Using Microservice Telemetry Data for System Dynamic Analysis

1st Abdullah Al Maruf
Department of Computer Science
Baylor University
Waco, Texas, United States
maruf_maruf1@baylor.edu

2nd Alexander Bakhtin
Software Engineering Group
Tampere University
Tampere, FI-33720, Finland
alexander.bakhtin@tuni.fi

3rd Tomas Cerny
Department of Computer Science
Baylor University
Waco, Texas, United States
Tomas_Cerny@baylor.edu

4th Davide Taibi
Software Engineering Group
Tampere University
Tampere, FI-33720, Finland
davide.taibi@tuni.fi

***Abstract*—Microservices bring various benefits to software systems. They also bring decentralization and lose coupling across self-contained system parts. Since these systems likely evolve in a decentralized manner, they need to be monitored to identify when possibly poorly designed extensions deteriorate the overall system quality. For monolith systems, such tasks have been commonly addressed through static analysis. However, given the decentralization and possible language diversity across microservices, static analysis tools are lacking. On the other hand, there are available tools commonly used by practitioners that offer centralized logging, tracing, and metric collection for microservices. In this paper, we assess the opportunity to combine current dynamic analysis tools with anomaly detection in the form of quality metrics and anti-patterns. We develop a tool prototype that we use to assess a large microservice system benchmark demonstrating the feasibility and potential of such an approach.**

***Index Terms*—Microservices, Software Architecture Reconstruction, Dynamic Analysis, Telemetry Data**

I. Introduction

Microservice architecture (MSA) has become the industry standard. MSA allows for efficient scaling, improved resiliency, and faster software delivery because of its loosely connected nature. It also enables teams to focus on a single task/module and produce reliable software faster by allowing them to use any programming language that suits the task. With the ability of agility and speed of development, MSA needs constant monitoring and analysis to tackle the complexity of the system. A microservice system's longevity and quality can be jeopardized without regular monitoring.

The system can be subjected to static and/or runtime analysis. While static analysis can detect bugs, vulnerabilities, and smells [1], [2] before putting software into production, it also has the disadvantage of requiring language-specific analysis. It is challenging to find language-specific static analyzer tools unless the modules are written in a few popular languages. Otherwise, the developers need to create their tool, which will increase maintenance workload, or they must exclude those modules from analysis, which is also not a good option. Runtime analysis can help solve this language-specific issue. Telemetry data is one form of dynamic analysis which provides a few key analysis perspectives, such as Service Dependency Graph (SDG), detecting architectural smells, and architectural evolution using the SDG.

This paper uses telemetry data to determine inter-service communication and build the Software Dependency Graph. Next, it uses this information to find architecture smells, calculate anti-pattern metrics automatically, detect architectural evolution, and effectively scale the microservice modules. This can help system operators and engineers better understand the system specifics and concerns that should be addressed throughout the system evolution.

The following is a breakdown of the paper's structure: Section II discusses static and dynamic analysis and prior studies on the analysis of microservice systems. Then Section III goes over various types of telemetry data and their applications. Section IV proposes a method for generating SDGs from telemetry logs. We discuss possible analysis directions in Section V and showcase a case study in Section VI. We conclude the paper with Section VII.

II. Background & Related Work

The static analysis uses software artifacts such as source code, deployment manifests, and API documentation. Dynamic analysis is the real-time testing or profiling of a system. Dynamic analysis can be applied to runtime data from a system in either a production or a staging/development environment. Users must use the system, or a script must replicate real-time user behavior, such as accessing all use cases and making reasonable user requests per minute, to generate runtime data. Data obtained in runtime includes application logs and telemetry data. Both these approaches have their specifics and

This material is based upon work supported by the National Science Foundation under Grant No. 1854049, grant from Red Hat Research, and Ulla Tuominen (Shapit).

suitability for certain tasks. However, certain overlaps exist, such as the aspects we address in this paper.

Detecting bad smells and poor design indicators in a decentralized environment have been addressed in the literature. For instance, Taibi et al. [3] identified recurrent smells in microservices. There has been approached by Walker et al. [1], or Pigazzini et al. [4] in order to detect them. Both these approaches involved static analysis or a combination of static and dynamic analysis, which is currently limited to a specific platform. The major challenge is to reconstruct the holistic system perspective. A perspective that makes it obvious for practitioners to understand how the system divides into specific microservices and how these interact and depend on one another. Rademacher et al. [5], Walker et al. [6] or Granchelli [7] considered the process software architecture reconstruction for microservices. While it can be performed manually from various artifacts [5], combining static and dynamic analysis [7] or an automated static analysis approach using distributed codebase is also viable [6] as demonstrated by Bushong et al. [8]. The major challenge with static analysis is the code analysis language dependency and distribution. While the initial approach to address this challenge has been proposed [9], the tooling support is not yet available.

