SN Computer Science (2023) 4:304
https://doi.org/10.1007/s42979-023-01721-4

ORIGINAL RESEARCH

Offline Mining of Microservice-Based Architectures (Extended Version)

Jacopo Soldani[1] · **Javad Khalili**[1] · **Antonio Brogi**[1]

Received: 1 August 2022 / Accepted: 1 February 2023 / Published online: 1 April 2023

Abstract
Designing applications adhering to the key design principles of microservice-based architectures (MSAs) enables fully exploiting the potentials of cloud computing platforms. A specification of an application's MSA can help determining whether it adheres to such principles, and reasoning on how to refactor it when this is not the case. However, manually generating such a specification is complex and costly, mainly due to the multitude of heterogeneous software services and service interactions forming an MSA. The main objective of this article is to automate the generation of the specification of an existing MSA. We introduce an offline technique for automatically mining the specification of an MSA from its Kubernetes deployment. The mined MSA is expressed in μ TOSCA, a microservice-oriented profile of the OASIS standard TOSCA. We also provide an open-source prototype implementation of the proposed mining technique, called μ TOM. Four case studies based on four different third-party applications show that our technique can effectively mine the MSAs of existing applications, being it more accurate than its state-of-the-art competitor. The proposed offline mining technique can help researchers and practitioners working with microservices, by enabling them to automatically mine the MSAs of their applications. The obtained MSAs can then be visualised and analysed with existing tools to enhance their adherence to the key design principles of MSAs.

Keywords Microservices · Microservices architecture · Software architecture mining

Introduction

Microservice-based architectures (MSAs) enable realising so-called *cloud-native* applications, viz., applications architected to fully exploit the potentials of cloud computing platforms [1]. As a result, MSAs have become commonplace for cloud-based applications. For instance, Amazon, Netflix, or Twitter are already exploiting MSAs to deliver their businesses [2].

MSAs are essentially service-oriented architectures satisfying some additional key design principles, e.g., ensuring services' independent deployability and horizontal scalability, or isolating failures [3]. It is hence crucial to determine whether a service-based application adheres to the key design principles of MSAs, and understanding how to refactor an application to resolve possible violations of such key design principles [4].

μ TOSCA and μ Freshener [5] enable modelling, analysing, and refactoring the architecture of a service-based application, to enhance its adherence to the key design principles of MSAs. μ TOSCA is a model enabling to specify MSAs with the human- and machine-readable OASIS standard TOSCA [6]. MSAs are represented by typed directed graphs, called *topology graphs*, where nodes model the services, integration components (e.g., load balancers or message queues), and databases forming an MSA. Directed arcs represent the interactions among such components.

μ Freshener [5] then provides a visual enviroment to manually edit the μ TOSCA specification of the MSA of an existing application, viz., its modelling as a μ TOSCA topology graph. Specified MSAs can then be automatically analysed to check whether the application includes some known architectural smells, viz., possible symptoms of violations of MSAs' key design principles. μ Freshener also enables

This article is part of the topical collection "Advances on Cloud Computing and Services Science" guest edited by Donald F. Ferguson, Claus Pahl and Maarten van Steen.

✉ Jacopo Soldani
jacopo.soldani@unipi.it

Javad Khalili
javad.khalili443@gmail.com

Antonio Brogi
antonio.brogi@unipi.it

[1] Department of Computer Science, University of Pisa, Largo B. Pontecorvo, 3, 56127 Pisa, Italy

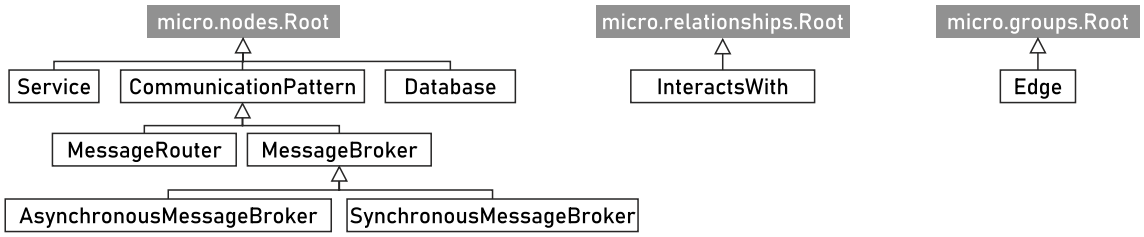


Fig. 1 The node types, relationship types, and group types defining μ TOSCA. The corresponding definitions in TOSCA are publicly available on GitHub at https://di-unipi-socc.github.io/microTOSCA/microTOSCA.yml

reasoning on how to refactor an application to resolve identified architectural smells, based on applying practitioner-shared refactorings known to resolve their occurrence [7].

