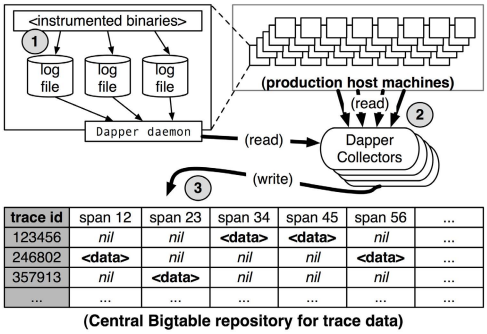

FIGURE 3. Dapper process collection [27].

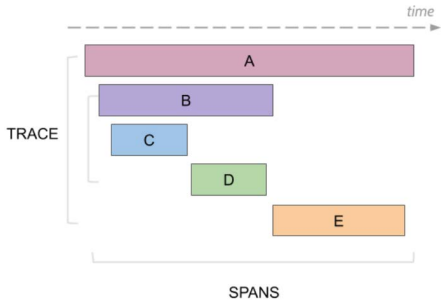

FIGURE 4. Jaeger generated trace/spans.

by Kiali as a means to provide developers with different levels of granularity. These include a high-level service topology, a low-level workload topology, and an application-level topology.[9] Each topology is organized by Kubernetes namespaces to display the most relevant information to developers. The flow of live traffic between services and the health of each service is also displayed by Kiali.

Kiali uses Istio data collected by Prometheus to produce its graphs.[5] Istio is an extension of Kubernetes used to aid with microservice deployment and operation. Information is provided to Kiali from tracing tools like Jaeger, container-level service data, health indexes, and other sources through the implementation of Istio proxies throughout the system that can intercept requests.[5]

Since Kiali is made to work on top of Istio and relies on information from Istio to construct a visualization, it cannot easily be extended to support other microservice-based systems. Additionally, while Kiali provides different levels of abstraction to view the system, all of the levels are fundamentally still dependency-graph views [28]. This means Kiali lacks visualization of more complex causally-related data that can be provided by tracing.

5) OTHER PUBLICLY AVAILABLE TOOLS

After extensive research, we have determined that there are few well-researched tools for architecture visualization in the dynamic perspective. In order to fully understand what instruments are available to practitioners and in addition to the systematic mapping study, we also performed a Google search using similar terms to Listing 1 to identify tools used in the industry.

One found microservice system visualization tool is DataDog. It is a monitoring and security platform for microservices that has a visualization tool built into it.[10] This platform allows for microservices to be visualized and monitored in real-time. Changes are constantly being tracked, and the graph is subsequently updated. This tool is focused more on identifying problems in efficiency or during run-time, but visualization is an important aspect of it.

ExplorViz is a visualization tool with the ability to capture live traces and generate two different views from their analysis, the landscape-level and the application-level [31]. The landscape-level view provides an overall look at the system, while the application-level view focuses more on one specific application or service and its properties. ExplorViz is completely open-source and has been used in several projects within the industry, including PPI AG and Adesso SE.

Another open-source tool available to visualize microservices is Mosaic. Mosaic generates a service map that allows for analysis of the health of the services and identification of dependencies.[11] This tool also can track custom events executed by the services and record them for analysis. Ortelius[12] is similar to Mosaic because it generates a dependency graph. Ortelius can also keep track of versions for both the entire system and individual services through the use of its proactive visualization maps.[10] These maps are able to provide a logical view of the system by which it can be shared, versioned, and extended.[10]

C. RQ3: TOOLS' FEATURES AND PROPERTIES

To adequately compare found tools and determine their unique features and properties, we constructed a set list of features to compare the tools across. We focused on two aspects, the reconstructed view and the recovery technique. For the view, we were looking for physical and logical architecture recovery and service identification. We denoted whether each tool utilized static or dynamic analysis to differentiate between what kind of data could be visualized. Process mining and distributed tracing were also included as features to classify tools by the dynamic analysis techniques they used. Lastly, we included whether a tool is open-source. Through this research, we discovered that many tools focused on distributed tracing as their dynamic analysis practice, and process mining was less common. Physical architecture recovery was also more common than logical architecture recovery. Overall, most tools were able to accomplish physical architecture recovery and service identification through the use of dynamic analysis. Table 6 compares each found tool across a collection of features discussed above.