The tooling support is broader in the realm of dynamic analysis. Common tools exist for centralized logging, distributed tracing, and telemetry. In addition, various system-centric perspectives are available in such tools (i.e., Jaeger, Kiali, etc.), and it is possible to preview trace-reconstructed system topology or dependency graphs. While these views cannot be as detailed as when assessing the code, it gives sufficient abstraction on the running system and language agnosticism. Given the broad availability of these tools, it is reasonable to consider the integration of detection of various patterns [10], anti-patterns [11], smells [3], or quality metrics (e.g. Coupling [12]) using dynamic analysis. While it might be assumed that development and operations (DevOps) engineers can easily see these indicators, Bento et al. [13] suggested that the current tools do not provide appropriate ways to abstract, navigate, filter, and analyze tracing data and do not automate or aid with trace analysis. Instead, the process relies on DevOps, but these might lack the expertise or the time necessary to determine the statistics in the ever-changing environment.

When using traces, it can be seen that SDG is commonly used. For instance, Ma et al. [14] use it to analyze and test microservices through graph-based microservice analysis and testing. However, Ma et al. made the process dependent on DevOps manual efforts to detect anomalies by analyzing risky service invocation chains and tracing the linkages between services.

In this paper, we look into the automation of quality metric detection using dynamic analysis of telemetry data, which would indicate the areas of concern for DevOps and let them prioritize their tasks.

III. Telemetry Data & Its Applications

In software engineering, telemetry data refers to collecting data from software and systems that indicate the source's state. Telemetry is one of the key factors in increasing the observability of a complex system [15]. Johnson et al. [16] defined software project telemetry as a method of defining, collecting, and analyzing software metrics that has the five characteristics listed below:

- Data should be collected automatically, with no manual intervention from humans.
- A timestamp must be assigned to each event in the data.
- Every project member has continuous and immediate access to the data.
- Telemetry analysis should be valuable even if it lacks complete data for the entire project's lifespan.
- Telemetry analyses depict the project's current state and how it evolves over time.

Opentelemetry [17] is the standardization of the telemetry data collection process. It is difficult for developers to switch tools and adapt to a new tool because of the different standardization prior to Opentelemetry. Opentelemetry provides a vendor-agnostic API for sending telemetry data to a backend and a set of language-specific libraries for instrumenting code and shipping data to one of the supported backends.

Telemetry data can be categorized into three main formats [15], [17], [18]:

- Traces
- Metrics, and
- Logs.

Karumuri et al. defined metrics as numeric data, logs as unstructured strings, and traces as a graph of a request's execution path [15]. Along with the available libraries, developers can add software-specific metrics to be shipped to the backend, such as Prometheus. For example, if there are many unsent emails in the queue. In that case, developers can ship the number of unsent emails to Prometheus and then use Alertmanager to trigger an alert. Grafana is a popular tool for visualizing metrics, and developers can use a shared grafana dashboard, a platform for sharing custom dashboards that use critical metrics.

There are a few tools that can be used to trace a request in order to analyze it. Some examples of such tools are Zipkin and Jaeger. We can use tracing to find the source of slow response and keep track of distributed transactions. The majority of tracing tools also provide a software architecture graph in the form of a service dependency graph.

Although there are a few tools for tracing, metrics analysis, and visualization, there is little for log analysis. Most developers use ELK (Elasticsearch, Logstash, and Kibana) to store and process a distributed system's log centrally. Elasticsearch stores large amounts of data while also allowing for faster data searches. Logstash serves as a log aggregator and processor, reading logs and sending them to Elasticsearch. Kibana allows users to run queries against log data. It is common to combine

tracing and logs so that developers can search for error logs with the request's traceID across the entire distributed system.

Even though telemetry requires a language-specific SDK library to produce metrics or traces, service mesh adds this capability without requiring the addition of a library to the source code. This eliminates the need for source code changes and the language barrier. In addition to tracing, service mesh includes metrics and access logs by default.

IV. Service Mesh & Service Dependency Graph (SDG)

Distributed systems are notoriously difficult to manage, and adding traffic management to the mix makes things even more difficult. A service mesh comes with traffic management, observability, security, encryption, access control, rate limiting, and other features out of the box. As a result, service mesh has become an essential tool in Kubernetes-based systems. Each service module is deployed separately in this system, and each of these modules has a sidecar [19] proxy that receives and sends traffic on behalf of the microservice module (shown in figure-1). These proxy sidecars can control traffic and improve the service's observability and security.

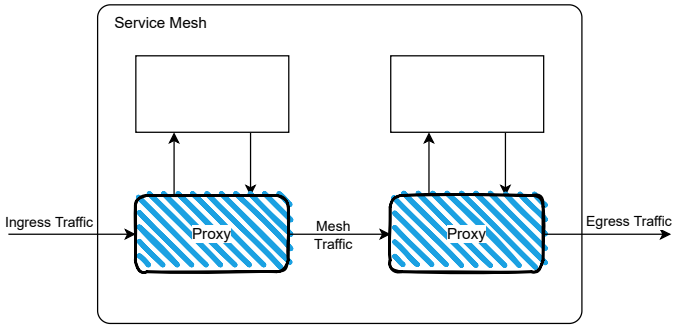

Fig. 1: Architecture of a service mesh where proxy sidecars handles all the network activity of a service.

