The $\mu$TOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures

Jacopo Soldani | Giuseppe Muntoni | Davide Neri | Antonio Brogi

Department of Computer Science, University of Pisa, Pisa, Italy

Correspondence
Jacopo Soldani, Department of Computer Science, University of Pisa, Largo B. Pontecorvo, 3, 56127 Pisa, Italy.
Email: soldani@di.unipi.it

[Correction added on 26 April 2021, after first online publication: “$\mu$ Topology and Orchestration Specification for Cloud Applications (TOSCA)” has been replaced with “$\mu$TOSCA” in this article.]

Summary
Exploiting microservices to architect enterprise applications is becoming commonplace. This makes it crucial to provide some support for designing and analyzing microservice-based applications, for example, for understanding whether a microservice-based application adheres to the main design principles of microservices and for choosing how to refactor it when this is not the case. To provide such support, in this article we present the $\mu$TOSCA toolchain. More precisely, we first introduce the $\mu$TOSCA model to represent the architecture of microservice-based applications with the OASIS standard TOSCA. We then describe a technique to automatically mine the architecture of a microservice-based application and represent it with $\mu$TOSCA, given the Kubernetes deployment of the application. We also present a methodology to analyze the $\mu$TOSCA representation of a microservice-based architecture to systematically identify the architectural smells potentially affecting the corresponding application and to resolve them. Finally, we present two prototype tools, $\mu$MINER and $\mu$FRESHENER, implementing our mining solution and the support for identifying and resolving architectural smells in microservice-based applications, respectively. We then assess —by discussing some case studies— how effectively $\mu$MINER, $\mu$FRESHENER, and the $\mu$TOSCA toolchain can support researchers and practitioners working with microservices.

KEYWORDS
architectural smells, microservices, refactoring, software architecture mining

1 | INTRODUCTION

Microservices are increasingly gaining momentum in IT enterprise, as they constitute an enabling technology to shorten the lead time in application development and to effectively scale application deployments.\textsuperscript{1,2} Besides their use in research,\textsuperscript{3,4} major IT companies like Amazon, Facebook, Google, Netflix, and Spotify already adopted microservices to deliver their businesses.\textsuperscript{5}
Microservice-based applications are essentially service-based applications whose architecture satisfies some additional design principles.\textsuperscript{6,7} Examples of such additional design principles are shaping services around business concepts, decentralizing all aspects of microservices (from governance to data management), ensuring the independent deployability and horizontal scalability of microservices, and isolating failures.\textsuperscript{2} As the exploitation of microservices to structure the architecture of enterprise applications is becoming commonplace, checking whether an application adheres to the main design principles of microservices and—if not—understanding how to refactor it are two key issues.\textsuperscript{8,9}

In this perspective, we present a methodology that enables to systematically and automatically identify the architectural smells possibly violating key design principles of microservices and to select architectural refactorings allowing to resolve such smells. The foundations for our methodology are given by the industry-oriented, multivocal review presented in our previous work.\textsuperscript{10} In such review, we singled out a set of architectural smells possibly violating some of the key design principles of microservices, as well as the architectural refactorings allowing to resolve each smell. We hereafter consider four of the architectural smells discussed in our industry-driven review,\textsuperscript{10} each with the architectural refactorings that allow resolve it.

Our proposal starts by modeling the architecture of a microservice-based application with the Topology and Orchestration Specification for Cloud Applications (TOSCA),\textsuperscript{11} the OASIS standard for specifying multiservice applications. We indeed introduce $\mu$TOSCA, a type system to specify microservice-based architectures as typed topology graphs in TOSCA. Intuitively, the nodes in a $\mu$TOSCA topology graph model the services, integration components (e.g., API gateways, load balancers, or message queues), and databases forming a microservice-based application, while the arcs indicate the runtime interactions occurring among them. Based on such representation, we then formally define the conditions to identify the occurrence of the considered architectural smells in a microservice-based application, and we illustrate how to refactor its architecture to resolve the identified smells. We also present $\mu$Freshener, a prototype tool enabling to edit $\mu$TOSCA topology graphs and implementing our methodology to provide researchers and practitioners with the support needed for identifying and resolving architectural smells in microservice-based architectures.

At the same time, given that microservice-based applications can include hundreds of interacting components,\textsuperscript{12} manually representing their architecture in $\mu$TOSCA constitutes a complex, time-consuming, and error-prone process.\textsuperscript{5} With $\mu$Freshener alone, the architect of a microservice-based application would have to manually specify all the various components forming her application and all the interactions occurring among such components.

To further support researchers and practitioners working with microservices, we hence propose a technique for automatically mining the architecture of a “black-box” microservice-based application. Our technique indeed works without needing to access the sources of the components in an application, but it rather enables deriving the architecture of a microservice-based application only from the declarative specification of its deployment in Kubernetes. This is done by performing three subsequent steps, that is, (i) by statically mining information on the software components forming an application from the manifest files specifying the application deployment in Kubernetes, (ii) by dynamically mining information on component interactions by monitoring a running application deployment, and (iii) by refining the information mined statically and dynamically to identify components implementing well-known message-based integration patterns, for example, message queues or load balancers.\textsuperscript{13} We also present $\mu$Miner, a prototype tool implementing the aforementioned technique to automatically derive a $\mu$TOSCA topology graph modeling the architecture of a microservice-based application, starting from its deployment in Kubernetes.

$\mu$Miner and $\mu$Freshener form a toolchain, which we hereafter call the $\mu$TOSCA toolchain, given that their integration relies on $\mu$TOSCA (Figure 1). The $\mu$TOSCA toolchain can effectively help researchers and practitioners in designing and analyzing their microservice-based applications. The architect of a microservice-based application can indeed take the Kubernetes deployment of her microservice-based application and run $\mu$Miner to automatically derive the $\mu$TOSCA file specifying the architecture of such application. The $\mu$TOSCA file can then be imported into $\mu$Freshener to analyze the specified microservice-based architecture to automatically identify the architectural smells affecting the corresponding application. In case any smell is detected, $\mu$Freshener also supports choosing which architectural refactoring to enact to resolve the occurrence of such smell. This in turn enables obtaining $\mu$TOSCA specifications representing “smell-free” applications.
microservice-based architectures, which researchers and practitioners can exploit to actually refactor the source code of their microservice-based applications. In summary, the main contributions of this article are the following:

- We present $\mu$TOSCA, a type-system for modeling the architecture of microservice-based applications with the OASIS standard TOSCA.\(^{11}\)
- We present a technique to automatically mine the $\mu$TOSCA representation of the architecture of existing microservice-based applications starting from their deployment in Kubernetes. We also present $\mu$MINER, a prototype tool implementing our mining technique.
- We present a methodology to systematically identify architectural smells possibly violating key design principles of microservices and to select refactorings allowing to resolve such smells. We also present $\mu$FRESHENER, a prototype tool implementing our methodology.
- We discuss a practical assessment of our solutions and tools by means of case studies, all based on existing, third-party microservice-based applications.

The rest of this article is organized as follows. Section 2 provides the necessary background on TOSCA and Kubernetes. Section 3 presents our modeling of microservice-based architectures ($\mu$TOSCA). Section 4 illustrates our solution for mining the architecture of existing microservice-based applications and its prototype implementation. Section 5 presents our support for analyzing and refactoring microservice-based applications to resolve the occurrence of architectural smells. Section 6 provides an assessment of our solutions for mining, analyzing, and refactoring microservice-based architectures. [Correction added on 26 April 2021, after first online publication: The phrase “and of their integration as a toolchain” was removed in the preceding sentence.] Section 7 discusses the usage and currently known limitations of the $\mu$TOSCA toolchain. Finally, Sections 8 and 9 discuss related work and draw some concluding remarks, respectively.

This article extends our previous work. A first, short description of $\mu$MINER (Section 4) and of $\mu$FRESHENER (Section 5) was presented in Muntoni et al.\(^{14}\) and Brogi et al.,\(^{15}\) respectively. This article presents a thorough description of the mining technique implemented by $\mu$MINER and of the analysis technique implemented by $\mu$FRESHENER. Most importantly, this article presents for the first time the usability and scope of the integrated $\mu$TOSCA toolchain, and assesses — by discussing some case studies and controlled experiments — how effectively $\mu$MINER, $\mu$FRESHENER, and the $\mu$TOSCA toolchain can support researchers and practitioners working with microservices.

2 | BACKGROUND

2.1 | TOSCA

TOSCA\(^{11}\) is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based and machine-readable modeling language that allows to describe cloud applications. TOSCA specifications can then be processed to automate the deployment and management of the specified applications.

TOSCA allows to specify a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template (Figure 2). The topology template is a topology graph, namely, a typed directed graph whose nodes represent the application components and whose edges represent the relations among

**FIGURE 2** The TOSCA metamodel\(^{11}\)
such components. More precisely, application components are represented by means of typed node templates, whereas intercomponent relations are represented by means of typed relationship templates.

Each application component appears in the topology as a node template, and each node template is in turn typed (Figure 2). This is because the purpose of node templates is to define application-specific instances of components, whereas the purpose of the corresponding node types is to describe a reusable software component that can be instantiated in multiple different applications. More precisely, a node type specifies the structure of the observable properties of a component, the management operations it offers, the requirements to instantiate it, and the capabilities it offers to satisfy the requirements of other components.

Intercomponent relations instead appear in the topology as relationship templates, each connecting a requirement of a node to the node offering the capability to satisfy such requirement (Figure 2). Similarly to node templates and node types, a relationship template defines a specific instance of a relationship, whose generic definition is given by the instantiated relationship type. A relationship type provides a reusable definition for a relationship (e.g., “hosted on,” “connects to”), by also indicating the operations to manage such relation and its observable properties.

Node templates can also be logically grouped, for example, to define groups of components to be managed together or to uniformly apply the same management policy to all the nodes forming a group. Groups of nodes appear in an application topology as group templates. Each group template is an instance of a logical grouping defined for a precise purpose. As such purposes can be many, the actual purpose of each group is specified by means of its group type.

Finally, it is worth highlighting that the TOSCA type system supports inheritance: A node type can be derived from another, thus allowing the former to inherit the latter’s properties, management operations, requirements, and capabilities. For instance, the generic characteristics of a web server could be defined in the parent node type Web Server, which could then be specialized into the node type Apache Web Server to enable specifying the properties and requirements specific to Apache web servers. The same holds for relationship types and group types, with new types that can be defined by extending other existing types to inherit and specialize their structure.

2.2 | Kubernetes

Kubernetes is a portable, extensible, and open-source platform that allows to deploy and manage a container-based application in a distributed cluster. A Kubernetes cluster is composed by nodes, whether physical or virtual machines, which are partitioned between a control plane and a set of compute machines. The control plane consists of one or more master nodes controlling the Kubernetes cluster itself. The compute machines instead consist of a set of worker nodes, which are those actually running the services of application deployed with Kubernetes. Intuitively, multiservice applications can then be deployed on a Kubernetes cluster by providing a specification of its deployment to the control plane of the cluster, which will then orchestrate the actual application deployment on the worker nodes. To do so, the application deployment must be specified in so-called manifest files and by complying with the Kubernetes object model, whose main entities are recapped hereafter. Interested readers can find a detailed, self-contained, and up-to-date introduction to Kubernetes in its official documentation.

2.2.1 | Pods

While Docker containers are the virtualization technology used to deploy applications, Kubernetes uses additional layers of abstraction to provide scaling, resiliency, and lifecycle management features. In particular, pods constitute the deployment units in Kubernetes: A pod is a deployable instance of an application component in Kubernetes, which is shipped within a single container or in several, tightly coupled containers. A pod can indeed encapsulate multiple Docker containers when such containers need to share the same resources. A typical use case for this is to ship the Docker container running an application component together with Istio “sidecar” containers monitoring it or proxying its incoming/outgoing communications.

Pods are designed as relatively ephemeral, disposable entities. When a pod gets spawned, it is scheduled to run on a worker node in a cluster. The pod remains on such node until the containers in the pod terminate or the pod is deleted. A pod can also be evicted from a node for lack of resources, or if the node fails. When a pod is deleted or evicted, it is actually removed from the cluster. Launching a new instance of the software component running in the pod hence requires to spawn a brand new instance of the pod.
2.2.2 | Managing pod deployments

Kubernetes provides a higher-level abstraction, called controller, which allows to manage pod instances. Controllers create and manage pods from so-called pod templates. A pod template is a declarative specification for creating a pod, which must be included in so-called workload resources, for example, Deployments, Jobs, or DaemonSets. Each controller for a workload resource exploits the pod template contained in the workload object to spawn pods, and to bring them in the desired state specified in the pod template itself.