[9] https://kiali.io/
[10] https://www.datadoghq.com/microservice-visualization/
[11] https://bit.ly/3c1zipD
[12] https://ortelius.io/blog/2021/03/26/microservice-monitoring-and-visualization/

TABLE 5. Tool comparison across list of analysis approaches (RQ1).

Tools	Analysis			References	
	Distributed Tracing	Process mining (Event log analysis)	Monitoring	Link to Project Source Repository	Paper
MicroArt	✓	✓	✓	https://github.com/microart/microART-Tool	[15], [16]
Dapper	✓		✓		[27]
Jaeger	✓		✓	https://github.com/jaegertracing/jaeger	[12], [14], [17]
Zipkin	✓		✓	https://github.com/openzipkin/zipkin	[12], [14], [17]
Kiali	✓		✓	https://github.com/kiali	[28]
μViz	✓		✓	Still being developed. No public repository yet.	[28]
Splunk APM	✓		✓		[21]
AWS X-Ray	✓		✓		[21], [22], [26]
DynaTrace	✓		✓		[13], [21]
DataDog	✓	✓	✓	https://github.com/DataDog	
ExplorViz		✓	✓	https://github.com/ExplorViz	[31]
Mosaic		✓	✓	https://github.com/oslabs-beta/mosaic	
Ortelius	✓	✓	✓	https://github.com/ortelius/ortelius	

TABLE 6. Tool comparison across list of features.

Features	Tools						
	MicroArt	Dapper	Jaeger	Zipkin	Kiali	μViz	Splunk APM
Service Identification	✓	✓	✓	✓	✓	✓	✓
Physical Architecture Recovery	✓		✓	✓	✓	✓	✓
Logical Architecture Recovery	✓				✓	✓	
Business Process Model	✓	✓	✓	✓	✓		✓
Service Dependency Graph		✓			✓		✓
Service Endpoint Call Graph	✓	✓		✓			
Platform Independent	All, Docker instrumentation	All, language independent	C#, C++, Go, Java, Python, Node.js	C#, Go, Java, JavaScript, Ruby, Scale, Ruby	No. Requires Istio	No, too early in development	All, stack independent
References	[15], [16]	[27]	[12], [14], [17]	[12], [14], [17]	[28]	[28]	[21]
	AWS X-Ray	DynaTrace	DataDog	ExplorViz	Mosaic	Ortelius	
Service Identification	✓	✓	✓	✓	✓	✓	
Physical Architecture Recovery		✓	✓	✓	✓	✓	
Logical Architecture Recovery		✓	✓			✓	
Business Process Model	✓	✓	✓		✓		
Service Dependency Graph		✓	✓		✓		
Service Endpoint Call Graph	✓	✓				✓	
Platform Independent	No. Requires trace data from AWS instance	600+ supported technologies	.Net, Go, Java, Node.js, PHP, Python, Ruby, Finagle	All, Docker instrumentation	No, too early in development	All, Docker instrumentation	
References	[21], [22], [26]	[13], [21]		[31]			

D. RQ4: EXTRACTING INFORMATION WITH ARCHITECTURE VISUALIZATION

Through dynamic practices, such as process mining, distributed tracing, and monitoring, information about a distributed system can be collected. Using the architecture visualization tools previously discussed, this information can be captured in different types of models and graphs. This section will describe which models can be visualized in this way and what benefits each model offers, thus answering our fourth research question. Table 7 gives an overview of the models and graphs and their corresponding references.