Istio [20], linkerd[1], and consul connect[2] are the most popular open source service mesh tools, according to GitHub stargazers. In addition to these tools, most service mesh tools provide an access log (also known as audit logging or proxy log), containing information about each inbound and outbound request in a customizable format. Our method uses these access logs to determine which service is calling which service, which endpoint is being called, and how frequently a service is called in a given time period. We can use this information to reconstruct software architecture and automatically detect anti-patterns and coupling.

Listing 1: Default Format of Access Logs in Istio service mesh

```
[%START_TIME%] \"%REQ(:METHOD)% %REQ(X-ENVOY-
    ORIGINAL-PATH?:PATH)% %PROTOCOL%\" %
    RESPONSE_CODE% %RESPONSE_FLAGS% %
    RESPONSE_CODE_DETAILS% %
    CONNECTION_TERMINATION_DETAILS%
\"%UPSTREAM_TRANSPORT_FAILURE_REASON%\" %
    BYTES_RECEIVED% %BYTES_SENT% %DURATION% %RESP(X-
    ENVOY-UPSTREAM-SERVICE-TIME)% \"%REQ(X-FORWARDED
    -FOR)%\" \"%REQ(USER-AGENT)%\" \"%REQ(X-REQUEST-
    ID)%\"
\"%REQ(:AUTHORITY)%\" \"%UPSTREAM_HOST%\" %
    UPSTREAM_CLUSTER% %UPSTREAM_LOCAL_ADDRESS% %
    DOWNSTREAM_LOCAL_ADDRESS% %
    DOWNSTREAM_REMOTE_ADDRESS% %
    REQUESTED_SERVER_NAME% %ROUTE_NAME%\n
```

[1]https://linkerd.io/

[2]https://www.consul.io/docs/connect

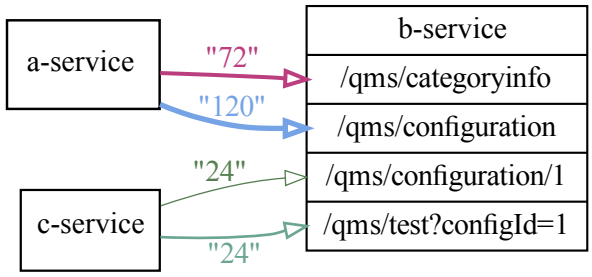

Fig. 2: A SDP prototype that can be generated from an access log. Each edge weight number represents the number of times the endpoint is called. When the weight value is greater, the edge thickens.

The default format of the access logs can be seen in the Listing 1. The Envoy proxy website lists the definitions of each term of this format, as well as other customizable formats [21]. Table I contains a sample inbound and outbound request's 'access-log', where both logs are from service-a's proxy sidecar, and its service DNS inside the Kubernetes cluster is 'a-service.default.svc.cluster.local'. One thing to note is that while both the inbound and outbound logs contain downstream and upstream addresses in the form of IPs, IPs are ephemeral in Kubernetes-based systems because they rely on ephemeral 'Pods' [22]. The service's DNS, on the other hand, is stable, and it will forward traffic to a Pod's or a group of pods' updated IP address [22].

The outbound request indicates that the event log contains upstream service information in DNS format, with the key 'upstream cluster' and 'path' indicating which upstream service endpoint is being requested. We can use this information to construct the SDG, in which a-service and b-service each represent a node, with a directed edge connecting a-service to b-service's endpoint `/api/v1/endpoint/`. We can also maintain the interaction count as edge-weight on this directed edge.'

We can analyze each access log of a distributed system and generate an edge in the graph for each interaction between two services within a time period. If the services already have an edge, we can increase the weight of the edge by one. Finally, we can use a visualizing tool to draw the graph automatically. We can make the edge visually thinner or thicker depending on the edge weight. Using different colors for different edges is also effective. Figure 2 shows a prototype of such an SDG, in which three services are represented by three nodes

Request Type	Sample Access Log

inbound request	{
	"start_time": "2022-05-26T06:22:02.661Z",
	"upstream_host": "10.244.0.65:12345",
	"downstream_local_address": "10.96.162.171:12345",
	"upstream_transport_failure_reason": null,
	"protocol": "HTTP/1.1",
	"upstream_service_time": "6",
	"authority": "b-service:12345",
	"requested_server_name": null,
	"response_code_details": "via_upstream",
	"connection_termination_details": null,
	"upstream_local_address": "10.244.0.41:33326",
	"downstream_remote_address": "10.244.0.41:48250",
	"path": "/api/v1/endpoint/",
	"bytes_sent": 44,
	"request_id": "4631dc4c-0a6e-9ad2-ba61-d257cdd6e50b",
	"bytes_received": 0,
	"route_name": "default",
	"duration": 7,
	"x_forwarded_for": null,
	"response_flags": "-",
	"response_code": 200,
	"method": "GET",
	"upstream_cluster": "outbound|12345||b-service.default.svc.cluster.local",
	"user_agent": "Apache-HttpClient/4.5.9 (Java/1.8.0_111)"
	}