Kubernetes also supports replicating pods, namely, creating multiple pod instances from the same pod template, and managing groups of replicated pods. Pod can indeed be spawned and horizontally scaled by exploiting specific controllers, namely, ReplicationController or ReplicaSet. They both ensure that a given number of replicas of a pod continue to run on a cluster. Replication controllers and replica sets indeed spawn new replicas of a pod if the actual number of replicas currently running in a cluster is lower than that desired, and it deletes pod replicas if there are too many replicas of a pod. In this way, the pods managed by such controllers are ensured to be respawned if they fail, or if they are evicted due to node failing or lacking resources. Hence, one such controller should always be used, even if an application consists of only one pod.

Replication controller and replica sets are typically deployed within Deployment workload resources. Deployment resources ease the lifecycle management of replicated pods, and they can include a replication controller or replica set specifying the pod template to be deployed and its desired number of replicas.

For instance, the deployment resource in Figure 3 specifies a replicated deployment of three nginx pods. The field .spec.template specifies the pod template, whose pods are labelled app: nginx using the .metadata.labels subfield. The specified pod runs one container, named nginx, which is to be created from the official Docker image nginx:1.14.2 publicly available on the Docker Hub. The field .spec.selector defines how the deployment finds which pods to manage, which in this example is the pod specified in the pod template in the same deployment (which is labeled as indicated by matchLabels). Finally, the field .spec.replicas specifies that the three instances of the pod are to be created and managed by a replica set.

2.2.3 | Kubernetes services

Kubernetes deployments enable spawning and destroying pods to ensure that a given number of replicas of a pod is running in the cluster. At the same time, since each pod gets its own IP address, different pod replicas running in different moments in time may get different IP addresses. This leads to a problem: If a replicated “backend” pod provides functionality to other “frontend” pods in a cluster, how do the frontends find out and keep track of the IP addresses assigned to the replicas of the backend pod?

Kubernetes services come exactly to that purpose. A Service is an abstraction component, which defines a logical set of pods and a policy by which to access them. The set of pods targeted by a Kubernetes service is usually determined by a selector. A Kubernetes service is essentially a message routing component balancing the load among the pods in the logical set defined by the Kubernetes service itself. A Kubernetes service indeed gets the incoming requests for the pods it abstracts, and it forwards such requests to such pods based on some user-specified balancing policy.

For instance, Figure 4 specifies a Kubernetes service, called my-service, targeting any pod labeled with app: MyApp and listening on TCP port 9376. Kubernetes assigns an IP address to my-service, which can be used to send requests to my-service, knowing that all such requests will be forwarded and handled by a replica of the pod labeled with app: MyApp.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata: { name: nginx-deployment, labels: { app: nginx } }
spec:
template:
  metadata: { labels: { app: nginx } }
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports: [ { containerPort: 80 } ]
  selector: { matchLabels: { app: nginx } }
replicas: 3
```

Figure 3 Example of Kubernetes Deployment
2.2.4 | Exposing services

Kubernetes services can be exposed outside of a Kubernetes cluster, hence making them directly accessible by external clients. This can be done by specifying that the service is of type `NodePort` or `LoadBalancer` (instead of `ClusterIP`, as in Figure 4). A `NodePort` service exposes the service on each node’s IP at a static port. A `LoadBalancer` service instead exposes the service externally by exploiting a cloud-hosted load balancer, which is to be provided by the cloud provider.

Kubernetes `Ingress` resources provide another alternative to expose Kubernetes services outside of a cluster. A Kubernetes ingress resource enables exposing HTTP/HTTPS routes from outside of a Kubernetes cluster to the services and pods running within the cluster. An ingress resource may be configured to provide services with externally-reachable URLs, to load balance traffic, or to offer name-based virtual hosting. Similar to all other Kubernetes resources, an ingress resource is actually a pod running in the cluster, which is to be deployed and managed by a controller, called ingress controller. It is worth noting that the manifest files specifying the Kubernetes deployment of an application can include ingress resources providing the message routes for implementing an API gateway, without associating such ingress resources to any ingress controller actually implementing them. By default, the ingress controllers that are already available in a Kubernetes cluster are used to implement the message routing defined by ingress resources specified in the application deployment and not associated with any controller.

3 | MODELING MICROSERVICE-BASED ARCHITECTURES

We hereby introduce the $\mu$TOSCA-type system, which provides the building blocks enabling to represent microservice-based architectures as typed topology graphs in TOSCA. Nodes can be services, communication patterns, or databases. A Service is a component running some business logic, for example, a service managing users’ orders in an e-commerce application. A CommunicationPattern is a component implementing a messaging pattern decoupling the communication among two or more components. Figure 5 contains two of the communication patterns defined by Hohpe and Woolf, namely, `MessageRouter` and `MessageBroker`. MessageBrokers are in turn distinguished based on whether they implement message brokering asynchronously (`AsynchronousMessageBroker`) or synchronously (`SynchronousMessageBroker`). Finally, a Database is a component storing the data pertaining to a certain domain, for example, a database of orders in an e-commerce application.

Nodes can be interconnected via `InteractsWith` relationships to model that a source node invokes functionalities offered by a target node. Such relationships can be enriched by setting the boolean properties `circuit_breaker`, `timeout`, and `dynamic_discovery`. The first two properties allow to indicate whether the source node is interacting with the target node via a circuit breaker or by setting proper timeouts. The property `dynamic_discovery` allows to...
specify whether the endpoint of the target of the interaction is dynamically discovered (e.g., by exploiting a discovery service). The properties circuit_breaker, timeout, and dynamic_discovery enable characterizing interactions in a way that is needed to check the occurrence of architectural smells possibly violating design principles of microservices, as we will show in Section 5.

Nodes can also be placed in an Edge group, to define which application components are publicly accessible from outside of the application. The rationale of Edge groups is to enable specifying which application components can be directly accessed by external clients, without requiring to actually model such external clients.

As an illustrative example, Figure 6 displays the μTOSCA topology graph modeling the architecture of a toy application.

The application is composed by three services (i.e., order, payment, and shipping), three communication patterns (i.e., gateway, router, and queue), and a database (i.e., orders). The application is intended to manage the orders of an e-selling systems, which can be uploaded by external clients by accessing the application gateway. The gateway allows to upload new product orders and it forwards them to the order service. The latter stores the newly uploaded order in the database orders and it starts interacting with the payment service to process the payment of the order. The actual instance of the payment service to be used for processing the payment of the order is dynamically discovered by a message router implementing server-side service discovery. In addition, the order service exploits a circuit breaker to avoid getting stuck waiting for answers from payment if the latter becomes unresponsive. Once the order payment is successfully processed, the order service enqueues the order in the asynchronous message broker implementing the queue of orders to be shipped. The latter is consumed by the shipping service, which pulls orders from the queue and proceeds with their shipping.

4 | Mining microservice-based architectures from their Kubernetes deployment

To support application developers in generating a μTOSCA topology graph modeling the architecture of an application, we hereafter propose a solution to automatically determine it from the application deployment in Kubernetes. More precisely, we first present a three-steps approach for mining the microservice-based architecture of an application from its Kubernetes deployment (Section 4.1). We then present μMiner, a prototype implementation of our approach that can automatically generate the μTOSCA topology graph modeling the architecture of an existing application (Section 4.2).

Examples of automatically mined μTOSCA topology graph can later be found in Section 6.1.

4.1 | Mining microservice-based architectures from their Kubernetes deployment

Our solution incrementally builds the μTOSCA topology graph modeling the architecture of a microservice-based application, as shown in Figure 7. We first mine information from the static description of the application deployment in Kubernetes (Section 4.1.1). We then dynamically mine component-to-component interactions to be included in the topology graph by sniffing the network packets exchanged among the components of a running instance of the application (Section 4.1.2). Finally, we refine the topology graph by analysing the sniffed network packets to automatically identify the integration patterns possibly exploited to structure the application (Section 4.1.7).
4.1.1 | Step 1: Static mining

Our solution starts by processing the Kubernetes manifest files specifying an application deployment to elicit the topology nodes modeling the application components. Since pods define the deployment units for the containers hosting application components, we add a separate node to the topology graph for each pod in the application deployment. In doing so, we follow the guidelines given by the Kubernetes documentation (Section 2.2): We assume that each pod in a Kubernetes deployment forms a single cohesive unit of service, that is, that a single container is deployed to host a service, integration component, or database. In addition, if a pod runs a container from the official Docker image of some database component, the topology node is assigned the type DataStore. Otherwise, the component is assumed to implement some business logic and its corresponding topology node is assigned the type Service.

The set of nodes in the $\mu$TOSCA topology graph is completed by including the message routers that are specified by the Kubernetes services and ingress resources in the manifest files specifying the deployment of an application in Kubernetes. A Kubernetes service is a message routing component that enables forwarding and balancing the traffic sent to the possibly multiple replicas of a pod (Section 2.2). We hence add a MessageRouter node to the topology graph for each Kubernetes service defined in an application deployment. We also model the fact that each Kubernetes service handles the requests incoming to a set of replicated pods. We indeed add InteractsWith relationships outgoing from each newly added MessageRouter node and targeting the topology nodes modeling the pods handled by the corresponding Kubernetes service. In addition, if a Kubernetes service is specified to be a NodePort or LoadBalancer, then it is publicly accessible from outside of the Kubernetes cluster where the application is deployed (Section 2.2). If this is the case, we place the MessageRouter node modeling a Kubernetes service in the Edge group of the $\mu$TOSCA topology graph (to reflect the fact that such component can be accessed from external clients).

Kubernetes ingress resources instead enable specifying message routing components acting as API gateways for applications deployed with Kubernetes, that is, components that can be accessed from outside of the Kubernetes cluster where an application is deployed and that handle the access of external clients to the services and pods running in the cluster. As Kubernetes ingress resources are actually implemented by associating them with ingress controllers (Section 2.2), we proceed as follows. For each ingress controller associated with an ingress resource in the manifest files specifying the Kubernetes deployment of an application, we add a MessageRouter node to the topology graph and we place it in the Edge group.

4.1.2 | Step 2: Dynamic mining

The $\mu$TOSCA topology graph obtained from the manifest files specifying the Kubernetes deployment of an application is enriched in the second step of our solution by dynamically mining information from a running instance of the application. In particular, we first configure the application deployment to enable sniffing the network packets containing the messages exchanged among the components forming the applications. We then enact a concrete application deployment on a Kubernetes cluster and monitor its execution to sniff the network packets exchanged among the application components. We exploit the monitored information to elicit the component-to-component interactions occurring in the applications, as well as to identify the usage of ingress controllers already existing in the Kubernetes cluster to implement some ingress resource, if any.
4.1.3 | Automatically configuring the Kubernetes deployment of an application

We enable the monitoring of the interactions occurring among the Kubernetes pods running the application components by automatically including a monitoring container within each of such pods. More precisely, the specification of each pod is automatically extended by including a container running a packet sniffer (i.e., WireShark\textsuperscript{19}), which is configured to sniff all the network packets sent and received by the container running in the pod. We also enable to uniquely identify the source and destination of each monitored network packet by automatically assigning each pod an unique hostname, which is either the pod hostname already specified in the Kubernetes manifest files or automatically generated.

It is worth noting that the manifest files specifying the Kubernetes deployment of an application can include ingress resources providing the message routes for implementing an API gateway, without associating such ingress resources to any ingress controller actually implementing them. By default, the ingress controllers that are already available in a Kubernetes cluster are used to implement the message routing defined by ingress resources specified in the application deployment and not associated with any controller (Section 2.2). To take also this case into account, we equip each ingress controller already available in the cluster with a sidecar packet sniffer, following the same approach explained above. This indeed enables use to monitor also the network packets sent and received by such ingress controllers, hence enabling use to check whether such controllers are exploited by Kubernetes to implement some ingress resources specified in the application deployment.

4.1.4 | Enacting and monitoring the application deployment

After automatically completing the configuration of the Kubernetes deployment of an application as described above, we actually enact the deployment of the application to start monitoring the network packets exchanged by the application components in component-to-component interactions. We keep the application running for a given amount of time, which can be customized to enable monitoring all the interactions occurring among the application components. In addition, to stress component-to-component interactions, we also run a load test. The latter can be directly included in the Kubernetes deployment of the application (e.g., as an additional pod that runs some logic for invoking the functionalities offered by the application components) or it can be a separate script invoking the application components that are accessible from outside of the Kubernetes cluster where the application is running. We then undeploy the application and we ensure that all containers and artifacts pertaining to the enacted deployment, load test, and monitoring of the application are removed from the Kubernetes cluster where it has been run (e.g., to avoid WireShark containers to continue sniffing the network packets sent and received by the ingress controllers available in the Kubernetes cluster).

We instead store all the network packets sniffed by the WireShark containers injected in the application deployment in separate log files. All such network packets indeed enable to determine the component-to-component interactions occurred while the application was running, as we show hereafter.

4.1.5 | Determining interactions among components

A component-to-component interaction occurs whenever a component invokes some functionality offered by another component at runtime. If this is the case, we include a InteractsWith relationship outgoing from the node modeling the invoker and targeting the node modeling the invoked component. To grasp this information from the sniffed network packets, we identify the invoker and invoked component in a component-to-component interaction as follows. We analyze the TCP segments of the network packets exchanged between application components. If a TCP segment sets SYN equal to 1 and ACK equal to 0, this means that a connection is being opened for allowing the component sending the packet to interact with the component receiving the packet. Starting from this observation, we include an InteractsWith relationship connecting a component to another if there exist a network packet sent by the former to the latter with SYN and ACK set to 1 and 0, respectively.

We also temporarily associate each of the InteractsWith added to the μTOSCA topology graph with all network packets exchanged in the corresponding interaction. We will use such network packets in the refinement step of our solution (Section 4.1.7), as they will enable identifying whether a topology node is implementing some message-based integration pattern.
4.1.6  Identifying the exploitation of ingress controllers

Kubernetes exploits the ingress controllers that are already available in a cluster to implement the ingress resources included in the deployment of an application, if no concrete ingress controller is associated with them in the manifest files specifying the application deployment itself (Section 2.2). We hence analyze the network packets sent by already existing ingress controllers to identify whether they were exploited to implement the message routing defined by some ingress resource in the Kubernetes deployment of the application. For each such ingress controller, we add a new MessageRouter node to the μTOSCA topology graph and we place such node in the Edge group. We also further analyze the network packets sent by such ingress controller to elicit the interactions starting from the ingress controller and targeting the components of the deployed application. We then model each identified controller-to-component interaction by adding a corresponding InteractsWith relationship, with the same approach as described above.

4.1.7  Step 3: Refinement

The third and last step of our solution starts from the μTOSCA topology graph obtained after the static and dynamic mining steps and refines it by analyzing the network packets associated with the InteractsWith relationships in the graph. We refine the obtained μTOSCA topology graph by identifying the nodes in the topology graph that implement well-known integration patterns (i.e., message routing or asynchronous message brokering) and by assigning such nodes with the corresponding type in μTOSCA (i.e., MessageRouter or AsynchronousMessageBroker). We also analyze the network packets associated with InteractsWith relationships to determine whether the corresponding interactions exploit client-side service discovery, that is, whether the source of the interaction has dynamically discovered the endpoint of the target. If this is the case, the property dynamic_discovery of the corresponding InteractsWith relationship is set to true.

4.1.8  Eliciting components implementing message routers

As microservices mostly rely on HTTP to intercommunicate, the components that implement message routing can set the HTTP header X-Forwarded-For. The header X-Forwarded-For is the standard approach for identifying the IP address of the client that sent a message, when such message passed through one or more HTTP proxies or load balancers. If all messages sent by a component have the HTTP header X-Forwarded-For set (and assuming the component to set the header in compliance with HTTP standard), then such component is implementing some form of message routing. We check this by inspecting the network packets associated with the InteractsWith relationships outgoing from each Service node in the μTOSCA topology graph: If they contain the HTTP header X-Forwarded-For, we change the type of such node into MessageRouter.

4.1.9  Eliciting components implementing asynchronous message brokers

Standard messaging protocols have been devised for implementing asynchronous message brokers, with AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport), and STOMP (Simple Text Oriented Message Protocol) being the most prominent examples. We can hence consider a component to implement the asynchronous message broker integration pattern if all messages sent and received by such component comply with one of such messaging protocols. We enact the corresponding check by inspecting the messages contained in the network packets associated with the InteractsWith relationships ingoing and outgoing from the Service node in the μTOSCA topology graph: The messages in such network packets must comply with either AMQP, MQTT, or STOMP.

The above check is however not enough: A service implementing some business logic and only communicating with an asynchronous message broker would be erroneously identified as being itself an asynchronous message broker. The messaging protocols AMQP, MQTT, and STOMP are however client–server, and they all distinguish whether a message is sent from a client to the server (or vice versa) in the header of the message itself. We hence also check whether the network packets received by a component include messages sent to the asynchronous message broker by its clients, and
that the network packets sent by a component include messages sent by the asynchronous message broker to its clients. If this is the case, we change the type of the node modeling such component into AsynchronousMessageBroker.

### 4.1.10 Eliciting interactions adopting service discovery

As explained by Richardson, service discovery occurs whenever a component dynamically resolves the IP address of another component, with which the former wishes to interact. To recognize whether this happened while the Kubernetes deployment of an application was running, we inspect the network packets associated with each InteractsWith relationship in the μTOSCA topology graph. If the IP address of the target of the interaction varies among such network packets, this means that dynamic service discovery occurred, with the source component connecting to different instances of the target component. We hence set the property dynamic_discovery of the corresponding InteractsWith relationship to true.

### 4.2 Prototype implementation

We have developed a prototype of the mining technique described in Section 4.1, called μMINER. The prototype is implemented in Python and it exploits the tshark command-line version of WireShark to enact packet sniffing. μMINER is open sourced under MIT license and it is publicly available on GitHub at https://github.com/di-unipi-socc/microMiner.

With μMINER, users can mine the architecture of microservice-based applications by issuing the following command:

```shell
$ sudo python3.8 -m microMiner generate kubernetes source target [time] [test] [name]
```

where `source` specifies the path to the folder containing the Kubernetes manifest files specifying the application deployment, while `target` indicates the path where to store the generated μTOSCA specification. The optional parameters `time`, `test`, and `name` instead allow to indicate how long the application deployment is to be run, the Python module containing the load test to run, and the name to be assigned to the application in the μTOSCA specification, respectively. Currently, μMINER must run on the master node of the Kubernetes cluster where to deploy and monitor the application. This enables μMINER to interact with the Kubernetes engine running on the master node to automatically configure, enact, and manage the deployment of a given application on the cluster.

Figure 8 illustrates the modular architecture of μMINER. The `main` module offers a command-line interface enabling to feed the manifest files specifying the Kubernetes deployment of the application whose architecture is to be mined and (optionally) the load test to run. The `main` module then starts orchestrating the `miner`, `refiner`, and `exporter` modules to enact the actual mining of the application architecture, by first (i) invoking the `miner` to enact the static and dynamic mining steps in our solution. The `main` module then (ii) invokes the `refiner` to refine the mined topology graph as described in the refinement step in our solution. Finally, the `main` module (iii) invokes the `exporter` to marshal the mined architecture to μTOSCA. Steps (i) and (ii) incrementally build and refine the topology graph modeling the architecture of the considered application by relying on the `topology` module, which enables instantiating and updating such topology graph. Step (iii) then picks the mined topology graph from the `topology` module and marshals it to μTOSCA.

It is worth noting that we developed μMINER with the strategy design pattern, with the ultimate goal of easing its adaptability to other deployment automation technologies than Kubernetes. To support this, the `main` module interacts

---

**Figure 8** The architecture of μMINER

[Colour figure can be viewed at wileyonlinelibrary.com]
with the core module right before the steps (i-iii) listed above. The main module indeed communicates to core the mining, refinement, and export strategies selected by the user and core loads such strategies into the environment of μMINER, so that the miner, refiner, and exporter enact the actual mining, refinement, and export accordingly. Currently, the prototype only supports the Kubernetes-based mining and refinement strategies described in Section 4.1 and the strategy to export the topology graph to μTOSCA.

5 | RESOLVING ARCHITECTURAL SMELLS VIA REFACTORING

Given the μTOSCA topology graph modeling the architecture of a microservice-based application, we now introduce a methodology (Section 5.1) to systematically and automatically identify architectural smells possibly violating key design principles of microservices. Our methodology also supports selecting the architectural refactorings that allow to resolve the occurrence of identified smells. We also present a prototype implementation of our methodology, which provides the support needed for analysing and refactoring the architecture of microservice-based applications (Section 5.2).

It is worth highlighting the actual contribution of this section resides in the methodology and in its prototype implementation, which enable automatically identifying the architectural smells, as well as reasoning on how to resolve them via refactoring. Indeed, the architectural smells and refactorings considered by our methodology have been shared as known issues and best practices by practitioners daily working with microservices, and they have been first collected in a multivocal review analyzing white and grey literature on the topic. 10

5.1 | Identifying and resolving architectural smells in microservice-based architectures

Our industry-oriented multivocal review10 singled out the most recognized architectural smells violating key principles of microservices and the architectural refactorings enabling to resolve the occurrence of such smells. Figure 9 provides an excerpt of the resulting taxonomy, showing four architectural smells violating three key design principles of microservices (viz., horizontal scalability, isolation of failures, and decentralization), together with the architectural refactorings to resolve such smells.

Starting from the taxonomy in Figure 9, we hereafter present a methodology for identifying the occurrence of architectural smells in microservice-based architectures and for reasoning on how to resolve such smells. After defining μTOSCA topology graphs, we formalize the conditions enabling to automatically determine the occurrence of smells in architectures modeled as μTOSCA topology graphs, and we show how to refactor architectures to resolve the occurrence of each identified smell. In doing so, we exploit the graphical support given by Figure 10.

(A) horizontal scalability  (B) isolation of failures  (C) decentralisation

FIGURE 9  A taxonomy for (A) design principles of microservices, (B) architectural smells, and (C) architectural refactorings10
### Defining topology graphs, formally

The microservice-based architectures modeled by $\mu$TOSCA topology graphs can be formally represented by triples, whose elements are (i) the typed nodes representing application components, (ii) the relationships forming the graph representing the architecture of an application, and (iii) the group of nodes defining the edge of the architecture.

**Notation 1 (Powerset).** We write $P(X)$ to denote the powerset of a set $X$.

**Definition 1 (Architecture).** Let $L = \{\text{circuit\_breaker, dynamic\_discovery, timeout}\}$ be the set of properties that can hold on an interaction. The *architecture* of an application is represented by a triple $A = (N, R, E)$, where

1. $N$ is a finite set of typed nodes representing application components,
2. $R \subseteq N \times N \times P(L)$ is a finite set of triples, each representing an interaction between two application components and the properties holding on the interaction, and
3. $E \subseteq N$ is a set of nodes defining the edge of the architecture.
Remark 1. Intercomponent relationships are represented by a set $R$ of triples to enable specifying multiple relationships from component $x$ to component $y$ to indicate that $x$ interacts with $y$ in different ways. For instance, $\langle x, y, \{\text{circuit\_breaker}\} \rangle$ denotes that $x$ interacts with $y$ via a circuit breaker, while $\langle x, y, \{\text{dynamic\_discovery}, \text{timeout}\} \rangle$ indicates another interaction between $x$ and $y$ where $x$ dynamically discovers the actual endpoint of $y$ and that it set proper timeouts. $\langle x, y, \emptyset \rangle$ instead denotes that none among circuit breakers, dynamic discovery, and timeouts is used in an interaction between $x$ and $y$.

Definition 1 is so general that it allows to describe an architecture where (a) a node interacts with itself, which would be senseless. It also allows to specify that (b) a database is invoking functionalities offered by another component or can be directly accessed by external clients of the application, which are not the cases in practice.\(^{2,23}\) Finally, Definition 1 allows to indicate that (c) no component is placing messages in an asynchronous message broker, and that (d) a message router is not routing messages towards other components or that it is never invoked. Both (c) and (d) would also be undesirable, as in both cases we would deploy integration components that are not used to integrate any other components. To avoid cases (a–d), we hereafter consider an architecture to be well-formed when none of them is occurring, namely, where (a) no self-interactions occur, (b) database are suitably accessed, and where (c) asynchronous message brokers, and (d) message routers are actually used, either as entry points for external clients or invoked by other components of the application.

Notation 2 (Types). We write $x$.type to denote the TOSCA type of a node $x$. Given two TOSCA types $t$ and $t'$, we also write $t \geq t'$ iff the TOSCA type $t$ is derived from or equal to the TOSCA type $t'$.

Definition 2 (Well-formed architectures). An architecture $A = \langle N, R, E \rangle$ is well-formed iff

\[
\begin{align*}
(\text{No Self-Interactions}) & \quad \forall \langle x, y, \cdot \rangle \in R : x \neq y, \\
(\text{Database Access}) & \quad \forall x \in N : x\.type \geq \text{Database} \Rightarrow (\exists \langle x, \cdot, \cdot \rangle \in R) \land x \notin E, \\
(\text{Message Brokers Usage}) & \quad \forall x \in N : x\.type \geq \text{AsynchronousMessageBroker} \Rightarrow (\exists \langle \cdot, x, \cdot \rangle \in R) \land \{x\} \cap E = \emptyset, \text{and} \\
(\text{Message Routers Usage}) & \quad \forall x \in N : x\.type \geq \text{MessageRouter} \Rightarrow (\exists \langle \cdot, x, \cdot \rangle \in R) \land (x \in E \land \exists \langle \cdot, \cdot, \cdot \rangle \in R)).
\end{align*}
\]

We hereafter assume architectures to be well-formed. One can also readily check that $\mu$MINER generates $\mu$TOSCA topology graphs representing well-formed architectures, assuming that (No Self-Interactions) holds, namely, that no component in an application is invoking its own functionalities by sending messages over the network. The way $\mu$MINER generates the topology graph is indeed such that none of other conditions gets violated.

\[
\begin{align*}
(\text{Database Access}) & \quad \text{Intuitively, this constraint is satisfied because databases are passive components, never initiating some interaction with other components, hence not generating interactions that would be monitored and that would result in \text{InteractsWith} relationship outgoing from some Database or AsynchronousMessageBroker node. In addition, Kubernetes deployments are such that databases are never directly accessible from outside of the applications.} \\
(\text{Message Brokers Usage}) & \quad \text{Asynchronous message brokers are recognized and included in a topology graph only if they correspond to components receiving messages complying with standard messaging protocols, hence meaning that some other component is interacting with them.} \\
(\text{Message Routers Usage}) & \quad \text{Similarly, message routers are recognized and included in a topology graph only if they correspond to some Kubernetes service handling the messages sent and received by some application component or if they correspond to components setting the HTTP headers of messages to explicitly indicate that they are routing messages to other components in the application.}
\end{align*}
\]

5.1.2 Resolving architectural smells possibly violating the horizontal scalability of microservices

The possibility of adding/removing replicas of a microservice is a direct consequence of the independent deployability of microservices. To ensure its horizontal scalability, all possible replicas of a microservice should be reachable by all other microservices invoking the functionalities it offers.\(^{24}\) Two architectural smells are known to possibly result in violating the horizontal scalability of microservices, namely, the no API gateway and endpoint-based service interactions smells.\(^{10}\)
5.1.3 | No API gateway

A microservice-based application is affected by a no API gateway smell whenever the external clients of the application directly access some internal application components. If one of such components is scaled out by adding one or more replicated component instances, the horizontal scalability of microservices may get violated. The external clients of the application may indeed keep invoking the original component instance, without reaching any of the newly introduced replicas. In other words, a no API gateway smell occurs in a microservice-based application whenever an application component are accessed by external clients without passing through an API gateway, with the latter being a message routing component used to redirect the messages of external clients to internal application components. Given the μTOSCA topology graph modeling the architecture of a microservice-based application, the above condition corresponds to checking whether the edge of the architecture contains something that is not a message router.

**Definition 3 (No API Gateway).** Let $A = \langle N, R, E \rangle$ be an architecture. A node $x \in N$ indicates a no API gateway smell iff

$$ x \in E \land x\.type \notin \text{MessageRouter}. $$

Figure 10 visually represents the possible occurrences of no API gateway smells in a μTOSCA topology graph, both due to placing a component $x$ (either a service or an asynchronous message broker) at the edge of an architecture. The figure also illustrates the architectural refactorings that resolve the occurrence of no API gateway smells. Both refactorings consists of introducing a message router acting as API gateway for the application, or reusing one already available in the application. This results in avoiding $x$ to get directly accessed from outside of the application.

5.1.4 | Endpoint-based service interaction

A microservice-based application is affected by an endpoint-based service interaction smell when a service directly invoke another service, for example, because the actual location of the instance of the invoked service is hardcoded in the source code of the invoker or because no message router/broker is exploited. When this happens, if the invoked service is scaled out by adding new service instances, the newly created instances cannot be reached by the invoker. Formally, an endpoint-based service interaction smell occurs whenever there is a direct interaction from a service $x$ to a service $y$, with $x$ not using any support for dynamically discovering the actual endpoint of $y$.