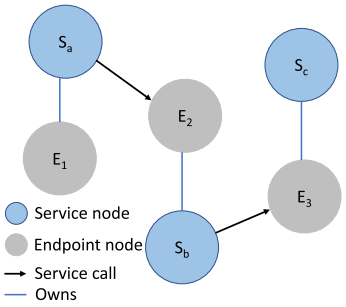

FIGURE 5. Service dependency graph.

1) SERVICE DEPENDENCY GRAPH

During our study, we found the most commonly extracted model is the service dependency graph (Figure 5). It visualizes the microservices and the connections between them, which are also known as dependencies. This kind of model can be extracted using information obtained from log analysis, distributed tracing, or monitoring. The service dependency graph is an important step in a process known as GMAT (Graph-based Microservice Analysis and Testing) [24]. After creating this model, cyclic dependencies can be detected, and microservice retrieval can be enabled to reuse services rather than creating new ones [23], [24]. The advantage of a model like this is it is easy to understand and generate. It also allows the opportunity for extension, as described below.

Abdelfattah [11] describes a way to capture this graph in a 3D environment. By implementing this 3D model, the developers would be able to see the system from different angles and move the services around to create different topologies [11]. In this proof of concept, three levels are

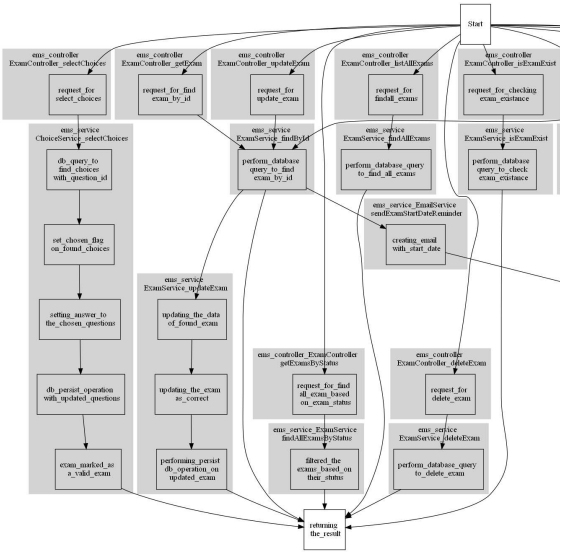

FIGURE 6. Business process model [20].

implemented: the system level, the service level, and the function level. The system level is a high-level overview of the entire system. The service level shows the perspective of the system from the services side. The function level attempts to visualize a UML communication diagram [11]. This study is one case of taking the service dependency graph and extending it to capture more information about the system.

2) BUSINESS PROCESS MODEL

This model is a graph describing business processes using information obtained from process mining [20]. The model follows multiple paths through the system, capturing the remote calls to other services. The paths converge at the end of the graph, producing one result. Process mining is used to extract the information needed to form this graph. By studying the origin and content of log messages, a particular path can be captured through the system [20]. This graph has an advantage over distributed tracing because it can capture multiple paths, while tracing can only capture one. The business process model helps to capture dependencies as well as detect anomalies, which is a key point of visualizing architecture. Figure 6 is an example of a business process model.

3) SERVICE ENDPOINT CALL GRAPH

The service endpoint call graph, also referred to as the microservice call graph, resembles a tree, with each node being a service endpoint and each edge being a remote call [22]. These kinds of graphs are generated through distributed tracing. They do not capture the system as a whole, but they can be used to partially construct the full system architecture. The more detailed view allows developers to get a low-level look at the system, which allows for anomaly detection on a smaller scale [22].