outbound request	{
	"start_time": "2022-05-26T06:22:02.661Z",
	"upstream_host": "10.244.0.65:12345",
	"downstream_local_address": "10.96.162.171:12345",
	"upstream_transport_failure_reason": null,
	"protocol": "HTTP/1.1",
	"upstream_service_time": "6",
	"authority": "b-service:12345",
	"requested_server_name": null,
	"response_code_details": "via_upstream",
	"connection_termination_details": null,
	"upstream_local_address": "10.244.0.41:33326",
	"downstream_remote_address": "10.244.0.41:48250",
	"path": "/api/v1/endpoint/",
	"bytes_sent": 44,
	"request_id": "4631dc4c-0a6e-9ad2-ba61-d257cdd6e50b",
	"bytes_received": 0,
	"route_name": "default",
	"duration": 7,
	"x_forwarded_for": null,
	"response_flags": "-",
	"response_code": 200,
	"method": "GET",
	"upstream_cluster": "outbound|12345||b-service.default.svc.cluster.local",
	"user_agent": "Apache-HttpClient/4.5.9 (Java/1.8.0_111)"
	}

TABLE I: An example of an inbound and outbound request access log. Inbound requests have a-service as their destination, while outbound requests have a-service as their source and b-service as their destination.

called 'a-service,' 'b-service,' and 'c-service,' and each edge represents a request from a source service to a destination service endpoint.

V. Possible Analysis Directions

Using runtime logs to generate SDG and obtain service connectivity information eliminates the need for a static method's language-specific analyzer. We can use this information to detect anti-patterns and the evolution of the system architecture after collecting communication information between services and generating an SDG. The criticality and reliability metrics of the microservice architecture (MSA) [23] can also be measured automatically.

A. Architecture smell/Anti-Pattern detection

Architectural smells are one of the causes of architectural decay, and technical debt [24], [25]. Finding architectural smells and anti-patters can thus aid the software's long-term viability. We compiled a list of anti-patterns from Borges et al. [26] and Rud et al. [23], which can be detected using SDG and information about microservice communication:

- Absolute Importance of the Service (AIS) [23]: The number of microservices that invoke the current service

'a' is the AIS of service 'a'. Our analysis can count the number of microservices with an in-degree directed towards service 'a' to find AIS (a).

- Absolute Dependence of the Service (ADS) [23]: The number of microservices on which service 'a' relies is represented by the ADS of service 'a'. We can count the unique microservices with a directed edge from service 'a' to find ADS(a).
- Cyclic dependency / Services Interdependence in the System (SIY): A service-to-service cycle exists. It could be one, two, or even more services. Even though the requests do not loop because the same endpoints are not being called, this is still a bad practice that can lead to architectural decay. The SIY metric was defined by Rud et al. [23] as the number of pairs that are dependent on each other. We can look at each pair (a,b) to see if there is a path from service 'a' to service 'b' and vice versa. The SIY metric is the count of such pairs.
- Unbalanced API / Bottleneck Service / Absolute Criticality of the Service (ACS): This anti-pattern appears when a service has a large number of consumers, and the service becomes a single point of failure [27], [28]. Rud et al. [23] defined ACS of service 'a' as the product of AIS(a) and ADS(a), which is effectively the product of the number of inbound and outbound microservices of service a. As stated in the definition, we can calculate ACS(a) by multiplying AIS(a) and ADS(a) together (a).
- Shared Persistency [29]: This anti-pattern indicates that databases are shared among multiple services. The SDG allows us to determine whether various services access a single database.
- API Versioning [29]: APIs should have versions to deal with API changes between versions. Because we have endpoints in our SDG, we can check if they have versioning.

B. Software Architecture evolution/Design Rule Change from SDG

The service dependency graph (SDG) can show the evolution of a system's high-level architecture or design rules. Although software architecture artifacts can provide a service dependency graph, they can quickly become outdated, and MSA commonly decays over time due to architectural debt and code debt. As a result, an SDG generated from runtime logs provides a realistic view of inter-service communication. Furthermore, displaying SDG design rule changes can aid in the verification of new microservice co-relations and the detection of anomalies. It also shows how the new design rule affects real-world traffic.

C. Efficiently scale module using SDG heatmap

SDG combined with a heat map (number of encounters shown as edge weight) can help scale the system efficiently in resource consumption and cost. For example, the most frequently accessed services in the dependency graph can be identified using the SDG heatmap. Once these services have been identified, developers can make them stateless and deploy multiple copies to increase system throughput. On the other hand, if a service receives little traffic, it is more cost-effective not to scale it. This way, developers can deploy the system more efficiently while reducing costs and resource consumption and increasing system throughput.

VI. Case Study

We developed the prototype istio-log-parser tool [3], which takes an access log as input and generates a Service Dependency Graph with heatmap. It also calculates anti-pattern metrics and examines the system for any cycles. It generates a CSV file with anti-pattern metrics calculated, and the cycles are displayed in the program's stdout. To create the graph, we used Graphviz [4], which generates a dot file, which we then converted to pdf format using the dot library [30]. We used a microservice benchmark called TrainTicket [31] to test the prototype. The microservice's most recent release has around 270,000 lines of code. To test the prototype, we utilized both the v0.2.1 and v0.1.0 releases. TrainTicket v0.1.0 has 41 modules, while v0.2.1 has 42, and both use a reverse proxy based on Nginx to route requests to modules. The system was deployed on Kubernetes, along with Istio. The Istio was set up in such a way that it generates and outputs a JSON-formatted access log. We installed Kubernetes in a machine with 32GB of RAM and an Intel i9-8950HK processor using kind (Kubernetes-in-docker)[5]. There were 6 CPU cores and 12 threads in the system. Kubernetes' resource limit was set at 22GB of RAM and 8 threads of CPU.