**Definition 4 (Endpoint-based service interaction).** Let $A = \langle N, R, E \rangle$ be an architecture. A relation $\langle x, y, P \rangle \in R$ indicates an endpoint-based service interaction smell iff

$$ x\.type \geq \text{Service} \land y\.type \geq \text{Service} \land \text{dynamic_discovery} \notin P. $$

Figure 10 visually illustrates the occurrence of an endpoint-based service interaction in the μTOSCA topology graph modeling the architecture of a microservice-based application, together with the architectural refactorings that allow to resolve the occurrence of such smell. All architectural refactorings share the ultimate goal of decoupling the interaction between the invoking service $x$ and the invoked service $y$ by introducing an intermediate integration pattern. The most common solution is to exploit a message router implement a server-side service discovery mechanism to dynamically resolve the endpoint of the service targeted by the interaction. The other two possible solutions consist of decoupling the interaction between $x$ and $y$ by exploiting a message router or an asynchronous message broker, respectively. It is worth noting that, in all cases, the interaction outgoing from $x$ must necessarily be updated. The message router/broker used to decouple the interaction between $x$ and $y$ may instead be already available in the application and reused to implement the architectural refactoring.

5.1.5 | Resolving architectural smells possibly violating the isolation of failures in microservices

The failures in microservice-based applications should be isolated, meaning that each service should tolerate failures when invoking the services it depends on. However, this is not the case when wobbly service interaction smells occur in the architecture of a microservice-based application.
5.1.6 | Wobbly service interaction

Consider the interaction between two services, with one service invoking the functionalities offered by the other service (directly or through a message router). The interaction between such services is “wobbly” when a failure in the invoked service can result in a failure in the invoker, potentially starting a cascade of failures. This typically happens when the invoker is consuming functionalities offered by the invoked service without handling the possibility of the invoked component to fail and get unresponsive, such as circuit breakers or timeouts.

**Definition 5** (Wobbly Service Interaction). Let $A = \langle N, R, E \rangle$ be an architecture. A relation $\langle x, y, P \rangle \in R$ indicates a wobbly service interaction smell iff

$$x.\text{type} \geq \text{Service} \land (y.\text{type} \geq \text{Service} \lor y.\text{type} \geq \text{MessageRouter}) \land \text{circuit}\_\text{breaker} \notin P \land \text{timeout} \notin P.$$ 

The possible occurrences of wobbly service interactions in $\mu$TOSCA topology graphs are visually displayed in Figure 10. The figure shows how wobbly service interactions occur when a service $x$ is interacting with another service or with a message router (dispatching the message outgoing from $x$ to other services), with such interaction not including any support for tolerating failures, that is, no circuit breaker or timeout is used. Figure 10 also displays the architectural refactorings that resolve the occurrence of wobbly service interactions smells, with four out of five only predicated on the value of the properties circuit\_breaker and timeout of the relationship outgoing from $x$. The less intrusive refactorings consist of replacing the wobbly service interaction between $x$ and $y$ with one exploiting a circuit breaker to wrap the invocations outgoing from service $x$ or using a timeout. Such refactorings allow $x$ not to get stuck waiting for an answer from $y$.

When $x$ and $y$ are both services, another possible solution is to decouple the interactions between $x$ and $y$ through an asynchronous message broker, with the latter being a new one or one already available in the application. The usage of an asynchronous broker allows $x$ to send its requests to the broker, with $y$ processing such requests when it is available, hence avoiding $x$ to get stuck or fail when $y$ fails. In addition, such refactoring allows to resolve also the occurrence of an endpoint-based service interaction smell, if any.

5.1.7 | Resolving architectural smells possibly violating the decentralization of microservices

Decentralization is key in all aspects of microservices, therein included the decentralization of data management in microservice-based architectures. Each database in a microservice-based architecture should indeed be accessed by only one service, otherwise we incur in a shared persistence smell.

5.1.8 | Shared persistence

A microservice-based architecture is affected by a shared persistence smell whenever multiple services interact with the same database, independently of whether they access it directly or through some intermediate message routers. To formalize this, we first introduce a shorthand notation to denote that a component interacts with another directly or through a sequence of message routers.

**Notation 3** (Interaction sequences). Given an architecture $A = \langle N, R, E \rangle$, we write $x \rightarrow^{\ast} y$ to denote that a component $x$ interacts with another component $y$ directly or through a sequence of message routers, that is,

$$x \rightarrow^{\ast} y \Leftrightarrow \exists \{\langle x, w_1, \cdot \rangle, \langle w_2, w_3, \cdot \rangle, \ldots, \langle w_n, y, \cdot \rangle \} \subseteq R . \forall i \in [1, n] . w_i \geq \text{MessageRouter}.$$ 

**Definition 6** (Shared Persistence). Let $A = \langle N, R, E \rangle$ be an architecture. A set of nodes $\{x_1, x_2, \ldots, x_n, y\}$ indicates a shared persistence smell iff

$$y.\text{type} \geq \text{Database} \land (\forall i \in [1, n] . (x_i \geq \text{Service} \land x_i \rightarrow^{\ast} y)).$$ 

Figure 10 provides the visual representation of a possible shared persistence smell, together with three architectural refactorings that allow to reduce the amount of services accessing the same database. For readability reasons, the figure
focuses on the case in which multiple services \( x_1 \ldots x_n \) directly interact with the same database \( y \). The representation of the possible smells and refactorings for the case with services exploiting message routers to access the databases are similar. They only differ for the fact that the service-to-database interactions shown in the figure would be replaced by sequences of interactions, each involving one or more message routers.

It is worth noting how all the architectural refactorings in Figure 10 allow to reduce the amount of services accessing the database \( y \), hence ultimately allowing to resolve the occurrence of a shared persistence smell. Although their goal is the same, such refactorings are very diverse in spirit, and apply to different situations, highly depending on the services accessing the same database. For instance, if a service \( x_1 \) is the only service accessing a portion of the data stored in \( y \), then splitting the database \( y \) is an option. The database \( y \) can indeed be split in two different databases \( y_1 \) and \( y_2n \), with \( y_1 \) only storing the portion of data accessed by \( x_1 \) and with \( y_2n \) storing the rest of the data. The service \( x_1 \) then becomes the only accessing \( y_1 \), while \( y_2n \) is accessed by the other services \( x_2 \ldots x_n \).

Other possible solutions to reduce the amount of services accessing the same database \( y \) are exploiting a data manager or merging some of the services accessing the database. Exploiting a data manager consists of adding a service \( y_m \), or reusing one already available, to proxy the access of services \( x_1 \ldots x_h \) (with \( h \leq n \)) to the database \( y \). The other refactoring instead consists in merging the services \( x_1 \ldots x_h \) (with \( h \leq n \)) into a single service \( x_1h \). The rationale behind this last refactoring is that, when multiple services access the same database, this may be indicating that the application has been split “too much” and resulted in too fine-grained services processing the same data.

\[ \text{FIGURE 11} \quad \text{Snapshots of (A) the editing/analysis and (B) refactoring views of } \mu\text{FRESHENER. [Colour figure can be viewed at wileyonlinelibrary.com]} \]

### 5.2 Prototype implementation

We hereby present \( \mu\text{FRESHENER} \), a prototype implementation of our approach for identifying and resolving architectural smells in microservice-based applications. \( \mu\text{FRESHENER} \) is open sourced under MIT license and it is publicly available on GitHub at https://github.com/di-unipi-socc/microFreshener.

\( \mu\text{FRESHENER} \) provides a web-based graphical user interface for editing \( \mu\text{TOSCA} \) topology graphs, for automatically identifying architectural smells in specified topology graphs, and for exploring/applying architectural refactorings for resolving the identified smells.

Figure 11 provides two snapshots of the graphical-user interface of \( \mu\text{FRESHENER} \). The figure shows (A) the view enabling to edit the \( \mu\text{TOSCA} \) topology graph modeling the architecture of a microservice-based application, either from scratch or after importing existing TOSCA files. The same view also enables running analyses on the \( \mu\text{TOSCA} \) topology graph under editing, which automatically identify the architectural smells affecting the corresponding microservice-based architecture, if any. The automatically identified smells are then displayed with icons placed on the nodes interested by
the smells. Clicking on one of such icons results in opening (B) the view enabling to select the architectural refactoring to resolve the corresponding architectural smell. Once selected, the μTOSCA topology graph is updated in accordance with the architectural refactoring, that is, by updating the graph by applying the transformations shown in Figure 10. The refactoring is actually applied only to the μTOSCA topology graph modeling the architecture of a microservice-based application (and not on the sources of the application itself). One can also go back and forth along the sequence of architectural refactorings by undoing or redoing them by clicking on the corresponding buttons.

Figure 12 illustrates the architecture of μFRESHENER. The core modules, that is, model, importer, analyser, and exporter, implement the core business logic of μFRESHENER. The importer enables importing μTOSCA specifications defining the topology graphs representing the architecture of microservice-based applications. Such topology graphs are represented internally to μFRESHENER by exploiting the object model defined by model, which is then stored in memory. The analyser implements the logic for automatically identifying the architectural smells in the topology graph stored in memory, while the exporter enables exporting in memory topology graphs back to μTOSCA. The core modules define a standalone Python library that is publicly available on GitHub at https://github.com/di-unipi-socc/microFreshener-core. They are imported by the backend of μFRESHENER, which is implemented as a Python-based REST api coordinating the core modules to offer the possibility of uploading μTOSCA specifications and analyzing them to return the list of architectural smells affecting the architecture of the corresponding microservice-based applications.

The frontend modules instead implement the graphical user interface offered by μFRESHENER. gui is a web application implemented with angular, which provides the editing and refactoring views in Figure 11. It interacts with the REST api in the backend of μFRESHENER to retrieve the list of architectural smells affecting the modeled μTOSCA topology graphs, while it interacts with the refactorer module to offer the support for reasoning on which architectural refactorings enable to resolve the smells affecting the architecture of a microservice-based application.

6 | ASSESSMENT

We hereafter illustrate how we assessed the support provided by the μTOSCA toolchain. More precisely, we first present an evaluation of our architecture mining support on two third-party applications (Section 6.1). We then demonstrate the support given by our solution for identifying and resolving architectural smells in microservice-based applications based on an industry-driven case study (Section 6.2).

6.1 | Assessing our solution for mining microservice-based architectures

To validate our solution to automatically determine the architecture of existing microservice-based applications, we run different experiments based on two open-source, third-party applications, that is, Online Boutique27 and Sock Shop.28

1. Since an explicit representation of the architecture of Online Boutique is publicly available online, we double-checked whether the architecture generated by running μMINER on Online Boutique was succeeding in suitably recognizing all the components and interactions structuring the architecture of Online Boutique.

2. We did the same for Sock Shop, whose architecture is also explicitly given in its online documentation. We also compared the architecture automatically generated by μMINER with that obtained with Weave Scope,29 a production-ready state-of-the-art solution for automatically mining the architecture of service-based applications.
In all such experiments, the applications were deployed and monitored in a Kubernetes cluster consisting of three virtual machines, one being the master node where \( \mu \text{Miner} \) was running and the other two being worker nodes. The master node was equipped with 4GB of RAM and two cores, while worker nodes were equipped with 3GB and two cores each.

Both experiments (i) and (ii) allow to compare the architecture obtained with \( \mu \text{Miner} \) with the “ground truth” given by the explicit architecture representation available in their online documentation (which we first double-checked by running the applications and by inspecting their sources). Experiment (ii) also shows that \( \mu \text{Miner} \) generates a more informative architecture representation than that obtained with WeaveScope.