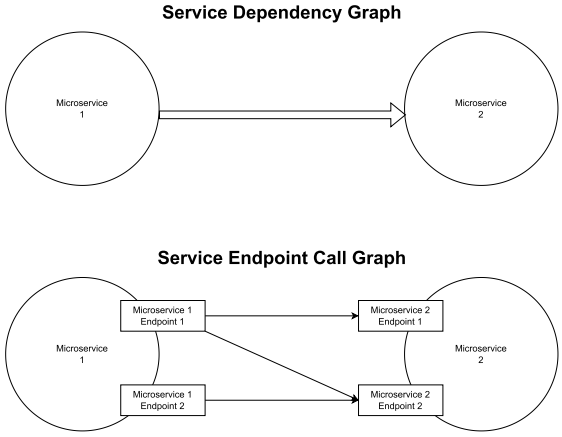

FIGURE 7. Service dependency graph versus service endpoint call graph.

4) DIFFERENCES

The service dependency graph, business process model, and service endpoint call graph are all ways of visualizing microservice architecture through dynamic analysis. What separates these visualizations? The service dependency graph focuses more on the overall visualization of the system. It showcases services and their dependencies through connections between nodes. Through the use of tracing and log analysis, this graph can be captured. The business process model and service endpoint call graph both focus more on individual traces with more information about the remote calls. The business process model is more equipped to handle complicated traces through multiple services, while the service endpoint call graph seems more suited for the smaller scale. The service endpoint call graph is generated from tracing, while the business process model is a result of log analysis. The log analysis approach the business process model takes is more suited to capturing messages than the service endpoint call graph, which depicts REST calls. The service endpoint call graph considers the dependencies between each endpoint in a microservice whereas the service dependency graph considers the microservice as a whole. Figure 7 illustrates this distinction. In order to capture an entire visualization with current behavior included, combining or extending these representations is the logical next step.

TABLE 7. Models/Graphs and corresponding references.

Service Dependency Graph	Business Process Model	Service Endpoint Call Graph
[11], [23], [24]	[20]	[22]

VI. GAPS AND CHALLENGES

This section outlines challenges uncovered by other researchers and offers potential solutions for future tools to provide a more complete and comprehensive visualization of microservice-based systems.

Heinrich [18] found that existing tools lack the ability to update as the system undergoes structural changes. Systems may change during run-time for many reasons, such as

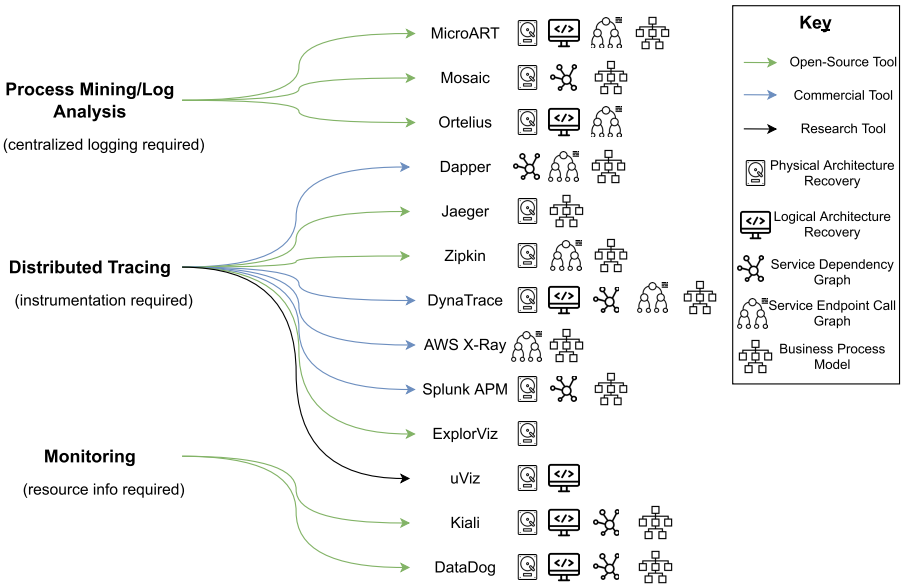

FIGURE 8. Summary of dynamic analysis approaches for microservices, mapped to identified tools providing architectural visualization.

workload balancing or migration, replication, or deallocation of a container. Without a dynamic visualization, modeled and actual views of the system may quickly grow inconsistent.