We used the benchmarking tool PPTAM [32] to simulate a real-time user. It had five scenarios to simulate, and five users were making requests in these five scenarios at the same time. The test lasted 30 minutes and resulted in a total of 10,000 requests being sent to the server. After the simulation was finished, we collected each service's Istio proxy logs and ran our prototype tool on them. Both v0.2.1 and v0.1.0 went through a similar simulation process.

A. Anti-Patterns

Figure 3 is the generated SDG with heatmap of trainticket-v0.1.0 and Figure 4 is the SDG of v0.2.1. Table II shows a comparison of the automatically calculated anti-pattern metrics. From this available information we can find the followings about anti-pattern:

- Absolute Importance of the Service (AIS): We can see from Table-II that 'ts-order-service.default' has the most AIS (9). In addition, few services with 0 AIS indicate that they have no consumers or clients. Although this is not true for 'ts-ui-dashboard.default', because it is a reverse proxy, there are no inbound services, and the only consumers are requests from outside the cluster/microservice, which is referred to as ingress.

[3]https://github.com/the-redback/istio-log-parsing
[4]https://graphviz.org/
[5]https://kind.sigs.k8s.io/

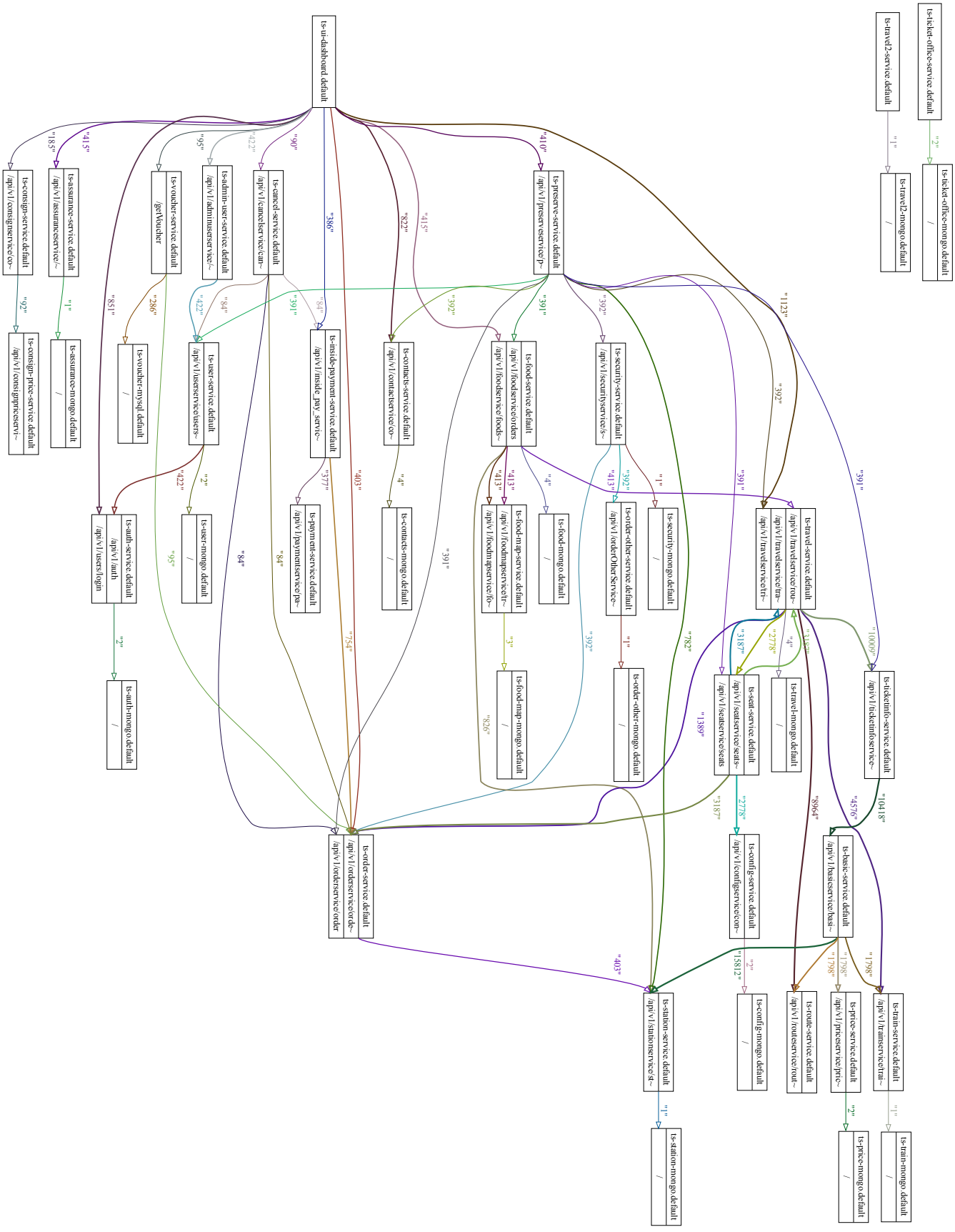

Fig. 3: The trainticket-0.1.0 release's Service Dependency Graph (SDG). The graph shows that 'ts-travel-service.default' has become one of the system's bottlenecks, while 'ts-travel-service.default' and 'ts-seat-service.default' have formed a cycle.