### 6.1.1 Determining the architecture of Online Boutique

*Online Boutique*\(^{27}\) is a demo microservice-based application developed by Google. The application is a web-based e-commerce application where users can browse the catalogue of items being sold, add them to the cart, and place orders. The application consists of eleven interacting components, which result in the architecture shown in Figure 13. The figure also shows that Frontend is the only component that is publicly accessible through the Internet, as well as that the application is equipped with an additional component (Load Generator), which enables to load test the application once deployed.

The objective of our experiment was to double-check whether \( \mu \text{Miner} \) was capable of determining all components and interactions shown in Figure 13. We hence gave the publicly available Kubernetes deployment of *Online Boutique* (https://github.com/GoogleCloudPlatform/microservices-demo/tree/master/kubernetes-manifests) as input to \( \mu \text{Miner} \). In addition, since the Load Generator can automatically generate traffic to load test the application, we did not to pass any test script to \( \mu \text{Miner} \). We indeed load tested our deployment of *Online Boutique* application with the Load Generator, which is already included in the Kubernetes manifest files.

The \( \mu \text{TOSCA} \) topology graph automatically mined by \( \mu \text{Miner} \) is shown in Figure 14. By comparing such topology graph with the explicit representation of the architecture of *Online Boutique* available in its online documentation (and reported in Figure 13), we can observe that \( \mu \text{Miner} \) identified all the services and databases forming the application, as well as all interactions occurring among them. \( \mu \text{Miner} \) also suitably recognized that frontend is the only service that is publicly accessible from outside of the application. In addition, the obtained topology graph is also richer in the sense that it represents the message routers implementing the Kubernetes services specified in the actual deployment of *Online Boutique* in Kubernetes. As already noticed in Section 4.1, Kubernetes services are message routing components that enable forwarding the traffic sent to the possibly multiple replicas of the application components they are associated with.\(^{16}\)

### 6.1.2 Determining the architecture of Sock Shop

*Sock Shop*\(^{28}\) is an open-source web-based application simulating the user-facing part of an e-commerce website selling socks. It is developed and maintained by Weaveworks and Container Solutions, with the goal of enabling to test and showcase solutions for microservice-based applications.

---

*FIGURE 13* The architecture of *Online Boutique*. The architecture representation is copied from that included in the application documentation publicly available on GitHub\(^{27}\).
Figure 14: µTOSCA topology graph representing the architecture of Online Boutique. The graph was automatically obtained with µMiner, which returned the TOSCA file publicly available on GitHub at https://github.com/di-unipi-socc/microMiner/blob/master/tests/kubernetes/online-boutique/microTOSCA.yml [Colour figure can be viewed at wileyonlinelibrary.com]

Figure 15 displays the architecture of Sock Shop as per its specification in the official application documentation. In particular, the figure visualizes all the application components in Sock Shop, the interactions occurring among them, and the fact that front-end is the only service that is publicly accessible from external clients of the application.

To further demonstrate that our solution can effectively mine the architecture of microservice-based applications, we repeated the same experiment as for the Online Boutique application. We hence given the publicly available Kubernetes deployment of Sock Shop (https://github.com/microservices-demo/microservices-demo/tree/master/deploy/kubernetes) as input to µMiner. We also configured µMiner to keep the application deployment up and running and we developed a script for load testing the Sock Shop application (https://github.com/di-unipi-socc/microMiner/tree/master/tests/kubernetes/sock-shop/test), which we also gave as input to µMiner.

The µTOSCA topology graph automatically mined by µMiner is shown in Figure 16. One can readily check that all the components and interactions that are in the declared architecture (Figure 15) have been recognized and included in the µTOSCA topology graph too. As for the declared components and interactions, a missing piece of information is the fact that catalogue-db and users-db are not classified as Database by µMiner, but rather considered Service nodes. This is because such databases are implemented by nonofficial Docker images built by the developers of Sock Shop, rather than by running official Docker images for databases. This provides a concrete example of a current limitation of our mining technique, which can recognize components as implementing databases only if implemented by running...
official Docker images for databases, and which we further discuss in Section 7. At the same time, it shows how easy is to recognize such a kind missing pieces of information, which an application architect can fix by editing the obtained topology graph, for example, with \(\mu\)FRESHENER.

The \(\mu\)TOSCA topology graph is also including additional components and interactions with respect to the declared ones. As for the case of Online Boutique, the graph includes the message routers implementing the Kubernetes services specified in the actual deployment of Sock Shop in Kubernetes. The graph is also including additional interactions outgoing from orders, as the latter is interacting at runtime not only with shipping and orders-db (as declared in Figure 15), but also with users and carts.

It is worth noting that the additional interactions were also identified when we run Sock Shop with Weave Scope. Figure 17 shows the visualization of the architecture of Sock Shop in Weave Scope, obtained by load testing the application
with the same load test we gave as input to \( \mu \text{MINER} \). If comparing the architecture representations obtained from \( \mu \text{MINER} \) (Figure 16) and Weave Scope (Figure 17) one can readily notice that, while they both identify all application components and the interactions occurring among them, the \( \mu \)TOSCA topology graph obtained from \( \mu \text{MINER} \) is more informative: The latter indeed explicitly represents the message routers automatically injected when deploying application components as Kubernetes services, and it also shows the actual type of components structuring the architecture of a microservice-based application (distinguishing whether a component is a Service, Asynchronous Message Broker, Message Router, or Database).

### 6.2 Assessing the methodology for identifying and resolving architectural smells

To assess our methodology for identifying and resolving architectural smells in microservice-based applications, we exploited \( \mu \text{FRESHENER} \) to run an industry-driven case study based on a real-world application developed and maintained by an Italian IT company we are cooperating with. The rationale of this case study is to show that each identified smell was resolved by actually implementing a corresponding architectural refactoring, and how this resulted in qualitative improvements of the performances, accessibility, and maintainability of the considered application.

The considered application is a platform involving 21 components, whose purpose is to enact predictive maintenance in a smart factory. As a result, we were able to identify five architectural smells affecting the considered application, which have been resolved by the company in accordance to what we presented in Section 5.1. The TOSCA files specifying the \( \mu \)TOSCA topology graphs modeling the original and refactored architectures of the considered application are publicly available on GitHub at https://github.com/di-unipi-socc/microFreshener-core/tree/master/data/examples/case-study.

Figure 18 illustrates the 12 services, 7 databases, and 2 asynchronous message brokers composing the considered application, anonymized for privacy reasons. The figure also illustrates the interactions between them, with service-to-service interactions being such that (d) the endpoint of the target of the interaction is dynamically resolved by exploiting a client-side service discovery mechanism and that (t) timeouts are exploited to enhance the fault resilience of the source of the interaction. Even if the considered topology is small, the number of components and interactions makes it not easy to manually identify all occurrences of architectural smells.

We hence modeled the application topology with \( \mu \text{FRESHENER} \), and this allowed to identify the five instances of architectural smells affecting the considered topology, that is, four instances of the no API gateway smell (regarding \( m_1, m_2, s_1, \) and \( s_2 \)), and one instance of the shared persistence smell (due to services \( s_6, s_7, s_8, \) and \( s_9 \) all accessing the same database \( d_6 \)). The identified architectural smells were to be refactored and solutions were found by still exploiting \( \mu \text{FRESHENER} \).

![Figure 18](https://example.com/filename.png)

**Figure 18** Anonymized architecture of the considered application. Labels d and t on relationships explicitly represent that properties dynamic_discovery and timeout are set to true (with the other being false) [Colour figure can be viewed at wileyonlinelibrary.com]
**FIGURE 19** Refactoring of the architecture in Figure 18. Labels d and t on relationships explicitly represent that properties `dynamic_discovery` and `timeout` are set to true (with the other being false). Updates due to refactorings are in grey [Colour figure can be viewed at wileyonlinelibrary.com]

μFRESHENER indeed enabled to explore the architectural refactorings allowing to resolve the occurrence of the identified smells. The actual refactorings to apply were then chosen based on business requirements and the actual cost for implementing them, that is, for refactoring the application sources by following the guidelines given by an architectural refactoring:

- A message router $g_1$ was first introduced for resolving the no API gateway smell indicated by $m_1$. Then, given that the external clients placing messages in $m_1$ and $m_2$ were the same (i.e., smart production machines sending monitored data to the platform), the gateway $g_1$ was exploited to resolve also the no API gateway smell indicated $m_2$. Similarly, since $s_1$ and $s_2$ were services accessed by the same clients, a message router $g_2$ was introduced for managing the access to $s_1$ and $s_2$ from outside of the platform.
- The shared persistence smell due to services $s_6$, $s_7$, $s_8$, and $s_9$ all accessing the same database $d_6$ was instead resolved by introducing a novel service $s_{13}$, acting as data manager for $d_6$. Services $s_6$, $s_7$, $s_8$, and $s_9$ were then directly interacting with $s_{13}$ to access the data in $d_6$, and this resulted in adding novel architectural smells (as each newly introduced service-to-service interaction was endpoint-based and wobbly). To resolve such smells, and similarly to the other service-to-service interactions in the considered topology, the newly introduced interactions were refactored in such a way that the endpoint of the target of the interaction was dynamically resolved and that proper timeouts were set.

The above listed refactoring where actually implemented by the IT company, which refactored the application and obtained the architecture illustrated in Figure 19. As a result, the company experienced noteworthy improvements in the performances and accessibility of the application, thanks to the introduction of the API gateways $g_1$ and $g_2$. The company also notified us about sensible improvements in the maintainability of the application, as introducing service $s_{13}$ as data manager for $d_6$ enabled to decouple the latter from services $s_5$, $s_6$, and $s_9$. The subsequent updates they had to release on $d_6$ only required to update the data manager service $s_{13}$, which anyhow continued to provide the interfaces needed by $s_5$, $s_6$, and $s_9$, hence not needing to update them as well.

7 | **DISCUSSION**

We hereafter discuss the usage of the proposed μTOSCA toolchain, therein included currently known limitations and potential threats to validity. Starting from our mining technique, it is worth noting that the accuracy of the topology graph representing the mined architecture of a microservice-based application depends on the load test run to stress
component-to-component interactions. Independent of whether it is included in the application deployment or provided in a separate script interacting with the publicly accessible application components, the load test must last long enough to enable all component interactions to occur multiple times. This is needed for the corresponding arcs to appear in the mined topology graph, as well as for the properties of such arcs to indicate whether dynamic discovery is exploited in the corresponding interactions.

At the same time, even with suitably configured load tests, our mining technique does not automatically determine whether circuit breakers or timeouts are used in component-to-component interactions, nor it recognizes as databases only containers running from official Docker images for databases. In addition, a potential threat to the validity of our mining technique is that it assumes application components to adhere to widely accepted standards to extract information. We indeed assume components to exploit the HTTP header \texttt{X-Forwarded-For} in compliance with the HTTP standard, as we use this information to identify whether a component is implementing some form of message routing. We also assume asynchronous message brokers to interact with other components by means of widely accepted standards, such as AMQP, MQTT, and STOMP. However, a developer may build its own broker by exploiting a brand new interaction protocol, or she may misuse the above listed standards, and this may potentially threat the validity of our mining technique.