Accordingly, tools should use real-time data to construct a visualization of the system as it changes or evolves. Such a tool could be useful for troubleshooting and root-cause analysis. Likewise, another interesting angle could visualize how a proposed change to the system would affect other system parts. This change impact analysis can help developers debug potential issues with an architecture modification before deployment of the new architecture.

Silva et al. [28] found that many available tools to reconstruct and visualize software architecture focus on providing a service dependency graph but do not take advantage of more complex data. Oftentimes, different perspectives are needed to get insight into the system beyond dependency graphs.

More enhanced visualizations are needed to utilize complex, causally-related data and offer more intuitive designs. A few examples might be a visualization capable of switching between different architectural views (perhaps combined with static analysis), using a 3D space to better fit complex systems, or creating interactive graphs that can be filtered and searched through to better manage a large amount of information. A tool capable of providing these functionalities would be scalable to support the visualization of large architectures with little added difficulty, flexible to supply varying points of view of a system, and useful to quickly find the relevant information needed.

Granchelli et al. [16] recommended the development of component resolution modules in order to visualize non-container system parts like databases, load balancers, and logging services. To be able to record and visualize interactions with non-service system parts would make visualizations more complete.

Another avenue is to combine dynamic and static analysis. MicroART begins to do this by combining dynamic analysis with service names and descriptors found by analyzing the codebase [16]. However, static analysis has a lot more potential. Cerny et al. found that static analysis is much better for understanding the system before deployment [6]. Since this is the case, the static analysis will not compete for processing power with the system during runtime and can be used to show the impact of a code change before it is implemented. Another paper introduced Microvision [32] to visualize microservices using static analysis. It was able to generate a call graph in a three-dimensional AR virtualization. Combined static and dynamic methods to generate visualizations would yield a more reliable perspective beyond what is visible at runtime.

VII. DISCUSSION

The analysis of the literature enabled us to identify three dynamic analysis techniques (process mining, distributed tracing, and monitoring) to reconstruct and visualize the system architecture. Figure 8 summarizes the analysis approaches, mapped to assessed tools, and provided architectural views. From the analysis of the literature, we were also able to distill different observations and provide recommendations.

A. RECOMMENDATIONS TO SYSTEM DEVELOPERS

It is obvious that systems produce logs, but it should be a common best practice that each microservice system utilizes tracing. This might require additional investment into an API gateway or the use of a service mesh; however, the benefits

outweigh the investment when it comes to debugging the system and understanding dependencies of log statements that interweave in time across services. Thus, with telemetry vendor-neutral APIs like OpenTracing and OpenCensus, such investment becomes feasible. Considering between plain event logs and tracing should no longer be a question. Building on such infrastructure enables many introduced tools to determine visualized system-centered perspective. In addition, monitoring can add valuable detail to such a perspective.

B. RECOMMENDATIONS TO TOOL DEVELOPERS

Our recommendation is that both monitoring and distributed tracing should be considered in future tools. On the other hand, we believe that approaches based on plain logs without tracing IDs are outdated for microservices and should not be considered in future research. The benefits of dynamic analysis are they come in the platform-agnostic format as opposed to static analysis. However, for tracing, we still need to add libraries to the microservice code to parse, embed, and use the tracking ID. Platforms like Spring Cloud makes it somewhat easy, but this is not yet the case for all platforms. Similarly, monitoring can produce basic statistics; however, for more details, we might need to influence the service code.

Throughout this study, we have named tools that implement these dynamic analysis techniques. Most of the tools use either a separate Docker container to instrument the system from its own independent component or have a library that must be included. The most commonly used languages for these libraries are C#, Java, Go, Python, Ruby, and JavaScript. Both of these approaches, as long as the library supports numerous languages, make setting up these tools fairly easy and straightforward as well as relatively platform-independent.