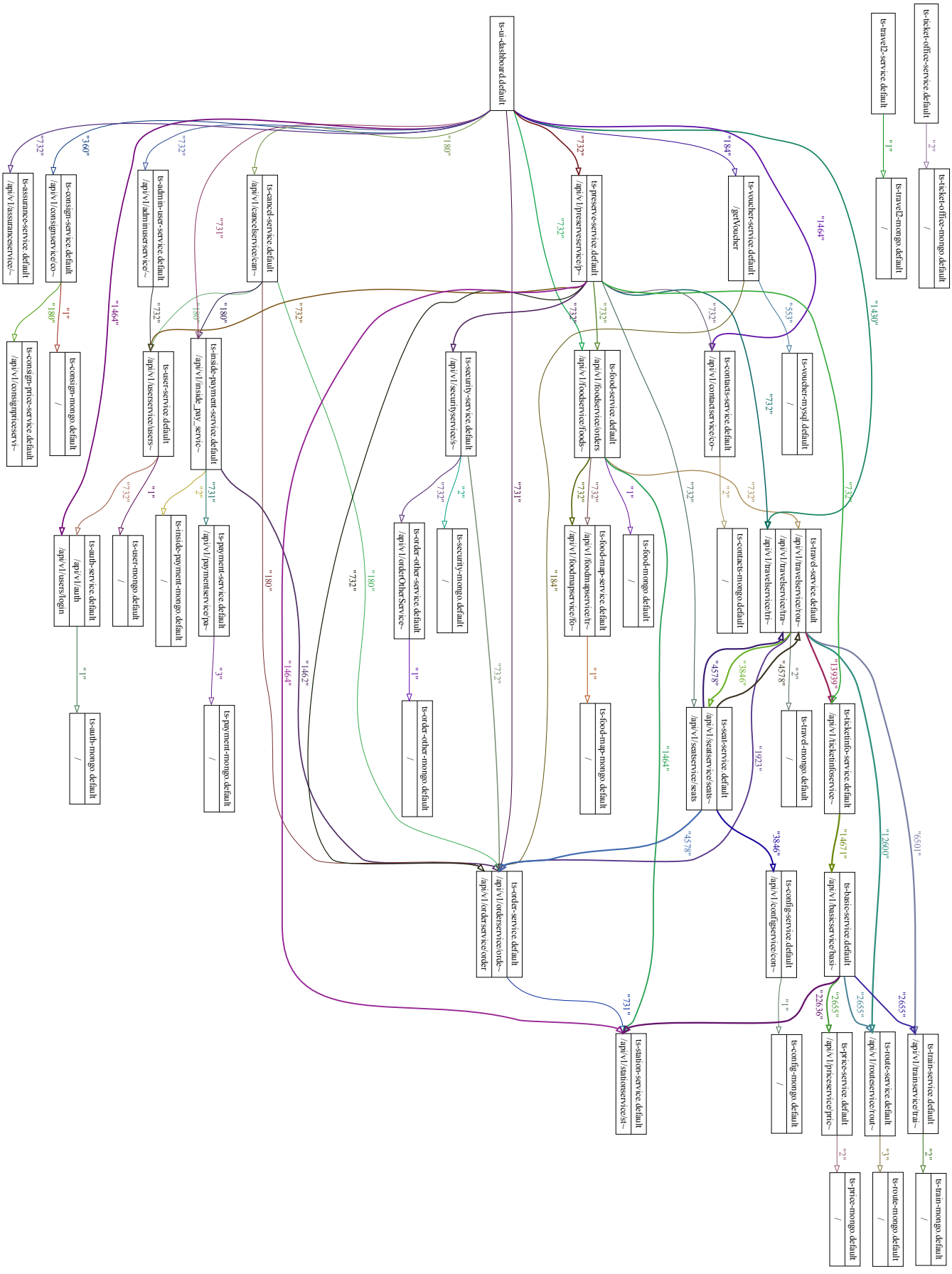

Fig. 4: Service Dependency Graph (SDG) of trainticket-0.2.1 release. Similar to v0.1.0, 'ts-travel-service.default' has become one of the system's bottlenecks, and a cycle has formed between 'ts-travel-service.default' and 'ts-seat-service.default'.