It follows that the \( \mu \)TOSCA topology graph generated by our mining technique can miss some information. However, the application architect can complete the \( \mu \)TOSCA topology graph in case some information is missing. She can indeed modify the TOSCA YAML\textsuperscript{11} file containing the \( \mu \)TOSCA topology graph either by editing it directly or, more comfortably, exploiting graphical environments, like Winery\textsuperscript{30} or the editing pane featured by \( \mu \)FRESHENER (which we described in Section 5.2).

Concrete examples of the above were given by the mined architectures of the applications considered when evaluating our mining technique (Section 6.1). For instance, all the interactions in the mined \( \mu \)TOSCA representation of the architecture for \textit{Sock Shop} (Figure 16) are such that no circuit breakers or timeouts are used by a service when consuming functionalities offered by other services. This is actually because \( \mu \)MINER, in its current implementation, is not capable of automatically detecting whether some circuit breaker or timeout is used in component-to-component interactions. In addition, the nodes \texttt{users-db} and \texttt{catalogue-db} are recognized as services, as they are implemented by running Docker images built by the developers of \textit{Sock Shop}. The current prototype of \( \mu \)MINER, however, recognizes as Databases only those component running from official Docker images for databases. On the other hand, we knew that \texttt{users-db} and \texttt{catalogue-db} were actually implementing databases, and —by analyzing the actual implementation of the services in \textit{Sock Shop}— we realized that all invocations enacted by such services are done with default timeouts set. We hence acted as/on behalf of the application architect and we exploited the editing pane of \( \mu \)FRESHENER to update the \( \mu \)TOSCA topology graph generated by \( \mu \)MINER to reflect the actual implementation of \textit{Sock Shop}. We indeed changed the type of \texttt{users-db} and \texttt{catalogue-db} into \texttt{Database} and we specified that timeouts are used in all service interactions. We then exploited \( \mu \)FRESHENER to analyze the obtained architecture, which resulted in no smell being identified.

The same happened for the other applications we considered in the evaluation of our mining technique. After updating the mined \( \mu \)TOSCA topology graph to actually reflect that they were using timeouts in service interactions, they all resulted to be smell-free when analyzed with \( \mu \)FRESHENER. This is actually what we expected by such applications, as they are reference implementations of microservice-based applications maintained by a community of experts. This is another reason why we decided to consider a real-world, production-ready application developed and maintained by an IT company we are cooperating with when evaluating the support given by our methodology for analyzing and refactoring microservice-based architectures.

On the analysis and refactoring side, it is worth stressing that our methodology enables identifying and resolving the smells affecting the \textit{architecture} of a microservice-based application. Our methodology indeed enables identifying the architectural smells affecting an application based on the interactions among the components forming an application, by also suggesting the architectural refactorings that enable resolving such architectural smells. The actual implementation of an architectural refactoring, that is, the concrete updates to be applied to the application sources, is left the application owner, much in the same way as the actual implementation of a design pattern is left to developers. The application owner can hence decide which refactoring is most suited for resolving a smell based on the application’s contextual requirements, therein included the cost for actually implementing a refactoring.

The above was actually the case, for example, when refactoring of the \textit{shared persistence} smell occurring in the application considered in Section 6.2, because of services \( s_6, s_7, s_8, \) and \( s_9 \) all accessing the same database \( d_a \). When deciding how to refactor the application to resolve the occurrence of such smell, the company evaluated the three possibilities,
concluding that merging the services was not possible, while splitting database $d_6$ was too costly and less maintainable if compared to introducing a data manager. This is why the company decided to resolve the shared persistence smell by implementing a new service $s_{13}$, acting as data manager for database $d_6$.

It is also worth noting that all the architectural smells that we discussed denote potential violations of key design principles of microservices. The occurrence of an architectural smell does not necessarily mean that one of such design principles is violated, hence not necessarily needing to be resolved by enacting one of the corresponding architectural refactorings. For instance, the lack of timeouts or circuit breakers in service-to-service interactions does not necessarily mean that the invoker is not handling possible failures of the invoked services, as the former may exploit other fault handling mechanisms (e.g., ad-hoc routines). Moreover, even if an architectural smell is indicating the actual violation of a design principle of microservices, the application owner may still consider to not apply any refactoring, for example, because the actual implementation of the corresponding updates is too expensive. Another possible reason for choosing to not apply any refactoring to resolve an architectural smell can be that an application architect intentionally structured the corresponding part of the application as that, due to some contextual requirement. For instance, she may have intentionally made an asynchronous message broker publicly accessible to external clients, to let them subscribe and receive messages in near real-time. If this is the case, she just ignores the corresponding No API Gateway smell, as an API gateway acting as “façade” for such broker might provide a bottleneck. The latter might worsen performances, hence breaking the contextual requirement of client receiving messages in near real time.

It is finally worth highlighting that, in any case, the overall process of manually specifying the microservice-based architecture of an application, detecting the architectural smells affecting such architecture, deciding whether to resolve them, and reasoning on which refactoring to apply is not easy. These tasks hence call for a support system for automatically mining the microservice-based architecture of an application, identifying the smells affecting the architecture of microservice-based applications, and exploring among multiple possible refactorings to resolve them. One such support system is precisely given by the $\mu$TOSCA toolchain.

8 | RELATED WORK

To the best of our knowledge, ours is the first toolchain enabling to automatically determine the architecture of microservice-based applications and providing support for automatically identifying and resolving the occurrence of architectural smells in microservice-based applications. It is also the first one doing so by exploiting an open standard, that is, the OASIS standard TOSCA, to actually represent the architecture of microservice-based applications. However, there exists solutions for modeling microservice-based architectures, for mining them, and for resolving architectural smells in such applications, which we separately discuss hereafter.

8.1 | Modeling the architecture of microservice-based applications

Several ad-hoc domain-specific languages have been proposed to model the architecture of microservice-based applications. For instance, Terzić et al. propose MicroDSL, a domain-specific language for modeling the specification of RESTish microservice-based architectures. The solution by Terzić et al. is however conceived for different purposes, that is, for enabling to automatically generate the sources and configuration needed to deploy a microservice-based application given its architecture. The latter is the main reason why MicroDSL focuses on only modeling services interacting throughout RESTish protocols to structure microservice-based applications. Given our different aims, we instead focus on enabling to model the different types of components that can be used to structure microservice-based architectures (i.e., not only services, but also communication patterns and databases), while at the same time abstracting from the actual communication protocols exploited to implement component interactions. This, together with the fact that we wished to exploit an open standard to model microservice-based architectures, is the reason why developed $\mu$TOSCA instead of reusing MicroDSL.

Similar considerations apply to other solutions for modeling the architecture of microservice-based applications, such as those conceived by Cardarelli et al. and Cockroft for different purposes. Cardarelli et al. provide a customizable approach for specifying, aggregating, and evaluating software quality attributes on microservice-based architectures, with the ultimate goal of evaluating the overall quality of the application. Cockroft instead proposes Spigo, a tool for analyzing protocol interactions in microservice-based architecture by simulating service calls and generating
performance statistics. Both Cardarelli et al.\textsuperscript{32} and Cockcroft\textsuperscript{33} share our idea of modeling microservice-based architectures as graphs, whose nodes represent software components and whose arcs represent component interactions. They however do not support modeling the edge of an architecture, nor distinguishing the actual type of software components, that is, whether a component is a service, communication pattern, or database. This, along with our willingness to exploit an open standard to model microservice-based architectures, is the reason why developed $\mu$TOSCA instead of reusing the modeling solutions proposed by Cardarelli et al.\textsuperscript{32} or by Cockcroft.\textsuperscript{33}

### 8.2 Mining the architecture of microservice-based applications

Recently, several solutions have been proposed to elicit the architecture of microservice-based applications. For instance, Ma et al.\textsuperscript{34} propose a solution for determining the service dependency graph representing the interactions occurring among the services forming a microservice-based application, based on the static analysis of the Java sources. Rademacher et al.\textsuperscript{35} illustrate a way to reconstruct the architecture of an existing microservice-based application by statically analyzing its source code under the different perspectives of domain experts, developers, and operators. Alshuqayran et al.\textsuperscript{36} propose a set of rules for mapping the source code of microservices to modeling constructs and a methodology that, based on such rules, allows to derive the architecture of microservice-based applications by statically analysing their source code. The solutions by Ma et al.,\textsuperscript{34} Rademacher et al.,\textsuperscript{35} and Alshuqayran et al.\textsuperscript{36} however differ from our solution for mining the architecture of microservice-based applications since they all follow a “white-box” approach, as they all require the source code of the software components forming a microservice-based application to be available. Our mining solution can instead work also in “black-box” scenarios, where the source code of such components is not available. We indeed only require the Kubernetes deployment of a microservice-based application to automatically determine its architecture. In addition, while our mining solution is fully automated, both those by Rademacher et al.\textsuperscript{35} and Alshuqayran et al.\textsuperscript{36} require application administrators to manually intervene while mining the architecture of microservice-based applications.

Similar considerations apply to MicroART,\textsuperscript{37,38} which also provides a semi-automatic support for determining the architecture of a microservice-based application. MicroART can however be considered a step closer to our solution: Even if it statically analyzes the source code to determine the services forming an application, it then dynamically runs and monitors such services to grasp the interactions occurring among them. MicroART is semi-automated as it always requires the application administrator to manually refine the obtained architecture by removing the infrastructure components used by the services forming an application (e.g., service discovery components) and the corresponding interactions. Our solution hence differs from MicroART since it fully automates the mining of the architecture of microservice-based applications, and since it can work also with applications whose source code is not available. In addition, our mining solution automatically distinguishes services and databases in an application from integration components implementing well-known integration patterns (e.g., message queues or load balancers), and it then enables automatically checking whether some architectural smells is affecting the mined architectures by means of its integration with $\mu$FRESHENER. This is something currently not featured by neither MicroART nor by the solutions proposed by Ma et al.,\textsuperscript{34} Rademacher et al.,\textsuperscript{35} or Alshuqayran et al.\textsuperscript{36}