1) CHOOSING THE RIGHT VISUALIZATION

The choice of a particular visualization should be driven by the problem, perspective, or concern we target. We cannot conclude that there is an obvious winner in what assessed research literature presented. However, the most common visual perspectives are business process models, models of system topology, and service dependency models. These are also easy to determine and are often provided as a by-product of tools meant for distributed tracing and debugging.

The choice of the right visualization corresponds to the anticipated system architecture description. There are different views to describe software architecture, and all are important. Among examples are the domain view, the service view, the topology, the used technology view, and data and control flows. It might be important for domain experts to analyze security or persistence, and they might need to recognize logical components, services, the deployment topology, etc. Considering that dynamic analysis sees microservices or containers as black boxes, we cannot demand for unrealistic expectations, such as a view with internal details. It is the task of static analysis to provide the internal perspective. At the same time, static analysis cannot describe how the system is used because such data does not exist.

2) KEEPING IN MIND WHO IS THE TARGET USER

There is another perspective relevant to microservices - separation of duty. Microservice developers do not manage telemetry data or are the primary consumer of the visualization, but the DevOps engineers are. DevOps might not know anything about the internal details of microservices, but they deploy and monitor them, and they are the primary consumers of visualizations described in this paper. Obviously, developers and especially system engineers need to know about the system-centered perspective, and thus they interact with DevOps. Architects prescribe the architecture and microservice specifics to developers, and these models might be one of the few instruments they have these days to analyze whether the system was designed as intended.

C. REALISTIC EXPECTATIONS

Finally, it holds for dynamic analysis - with no run-time data, there is no visualization. The results can only be as good as the quality of the run-time data [26]. This does not necessarily mean the quality with which we log and collect metrics. The entire process is dependent on system execution, typically driven by run-time interaction. If we deploy a system with ten features and only one is used, we will not see the entire picture. Similarly, the required interaction and log generation introduce a considerable delay between when a change was introduced by developers and when we can detect the new code through the trace log. While one can argue that quality testing assists with preventing faulty deployment to production, we must consider that microservice systems can be really complex and what it implies for the testing infrastructure and its evolvability.

VIII. THREATS TO VALIDITY

The main threat that can arise when performing a mapping study relates to the inclusion and exclusion of relevant studies. It is important to ensure that the works considered are applicable to our research questions and that relevant papers were not excluded from our study.

Several steps throughout the literature review process were taken to mitigate this threat. Firstly, we have followed standard guidelines and practices outlined by similar mapping studies in our field (see section 3). Additionally, we tested 37 different search queries with our four indexers and examined the results of each query to determine how applicable the results were to our research. Our final query was broad enough in scope to encompass multiple methods for dynamic analysis and proved to return the most applicable results while keeping unrelated papers to a minimum. Finally, each paper was reviewed by multiple reviewers to minimize any bias or error of any one reviewer during the filtration and extraction process.

IX. CONCLUSION

In this study, we have analyzed how creating and maintaining a unified view of a decentralized system is possible through the use of dynamic analysis. There are three main dynamic analysis techniques: process mining, distributed tracing, and monitoring. All three practices focus on collecting data at runtime and creating visualizations by analyzing that data. There are several well-researched tools available to accomplish these practices, such as MicroArt, Dapper, Jaeger, Zipkin, and Kiali. In the industry, more tools exist, like DataDog and ExplorViz. It was a common theme that these tools use distributed tracing to visualize architecture, while monitoring and process mining was less common. Once the data is collected and analyzed, different models or graphs can be captured, such as a service dependency graph, business process model, or service endpoint call graph. These graphs visualize different things, but the main focus of all is that the main architecture is captured while the behavior of the components is also shown. Visualizing both the system and its behavior is an instrumental part of software development, and dynamic analysis is a defined way of accomplishing that.