Service_Name	Train Ticket v0.2.1					Train Ticket v0.1.0				
	Service Dependencies		Criticality and Reliability Metrics			Service Dependencies		Criticality and Reliability Metrics		
	In-Degrees	Out-Degrees	Absolute Importance of the Service (AIS)	Absolute Dependence of the Service (ADS)	Absolute Criticality of the Service (ACS)	In-Degrees	Out-Degrees	Absolute Importance of the Service (AIS)	Absolute Dependence of the Service (ADS)	Absolute Criticality of the Service (ACS)
ts-admin-user-service.default	1	1	1	1	1	1	1	1	1	1
ts-assurance-mongo.default						1	0	1	0	0
ts-assurance-service.default	1	0	1	0	0	1	1	1	1	1
ts-auth-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-auth-service.default	2	1	2	1	2	2	1	2	1	2
ts-basic-service.default	1	4	1	4	4	1	4	1	4	4
ts-cancel-service.default	1	4	1	4	4	1	4	1	4	4
ts-config-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-config-service.default	1	1	1	1	1	1	1	1	1	1
ts-consign-mongo.default	1	0	1	0	0					
ts-consign-price-service.default	1	0	1	0	0	1	0	1	0	0
ts-consign-service.default	1	2	1	2	2	1	1	1	1	1
ts-contacts-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-contacts-service.default	2	1	2	1	2	2	1	2	1	2
ts-food-map-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-food-map-service.default	2	1	2	1	2	2	1	2	1	2
ts-food-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-food-service.default	2	5	2	5	10	2	5	2	5	10
ts-inside-payment-mongo.default	1	0	1	0	0					
ts-inside-payment-service.default	2	3	2	3	6	2	2	2	2	4
ts-order-other-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-order-other-service.default	1	1	1	1	1	1	1	1	1	1
ts-order-service.default	9	1	9	1	9	9	1	9	1	9
ts-payment-mongo.default	1	0	1	0	0					
ts-payment-service.default	1	1	1	1	1	1	0	1	0	0
ts-preserve-service.default	1	9	1	9	9	1	9	1	9	9
ts-price-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-price-service.default	1	1	1	1	1	1	1	1	1	1
ts-route-mongo.default	1	0	1	0	0					
ts-route-service.default	2	1	2	1	2	2	0	2	0	0
ts-seat-service.default	2	4	2	4	8	2	4	2	4	8
ts-security-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-security-service.default	1	3	1	3	3	1	3	1	3	3
ts-station-mongo.default						1	0	1	0	0
ts-station-service.default	4	0	4	0	0	4	1	4	1	4
ts-ticket-office-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-ticket-office-service.default	0	1	0	1	0	0	1	0	1	0
ts-ticketinfo-service.default	2	1	2	1	2	2	1	2	1	2
ts-train-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-train-service.default	2	1	2	1	2	2	1	2	1	2
ts-travel-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-travel-service.default	5	6	5	6	30	5	6	5	6	30
ts-travel2-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-travel2-service.default	0	1	0	1	0	0	1	0	1	0
ts-ui-dashboard.default	0	12	0	12	0	0	12	0	12	0
ts-user-mongo.default	1	0	1	0	0	1	0	1	0	0
ts-user-service.default	3	2	3	2	6	3	2	3	2	6
ts-voucher-mysql.default	1	0	1	0	0	1	0	1	0	0
ts-voucher-service.default	1	2	1	2	2	1	2	1	2	2

TABLE II: Comparison of antipattern metrics that are calculated automatically.

- Absolute Dependence of the Service (ADS): The out-degree of 'ts-ui-dasshboard.default' is twelve. As a result, it has the highest ADS in the system. ts-preserve-service.default has the second-highest ADS at nine.

- Cyclic dependency / Services Interdependence in the System (SIY): Both versions have a cycle between the services 'ts-travel-service.default' and 'ts-seat-service.default'. We can verify that the prototype's findings are correct by looking at Figure-3 and Figure-4.
- Unbalanced API / Bottleneck Service / Absolute Criticality of the Service (ACS): We can see from Table-II that the ACS value for 'ts-travel-service.default' is the highest. It applies to both versions v0.1.0 and v0.2.1. When we look at the SDG, we can see how this service has become a system bottleneck and a single point of failure. There are five services that 'ts-travel-service.default' is a client of, and six services that 'ts-travel-service.default' accesses. The ACS value for 'ts-food-service.default' is the second highest.
- Shared Persistency: It is clear from the SDGs (Figure-3 and Figure-4) that no MongoDB is shared by multiple services.
- API Versioning: We can see from both SDGs (Figure-3 and Figure-4) that each endpoint has 1/api/v1' as a prefix to the endpoint. This indicates that API versioning is in effect.

B. Evolution of MSA

The SDGs in versions 0.2.1 and 0.1.0 have the same number of services, with the exception that version 0.1.0 lacks a few MongoDB connections. Despite the fact that version 0.2.1 includes an additional module called 'ts-delivery-service,' it is not visible in the graph. This is because the PPTAM testing artifacts are older and do not cover the scenario of newly added business logic/service. Although this is a limitation of dynamic analysis, it does provide a scenario to verify that, due to API versioning, older tests are still functional without introducing new business flows.