It is also worth discussing the relation between our mining solution and the existing tooling for visualizing and monitoring Kubernetes-based application deployments. Kiali,\textsuperscript{39} KubeView,\textsuperscript{40} and WeaveScope\textsuperscript{29} are three different tools displaying the structure of applications deployed in Kubernetes. Their goal is to enable monitoring the Kubernetes-based deployment of generic applications. As a consequence, they only visualize the deployed Kubernetes objects (e.g., pods and services) and how they are interconnected. Our solution instead enables distinguishing among services, integration components, and databases forming the architecture of a microservice-based application, as well as to recognize whether component interactions involve some form of client-side service discovery. We have given a concrete example of such difference in Section 6.1.

Instana\textsuperscript{41} makes a step further in this direction, by enabling to visualize services and databases forming a microservice-based application deployed in Kubernetes, together with service-to-service and service-to-database interactions. Our solution however differs from Instana as we not only enable visualizing services and databases, but also recognizing whether some software component is implementing message-based integration patterns, as well as whether client-side service discovery is enacted in some service-to-service interaction. In addition, while Instana is a commercial and subscription-based tool, our solution is publicly available in a free and open-source implementation.

Finally, it is worth relating our solution with existing systems for distributed tracing in multiservice applications, for example, AppDash,\textsuperscript{42} Jaeger,\textsuperscript{43} and Zipkin,\textsuperscript{44} which are specifically tailored for microservice-based applications.
Distributed tracing systems allow to monitor and log the component interactions in a running deployment of a microservice-based application, provided that the application sources include invocations to the distributed tracing system itself, for example, by exploiting the OpenTracing API\textsuperscript{45} or the OpenTelemetry API.\textsuperscript{46} Distributed tracing systems hence follow a more intrusive approach than ours to elicit the component interactions in a running application deployment, as they require the tracing code to get included in the sources of the application components. As we aimed also at enabling to mine the architecture of applications whose sources are not available, we decided not to opt for distributed tracing. Our mining technique indeed automatically includes monitoring containers in the deployment of an application, without requiring any access to the sources of the application components.

### 8.3 Resolving architectural smells in microservice-based applications

Various existing contributions classify the possible architectural smells for microservices, for example, the reviews by Carrasco et al.\textsuperscript{47} or by Taibi and Lenarduzzi,\textsuperscript{8} and our industry-driven review.\textsuperscript{10} At the same time, to the best of our knowledge, ours is the first solution enabling to both identify and resolve the smells affecting the architecture of a microservice-based application, with such architecture being represented with an open standard, that is, the OASIS standard TOSCA.\textsuperscript{11}

Pigazzini et al.\textsuperscript{48} propose a first solution for automatically identifying three smells (viz., cyclic dependencies, hard-coded endpoints, and shared persistence) in microservice-based applications. Their solution builds on top of an existing tool for detecting architectural smells in software projects,\textsuperscript{49} which the authors adapted to work with microservice-based architectures. Similarly to our proposal, the authors pick the considered smells from an industry-driven review\textsuperscript{8} and show how to automatically identify such smells in existing applications. Besides considering a different set of smells, we also try to make a further step in supporting microservice architects: We indeed aim at also enabling architects to refactor their application to resolve identified smells, based on the best practices shared by practitioners and recapped in our industry-oriented multivocal review.\textsuperscript{10}

Balalaie et al.\textsuperscript{50} report on the patterns to follow while migrating an application to microservices to not incur in well-known smells. Haselböck et al.\textsuperscript{51} instead illustrate decision models supporting the design of microservice-based applications. Both contributions are based on the information retrieved by the authors from practitioners or industry-scale projects, which was organized into informal guidelines to be followed while designing microservice-based applications to not incur in well-known smells. We try to further support the design of “smell-free” microservice-based applications by providing a solution to systematically identify the architectural smells affecting an existing application and to reason on how to resolve such smells.

Solutions for systematically analyzing the architecture of microservice-based applications anyway exist, even if devised for different purposes. For instance, Camilli et al.\textsuperscript{52} present a Petri net-based solution to verify the runtime orchestration of microservice-based applications on top of Netflix’s Conductor.\textsuperscript{53} The proposed solution shares the our baseline idea of eliciting all interactions occurring among the software components forming a microservice-based application to analyze its architecture. The solution by Camilli et al.\textsuperscript{52} however differs from ours in its ultimate goal, due to which the authors focus on modeling a specific type of microservice-based applications, that is, those devised to run on Netflix’s Conductor. We instead support modeling and analyzing microservice-based applications in general, therein included those intended to run on Conductor.

Savchenko et al.\textsuperscript{54} present an approach to automate the testing of microservice-based applications. The approach by Savchenko et al.\textsuperscript{54} relates to ours because it enables systematically checking whether the interfaces of the microservices forming an application adhere to a given specification. At the same time, the approach by Savchenko et al.\textsuperscript{54} requires to run the microservice-based application to be tested. Our solution instead does not necessarily require to run such applications, unless application administrators wish to automatically determine the architecture of their applications by running $\mu$Miner.

It is also worth relating our solution for identifying and resolving architectural smells in microservice-based applications with existing solutions for detecting smells in classical services. García et al.\textsuperscript{55} and Sanchez et al.\textsuperscript{56} present two different approaches to automatically detect smells in the design of a single service (specified in UML and Archery, respectively). Arcelli et al.,\textsuperscript{57} Fontana et al.,\textsuperscript{49} and Vidal et al.\textsuperscript{58} propose three different for identifying smells in the sources of a service and for refactoring such sources to resolve the identified smells. However, all such solutions differ from our solution as they target the design of a single service. We instead focus on identifying and resolving the architectural smells due to the interaction occurring among the multiple components structuring a microservice-based application. To some extent, the solutions by García et al.,\textsuperscript{55} Sanchez et al.,\textsuperscript{56} Arcelli et al.,\textsuperscript{57} Fontana et al.,\textsuperscript{49} and Vidal et al.\textsuperscript{58} are
complementary to ours: By exploiting both such solutions and ours, one could indeed identify and resolve both the
smells due to the internals of a service and those due to the interactions occurring among the components structuring a
microservice-based application.

Similar considerations apply to the microservice-oriented solutions proposed by Hassan and Bahsoon\textsuperscript{59} and by Hassan
et al.\textsuperscript{60} Both such solutions focus on analyzing a single microservice to determine whether its granularity is optimal or
whether it needs some adaptation to rightsize its granularity. Our solution instead focuses analyzing the interactions
among all microservices forming an application to identify and resolve architectural smells.

Finally, it is worth relating our solution with the language-based approach to structuring microservices introduced by
Guidi et al.\textsuperscript{61} Starting from the idea that service interactions are the main mechanism to program the microservices form-
ing an application, Guidi et al.\textsuperscript{61} propose to develop microservice-based applications with Jolie, a language for developing
service compositions by programming their interactions. Our approach shares the same baseline idea, as we consider
service interactions as the basis for identifying architectural smells.

9  |  CONCLUSIONS

In this article, we have presented a set of solutions for mining, analyzing, and refactoring the architecture of
microservice-based applications to resolve the occurrence of architectural smells possibly violating key design prin-
ciples of microservices. More precisely, after proposing $\mu$TOSCA to enable modeling microservice-based architec-
tures with the OASIS standard TOSCA, we have presented a technique to automatically mine the $\mu$TOSCA topology
graph modeling the architecture of a microservice-based application. We have then presented a methodology
to systematically identify and resolve the architectural smells affecting a microservice-based application, based on
the formal analysis of the $\mu$TOSCA topology graph modeling its architecture. We have also presented the prototype
tools ($\mu$Miner and $\mu$Freshener) implementing our proposals, which we also exploited to evaluate our proposals
in practice.

$\mu$Miner and $\mu$Freshener constitute the integrated $\mu$TOSCA toolchain, which is of practical value for both
researchers and practitioners working with microservices. $\mu$Miner can indeed be used to automatically mine the archi-
tecture of an existing microservice-based application, with such architecture being represented in a TOSCA file as a $\mu$TOSCA topology graph. This enables visualizing the possibly complex architecture of an existing microservice-based application, eliciting all services, integration components, and databases forming the application, together with all interactions occurring among them. This is of practical value when working with microservices, as manually representing the architecture of a microservice-based application can be complex, error-prone, and time-consuming.\textsuperscript{5}

The TOSCA file specifying the $\mu$TOSCA topology graph modeling the architecture of a microservice-based application
can then be imported in $\mu$Freshener. The latter enables further editing the architecture modeling to include additional
information, if needed, as well as editing $\mu$TOSCA topology graphs from scratch. $\mu$Freshener also enables automati-
cally identifying the architectural smells affecting the microservice-based architecture modeled by the $\mu$TOSCA topology
graph under editing. In case any smell is detected, $\mu$Freshener enables reasoning on which architectural refactoring to
enact to resolve the occurrence of such smell. This is also of practical value for researchers and practitioners working with
microservices, as we concretely exemplified in the industry-driven case study we presented in Section 6.2: $\mu$Freshener
can indeed be used to obtain $\mu$TOSCA topology graphs representing “smell-free” microservice-based architectures, which
researchers and practitioners can exploit as recipes to actually refactor the source code of their microservice-based applica-
tions. As we already noticed, the actual implementation of an architectural refactoring (i.e., the concrete updates to be
applied to the application sources) is left to the application owner, much in the same way as the actual implementation
of a design pattern is left to the developers. The application owner can hence decide which refactoring is most suited for
resolving a smell based on the application’s contextual requirements, therein included the cost for actually implementing
a refactoring.

We see multiple possible extensions for the solutions we presented, some of which we already plan to pursue. Firstly,
we plan to enhance our mining technique by enabling it to also recognize circuit breakers and timeouts exploited in
component interactions. For instance, we plan to investigate how to mine information within the pod running a service
to identify whether sidecar containers (e.g., Istio sidecars)\textsuperscript{63} are used to implement circuit breaking policies or to set
timeouts for the interactions outgoing from the services running in a pod and targeting other components running in
other pods. We also plan to enhance our mining technique by enabling it to recognize databases not only based on the
Docker image they run but also based on the communication protocols used to access their data, for example, by checking
whether the MongoDB wire protocol\textsuperscript{63} or the Redis serialization protocol\textsuperscript{64} are used to communicate with a component, we may recognize that the latter is implementing a MongoDB or Redis database, respectively.

In addition, our mining technique is currently devised to start from the Kubernetes deployment of a microservice-based application. We plan to extend our solution to work also with other container-based orchestration systems, for example, Docker Swarm, OpenShift. In this perspective, we exploited the strategy design pattern\textsuperscript{22} to make $\mu$MINER pluggable by design, with the Kubernetes-based architecture mining plugged as a strategy supported by the implementation itself. Adding support for other existing container-based orchestration systems hence just requires to implement the corresponding strategy and to plug it into the current implementation.