In future work, we plan to develop a new prototype visualization involving hierarchical navigation, various views, and rendering in 3D space.

REFERENCES

[1] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, "Software architecture degradation in open source software: A systematic literature review," *IEEE Access*, vol. 8, pp. 173681–173709, 2020.

[2] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *J. Syst. Softw.*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[3] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Inf. Softw. Technol.*, vol. 64, pp. 1–18, Aug. 2015.

[4] D. R. F. Apolinário and B. B. N. de França, "A method for monitoring the coupling evolution of microservice-based architectures," *J. Brazilian Comput. Soc.*, vol. 27, no. 1, p. 17, Dec. 2021.

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

[6] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice architecture reconstruction and visualization techniques: A review," in *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Aug. 2022, pp. 39–48.

[7] A.-L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väätäjä, "Software visualization today: Systematic literature review," in *Proc. 20th Int. Academic Mindtrek Conf. (AcademicMindtrek)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 262–271. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/2994310.2994327

[8] H. B. Salameh, A. Ahmad, and A. Aljammal, "Software evolution visualization techniques and methods—A systematic review," in *Proc. 7th Int. Conf. Comput. Sci. Inf. Technol. (CSIT)*, Jul. 2016, pp. 1–6.

[9] M. A. Javed and U. Zdun, "A systematic literature review of traceability approaches between software architecture and source code," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng. (EASE)*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–10. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/2601248.2601278

[10] T. W. W. Aung, H. Huo, and Y. Sui, "A literature review of automatic traceability links recovery for software change impact analysis," in *Proc. 28th Int. Conf. Program Comprehension (ICPC)*. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 14–24. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/3387904.3389251

[11] A. S. Abdelfattah, "Microservices-based systems visualization: Student research abstract," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput. (SAC)*. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1460–1464.

[12] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *J. Grid Comput.*, vol. 19, no. 1, p. 9, Feb. 2021.

[13] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *J. Syst. Softw.*, vol. 159, Jan. 2020, Art. no. 110432.

[14] D. Ernst and S. Tai, "Offline trace generation for microservice observability," in *Proc. IEEE 25th Int. Enterprise Distrib. Object Comput. Workshop (EDOCW)*, Oct. 2021, pp. 308–317.

[15] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "MicroART: A software architecture recovery tool for maintaining microservice-based systems," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, Apr. 2017, pp. 298–302.

[16] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, Apr. 2017, pp. 46–53.

[17] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1387–1397.

[18] R. Heinrich, "Architectural runtime models for integrating runtime observations and component-based models," *J. Syst. Softw.*, vol. 169, Nov. 2020, Art. no. 110722.

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

[20] M. R. Islam, A. Al Maruf, and T. Cerny, "Code smell prioritization with business process mining and static code analysis: A case study," *Electronics*, vol. 11, no. 12, p. 1880, Jun. 2022.

[21] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," *Empirical Softw. Eng.*, vol. 27, no. 1, p. 25, Nov. 2021.

[22] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput. (SoCC)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 412–426.

[23] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Gener. Comput. Syst.*, vol. 100, pp. 724–735, Nov. 2019.

[24] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2018, pp. 81–86.

[25] A. Mazak, M. Wimmer, and P. Patsuk-Bösch, "Execution-based model profiling," in *Data-Driven Process Discovery and Analysis*, P. Ceravolo, C. Guetl, and S. Rinderle-Ma, Eds. Cham, Switzerland: Springer, 2018, pp. 37–52.

[26] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 539–543.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Mountain View, CA, USA, Tech. Rep. dapper-2010-1, 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[28] S. Silva, J. Correia, A. Bento, F. Araujo, and R. Barbosa, "μViz: Visualization of microservices," in *Proc. 25th Int. Conf. Inf. Vis. (IV)*, Jul. 2021, pp. 120–128.