C. Efficiently scale module

The services with the most inbound edges should be scaled first, as they are the most important for others to communicate with. In this case, it's necessary to scale 'ts-order-service. default' first. 'ts-travel-service.default' is the second service that needs to be scaled, but it has become the system's bottleneck with the highest absolute critical ACS value. Therefore, developers of such microservices must first reduce tanglement before scaling the service. The following microservices to scale are those with the highest ADS values and the most outbound edges. Because these services rely on various other services, the response time for each request to this service may increase, resulting in a queue for its users. Nevertheless, it is still necessary to distribute it in order to handle more users.

VII. Conclusion

This paper presents broad advancement to the microservice community, combining the perspective of dynamic analysis with quality assessment applicable to system evolution. It addresses current challenges in microservices. Challenges that the static analysis did not yet overcome due to decentralization and possible language heterogeneity across microservice codebases. It demonstrates the feasibility of anomaly detection on established telemetry tools and combines quality assurance knowledge established in service-oriented systems. We demonstrated the feasibility of the approach on a complex system benchmark with 42 microservices, showed detection of anti-patterns and metrics and presented a use case when applied to the evolving system. The results can give practitioners an automated tool to detect anomalies in non-centrally evolving systems and the ability to prioritize identified tasks.

In future work, we anticipate integrating business process analysis and enabling inter-weaving with anti-patterns. We also aim to study other microservice-relevant patterns that could be detected through dynamic analysis, such us `microservice-api-patterns.org`. We aim to integrate our detection and metrics into established tooling. Furthermore, despite the potential, we plan to compare it with static analysis.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1854049, grant from Red Hat Research, and Ulla Tuominen (Shapit).

References

[1] A. Walker, D. Das, and T. Cerny, "Automated code-smell detection in microservices through static analysis: A case study," *Applied Sciences*, vol. 10, no. 21, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/21/7800

[2] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano, "On the accuracy of sonarqube technical debt remediation time," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019, pp. 317–324.

[3] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[4] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 92–97. [Online]. Available: https://doi.org/10.1145/3387906.3388625

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

[6] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," *Information Science and Applications: Proceedings of ICISA 2020*, vol. 739, p. 223, 2021.

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

[8] V. Bushong., D. Das., and T. Cerny., "Reconstructing the holistic architecture of microservice systems using static analysis," in *Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER,*, INSTICC. SciTePress, 2022, pp. 149–157.

[9] M. Schiewe, J. B. Curtis, V. Bushong, and T. Cerny, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, 2022.

[10] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2018, pp. 221–232.

[11] ——, *Microservices Anti-patterns: A Taxonomy*. Cham: Springer International Publishing, 2020, pp. 111–128.

[12] S. Panichella, M. R. Imranur, and D. Taibi, "Structural coupling for microservices," in *11th International Conference on Cloud Computing and Services Science*, 04 2021.

[13] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, no. 1, p. 9, 2021. [Online]. Available: https://doi.org/10.1007/s10723-021-09551-5

[14] 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 *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, 2018, pp. 81–86.

[15] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards observability data management at scale," *ACM SIGMOD Record*, vol. 49, no. 4, pp. 18–23, 2021.

[16] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," *IEEE software*, vol. 22, no. 4, pp. 76–85, 2005.

[17] "Opentelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability," 2020. [Online]. Available: https://opentelemetry.io/

[18] R. Picoreti, A. P. do Carmo, F. M. de Queiroz, A. S. Garcia, R. F. Vassallo, and D. Simeonidou, "Multilevel observability in cloud orchestration," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 776–784.

[19] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[20] "Istio: Simplify observability, traffic management, security, and policy with the leading service mesh." 2020. [Online]. Available: https://istio.io/

[21] E. Authors, "Envoy-proxy access logging format rules," Aug 2020. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/access_log/usage

[22] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," in *Proceedings of netdev 0.1*, 2015, pp. 14–17.

[23] D. Rud, A. Schmietendorf, and R. R. Dumke, "Product metrics for service-oriented infrastructures," *IWSM/MetriKon*, pp. 161–174, 2006.

[24] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 235–245.

[25] A. Fellah and A. Bandi, "On architectural decay prediction in real-time software systems," in *Proceedings of 28th International Conference*, vol. 64, 2019, pp. 98–108.

[26] R. Borges and T. Khan, "Algorithm for detecting antipatterns in microservices projetcs," in *SSSME-2019: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. in SSSME-2019, 2019, pp. 21–29.

[27] M. Nayrolles, N. Moha, and P. Valtchev, "Improving soa antipatterns detection in service based systems by mining execution traces," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 321–330.

[28] F. Palma and N. Mohay, "A study on the taxonomy of service antipatterns," in *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*. IEEE, 2015, pp. 5–8.

[29] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.

[30] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: practice and experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[31] X. ZHOU, X. PENG, T. XIE *et al.*, "Benchmarking microservice systemsforsoftwareengineeringresearch," in *40th ACM/IEEE International Conference on Software Engineering (ICSE)*, 2018.

[32] A. Avritzer, D. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, and H. Schulz, "Pptam: production and performance testing based application monitoring," in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 39–40.