We also plan to extend the set of architectural smells that can be identified and resolved by our methodology, extending the support to other smells and refactoring classified in our industry-driven review,\textsuperscript{10} as well as to those described by Carrasco et al.\textsuperscript{47} or by Taibi and Lenarduzzi.\textsuperscript{8} This can be done, for example, by exploiting the inheritance featured by the TOSCA types system to extend our $\mu$TOSCA types to model additional entities, by defining the conditions to detect additional smells based on such entities, and by then adapting $\mu$FRESHENER to support the additional entities and smells. As a concrete example, we have already extended $\mu$TOSCA with a type for grouping nodes to represent team assignment (i.e., which components are assigned to which team), and we now plan to formalize the team-related architectural smells described in our industry-oriented multivocal review\textsuperscript{10} and to correspondingly extend $\mu$FRESHENER to identify and resolve such smells.

We also plan to extend our analysis and refactoring methodology to natively consider the fact that container orchestrators impact on the actual behavior of a microservice-based application when enacting its deployment. While presenting our mining technique, we already illustrated how encapsulating a component behind a Kubernetes service corresponds to adding a message router in front of it, which takes incoming requests and load balances them across the possibly multiple replicas of the component. Hence, even if originally there was a direct, endpoint-based interaction between two services, this is no more the case when deploying them as Kubernetes services. This is just an example for a given deployment with Kubernetes, but similar observations apply to other possible deployments with other container orchestrators. We plan to make our methodology parametric with respect to the actual deployment of an application, to avoid highlighting smells that will automatically get solved once enacting such deployment, for example, endpoint-based service interaction smells “disappear” when components are deployed as Kubernetes service. On the other hand, we plan to exploit our methodology to devise a recommender for the container orchestrator best suited to deploy an application. The idea is to suggest choosing the container orchestrators that make the architectural smells affecting an application “disappear” when enacting its deployment.

Finally, it is worth highlighting that our mining, analysis, and refactoring solutions are computationally expensive, with the mining and analysis techniques polynomially growing with the size of the microservice-based applications. The more components and interactions are structuring the architecture of a microservice-based application, the more computing power and time are needed to mine and analyze such architecture. While this is negligible for applications involving tens of components, such as those considered in Section 6, it may become a concrete issue when considering large-sized microservice-based applications involving hundreds of interacting components. The latter may also get hard to visualize in $\mu$FRESHENER, due to the huge amount of components and interactions. We hence plan to enhance our mining, analysis, and refactoring solutions to scale to large-scale applications, as well as to engineer $\mu$MINER and $\mu$FRESHENER accordingly. Extending our solutions and prototypes to feature a team-wise usage can be a concrete solution in this direction. Different teams working on different subsets of microservices would indeed be able to focus only on the portion of the application they work on, hence again reducing the mining, analysis, and refactoring problems from hundreds to tens of microservices, which we already shown to be manageable in Section 6.

All links were last followed 24 March 2021.

**ORCID**

Jacopo Soldani \@ https://orcid.org/0000-0002-2435-3543

Davide Neri \@ https://orcid.org/0000-0002-4758-4328

**REFERENCES**

1. Fowler M, Lewis J. Microservices; 2014. http://martinfowler.com/articles/microservices.html.
2. Newman S. *Building Microservices*. 1st ed. Sebastopol, CA: O’Reilly Media Inc; 2015.
3. Brogi A, Neri D, Soldani J. A microservice-based architecture for (customisable) analyses of Docker images. *Softw Pract Exper.* 2018;48(8):1461-1474. https://doi.org/10.1002/spe.2583.
4. De Iasio A, Zimeo E. A framework for microservices synchronization. *Softw Pract Exper.* 2021;51(1):25-45. https://doi.org/10.1002/spe.2877.

5. Soldani J, Tamburri DA, Van Heuvel WJ. The pains and gains of microservices: a systematic grey literature review. *J Syst Softw.* 2018;146:215-232. https://doi.org/10.1016/j.jss.2018.09.082.

6. Zimmermann O. Microservices tenets. *Comput Sci Res Develop.* 2017;32(3–4):301-310. https://doi.org/10.1007/s00450-016-0337-0.

7. Joseph CT, Chandrasekaran K. Straddling the crevasse: a review of microservice software architecture foundations and recent advances. *Softw Pract Exper.* 2019;49(10):1448-1484. https://doi.org/10.1002/spe.2729.

8. Taibi D, Lenarduzzi V. On the definition of microservice bad smells. *IEEE Softw.* 2018;35(3):56-62. https://doi.org/10.1109/MS.2018.2141031.

9. Mora M, O’Connor RV, Tsui F, Marx GJ. Design methods for software architectures in the service-oriented computing and cloud paradigms. *Softw Pract Exper.* 2018;48(2):263-267. https://doi.org/10.1002/spe.2547.

10. Neri D, Soldani J, Zimmermann O, Brogi A. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Softw Intens Cyber-Phys Syst.* 2020;35(1):3-15. https://doi.org/10.1007/s00450-019-00407-8.

11. OASIS TOSCA simple profile in YAML, Version 1.3 OASIS Standard; 2020.

12. Hassan S, Bahsoon R, Kazman R. Microservice transition and its granularity problem: a systematic mapping study. *Softw Pract Exper.* 2020;50(9):1651-1681. https://doi.org/10.1002/spe.2869.

13. Hohe G, Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* 1st ed. Boston, MA: Addison-Wesley Longman Publishing Co.Inc; 2003.

14. Muntoni G, Soldani J, Brogi A. Mining the architecture of microservice-based applications from their kubernetes deployment. Paper presented at: Proceedings of the Advances in Service-Oriented and Cloud Computing, Workshops of ESOC; 2020; Springer International Publishing, Cham, Switzerland.

15. Brogi A, Neri D, Soldani J. Freshening the air in microservices: resolving architectural smells via refactoring. Paper presented at: Proceedings of the Service-Oriented Computing, ICSOC 2019 Workshops; 2020;17-29; Springer International Publishing, Cham, Switzerland.

16. Kuberentes documentation; 2020. https://kubernetes.io/docs.

17. Larsson L, Tärneberg W, Klein C, Elmroth E, Kihl M. Impact of etcd deployment on Kubernetes, Istio, and application performance. *Softw Pract Exper.* 2020;50(10):1986-2007. https://doi.org/10.1002/spe.2885.

18. Google Inc Docker Hub; 2020. https://hub.docker.com.

19. The Wireshark Foundation Wireshark; 2020. https://www.wireshark.org.

20. Richardson C. *Microservices Patterns.* 1st ed. Shelter Island, NY: Manning Publications; 2018.

21. Korab J. *Understanding Message Brokers.* 1st ed. Sebastopol, CA: O’Reilly Media; 2017.

22. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st ed. Boston, MA: Addison-Wesley Longman Publishing Co.Inc; 1995.

23. Familiar B. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions.* Berkeley, CA: Apress; 2015.

24. Indrasiri K. Microservices in practice: from architecture to deployment DZone; 2019. https://dzone.com/articles/microservices-in-practice-1.

25. Richardson C. *Microservices patterns.* 1st ed. Shelter Island, New York: Manning Publications; 2018.

26. Jamshidi P, Pahl C, Mendonca NC, Lewis J, Tilkov S. Microservices: the journey so far and challenges ahead. *IEEE Softw.* 2018;35(3):24-35. https://doi.org/10.1109/MS.2018.2141039.

27. Google Online boutique: a cloud-native microservices demo application; 2020. https://github.com/GoogleCloudPlatform/microservices-demo.

28. Weaveworks Sock shop: a microservices demo application; 2020. https://microservices-demo.github.io.

29. Weaveworks WeaveScope; 2020. https://www.weave.works/oss/scope.

30. Kopp O, Binz T, Breitenbüber L, Leymann F. Winery — a modeling tool for TOSCA-based cloud applications. Paper presented at: Proceedings of the 11th International Conference on Service-Oriented Computing - Volume 8274 ICSOC; 2013:700-704; Springer-Verlag, Berlin, Heidelberg.

31. Terzić B, Dimitrievski V, Kordić S, Milosavljević G, Luković I. Development and evaluation of MicroBuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Inf Syst.* 2018;12(8–9):1034-1057.

32. Cardarelli M, Iovino L, Di Francesco P, Di Salle A, Malavolta I, Lago P. An extensible data-driven approach for evaluating the quality of microservice architectures. Paper presented at: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing; 2019:1225-1234; ACM, New York, NY.

33. Cockroft A. Spigo; 2016. https://github.com/adrianco/spigo.

34. Ma S, Fan C, Chuang Y, Lee W, Lee S, Hsueh N. Using service dependency graph to analyze and test microservices. Paper presented at: Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan: IEEE Computer Society; 2018:81-86.

35. Rademacher F, Sachweh S, Zündorf A. A modeling method for systematic architecture reconstruction of microservice-based software systems. *Enterprise, Business-Process and Information Systems Modeling, BPMDS 2020, EMMSAD 2020.* Cham: Springer International Publishing; 2020:311-326.
37. Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. MicroART: a software architecture recovery tool for maintaining microservice-based systems. Paper presented at: Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops, ICSA 2017 Workshops, Gothenburg, Sweden: IEEE Computer Society; 2017:298-302.

38. Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. Towards recovering the software architecture of microservice-based systems. Paper presented at: Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops, ICSA 2017 Workshops, Gothenburg, Sweden: IEEE Computer Society; 2017:46-53.

39. Kiali Development Group Kiali; 2020. https://kiali.io.

40. Coleman Ben. KubeView; 2020. https://github.com/benc-uk/kubeview.

41. Instana Development Group Instana; 2020. https://www.instana.com.

42. SourceGraph AppDash; 2020. https://github.com/sourcegraph/appdash.

43. Jaeger Authors Jaeger; 2020. https://www.jaegertracing.io.

44. Jaeger Authors Zipkin; 2020. https://zipkin.io.

45. OpenTelemetry Authors OpenTelemetry; 2020. https://opentelemetry.io.

46. Carrasco A, Bladel BV, Demeyer S. Migrating towards microservices: migration and architecture smells. Paper presented at: Proceedings of the 2nd International Workshop on Refactoring, IWoR; 2018:1-6; ACM, New York, NY.

47. Pigazzini I, Fontana FA, Lenarduzzi V, Taibi D. Towards Microservice Smells Detection. In: Proceedings of the 3rd International Conference on Technical Debt TechDebt 2020, Association for Computing Machinery; 2020:92-97; New York, NY.

48. Fontana FA, Pigazzini I, Roveda R, Tamburri D, Zanoni M, Di Nitto E. Arcan: a tool for architectural smells detection. Paper presented at: Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops, ICSA 2017 Workshops, Gothenburg, Sweden: IEEE Computer Society; 2017:282-285.

49. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T. Microservices migration patterns. Softw Pract Exper. 2018;48(11):2019-2042. https://doi.org/10.1002/spe.2608.

50. Haselböck S, Weinreich R, Buchgeher G. Decision models for microservices: design areas, stakeholders, use cases, and requirements. Software Architecture, ECSA. Vol 2017. Cham, Switzerland: Springer International Publishing; 2017:155-170.

51. Camilli M, Bellettini C, Capra L, Monga M. A formal framework for specifying and verifying microservices based process flows. Software Engineering and Formal Methods, SEFM. Vol 2017. Cham, Switzerland: Springer International Publishing; 2018:187-202.

52. Netflix Conductor; 2020. https://netflix.github.io/conductor.

53. Savchenko DI, Radchenko GI, Taipole O. Microservices validation: mjolnir platform case study. Paper presented at: Proceedings of the 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia: IEEE Computer Society; 2015:235-240.

54. Garcia J, Popescu D, Edwards G, Medvidovic N. Identifying architectural bad smells. Paper presented at: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, CSMR, Kaiserslautern, Germany: IEEE Computer Society. 2009:255-258.

55. Sanchez A, Barbosa LS, Madeira A. Modelling and verifying smell-free architectures with the archeity language. Software Engineering and Formal Methods, SEFM. Vol 2015. Cham, Switzerland: Springer International Publishing; 2015:147-163.

56. Arcelli D, Cortellessa V, Pompeo DD. Automating performance antipattern detection and software refactoring in UML models. Paper presented at: Proceedings of the 2019 International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China: IEEE Computer Society; 2019:639-643.

57. Hasnain S, Bahsoon R. Microservices and their design trade-offs: a self-adaptive roadmap. Paper presented at: Proceedings of the 2016 International Conference on Services Computing, SVC 2016, San Francisco, CA: San Francisco, CA; 2016:813-818.

58. Hassan S, Ali N, Bahsoon R. Microservice ambients: an architectural meta-modelling approach for microservice granularity. Paper presented at: Proceedings of the 2017 International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden: IEEE Computer Society; 2017:1-10.

59. Guidi C, Lanese I, Mazzara M, Montesi F. Microservices: a language-based approach. Present and Ulterior Software Engineering, Computer Science. Cham, Switzerland: Springer; 2017:217-225.

60. MongoDB Documentation MongoDB wire protocol; 2020. https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/.

61. Redis.io Redis serialization protocol; 2020. https://redis.io/topics/protocol.

How to cite this article: Soldani J, Muntoni G, Neri D, Brogi A. The \( \mu \)TOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. Softw Pract Exper. 2021;51:1591–1621. https://doi.org/10.1002/spe.2974