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

[30] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," in *Information Science and Applications*, H. Kim, K. J. Kim, and S. Park, Eds. Singapore: Springer, 2021, pp. 223–234.

[31] W. Hasselbring, A. Krause, and C. Zirkelbach, "ExplorViz: Research on software visualization, comprehension and collaboration," *Softw. Impacts*, vol. 6, Nov. 2020, Art. no. 100034.
[32] 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 *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Aug. 2022, pp. 49–58.

MIA E. GORTNEY is currently a Junior Student in computer science from Baylor University. Since Fall 2020, she has been featured on the Baylor University Dean's List. Her research interests include static and dynamic code analysis and visualization. In addition, she received the Computer Science Scholarship Award, in Spring 2022.

PATRICK E. HARRIS is currently pursuing the bachelor's degree in computer science with Baylor University. He has been recognized on the Baylor University Dean's List and with scholarship awards, including the Baylor Computer Science Scholarship and the Baylor Association of Computing Machinery Scholarship. His research interests include the visualization of distributed systems and the security of computer systems.

TOMAS CERNY received the master's and Ph.D. degrees from the Faculty of Electrical Engineering, Czech Technical University in Prague, and an M.S. degree from Baylor University.

He is currently a Professor of computer science with Baylor University. His research interests include software engineering, cloud systems, and code analysis. In 2009, he started his academic career at the Czech Technical University, FEE, from where he transferred to Baylor University, in 2017. He served more than ten years as the lead developer of the International Collegiate Programming Contest Management System. He authored over 100 publications, mostly related to code analysis and enterprise systems. Among his awards are best papers at Microservices 2022, IEEE SOSE 2022, Closer 2022, LXNLP 2022, the Outstanding Service Award ACM SIGAPP, in 2015 and 2018; or the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He served on the committee of multiple conferences in the past few years, including program or conference chairs at ACM SAC, ACM RACS, and ICITCS.

ABDULLAH AL MARUF (Member, IEEE) received the bachelor's degree from the Department of Computer Science and Engineering, Chittagong University of Engineering and Technology, Bangladesh. He is currently pursuing the degree in computer science with Baylor University. He has four years of professional experience as a software developer and a DevOps engineer. He is an Open-Source Enthusiast. His research interests include software engineering, code analysis, and runtime log analysis.

MIROSLAV BURES leads the System Testing Intelligent Laboratory (STILL), Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague. In 2010, he was appointed at the Czech Technical University in Prague, where he is currently an Associate Professor of computer science. His research interests include quality assurance and reliability methods, model-based testing, path-based testing, combinatorial interaction testing, and test automation for software, the Internet of Things, and mission-critical systems. He leads several projects in the field of test automation for software and Internet of Things systems, covering the topics of automated generation of test scenarios as well as automated execution of the tests.

DAVIDE TAIBI is currently a Full Professor with the University of Oulu, Finland, where he is the Head of the M3S Cloud Research Group. His research interests include empirical software engineering applied to cloud-native systems, with a special focus on the migration from monolithic to cloud-native applications. He is investigating processes and techniques for developing Cloud Native applications and identifying cloud-native-specific patterns and anti-patterns. He has been a member of the International Software Engineering Network (ISERN), since 2018. Before moving to Finland, he has been an Assistant Professor with the Free University of Bozen/Bolzano (2015–2017), a Postdoctoral Research Fellow at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering—IESE (2013–2014), and a Research Fellow at the University of Insubria (2007–2011).

PAVEL TISNOVSKY received the Ph.D. degree from the Brno University of Technology, Czech Republic. He was an Assistant Professor, from 1999 to 2005. He is currently the Principal Quality Engineer with Red Hat, Inc., with over ten years of experience. He is a Programming Language Enthusiast and the author of many articles and series at Linux magazine ROOT.cz. He holds one software patent on testing and currently works on tools for OpenShift.io—open development services for creating, building, and testing container applications.