At the same time, manually specifying the whole MSA of an existing application is a complex, time-consuming, and error-prone process, even in a visual environment like that provided by μ Freshener [4]. This is mainly because of the multitude of heterogeneous software services forming an MSA, and of the many complex interactions occurring among them to deliver the application's businesses [2]. For this reason, the main objective of this article is enabling to automatically generate the μ TOSCA specification of an existing MSA.

In this perspective, we propose a novel technique for mining the μ TOSCA specification of the MSA of an application, which starts from the Kubernetes deployment of an application, configured to also exploit Istio [8] and Kiali [9], two Kubernetes-native tools for proxying deployed services and monitor their interactions. It then processes, *offline*, the Kubernetes manifest files specifying the application deployment and the Istio-based proxying of its services, as well as a graph generated by Kiali in any former run of the application, e.g., its production run. The Kiali graph models the deployed software components as nodes, and their monitored interactions as directed arcs. Given such inputs, our technique can automatically mine the MSA of an application in two steps. It first elicits the software components and their interactions, producing a first draft of the MSA of an application. The draft is then refined by distinguishing services from integration components and databases, and by characterising the mined interactions, e.g., determining whether circuit breakers or timeouts are used therein. The refined architecture is finally marshalled to μ TOSCA.

To illustrate the feasibility of the proposed mining technique, we present an open source prototype implementation, called μ TOM (μ TOSCA *Offline Miner*). We also show how we used μ TOM to assess our technique by applying it to mine MSAs in four case studies based on four existing, third-party applications, viz., *Sock Shop* [10], *Online Boutique* [11], *Robot Shop* [12], and *Book Info* [13]. The case studies show that μ TOM effectively mines the MSAs of the considered applications, and that it is more accurate in mining MSAs if compared with μ Miner, viz., the state-of-the-art competitor that we presented in our previous work [14].

Our mining technique and its prototype implementation can be of practical value to researchers and practitioners working with microservices. They can indeed be exploited to automatically mine the μ TOSCA specification of the MSAs of existing applications, by simply processing their existing Kubernetes deployments, rather than requiring to deploy and run them in suitably configured testing environments, as instead required by μ Miner [14]. In addition, the MSAs obtained with our mining technique can be visualised and analysed with μ Freshener [5] to identify and resolve the architectural smells therein, to enhance their adherence to the key design principles of MSAs.

The rest of this article is organised as follows. Section ("Background") provides the necessary background on μ TOSCA, Kubernetes, Istio, and Kiali. Section ("Mining MSAs") presents our technique for mining MSAs offline. Section ("Prototype") introduces μ TOM, the open source prototype implementation of our mining technique. Section ("Case Studies") illustrates four case studies assessing our technique and discusses the accuracy of μ TOM in mining the considered MSAs. Finally, Sects. ("Related Work") and ("Conclusions") discuss related work and draw some concluding remarks, respectively.[1]

Background

μ TOSCA

The μ TOSCA type system (Fig. 1) allows specifying MSAs as typed topology graphs in TOSCA, the *Topology*

[1] This article extends [15] by providing a more detailed description of our mining technique, showing its application in two new case studies (Sects. "Sock Shop" and "Online Boutique"), and discussing the accuracy of and setup needed to run our mining technique (Sect. "Discussion").

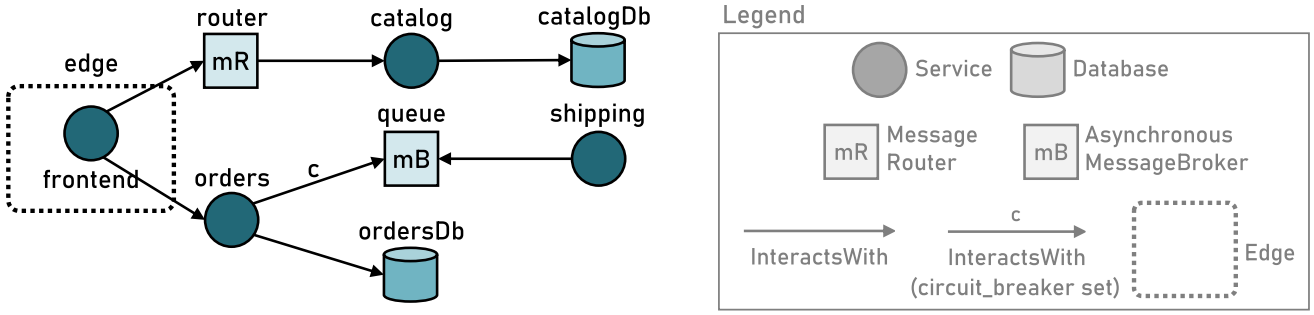


Fig. 2 An example of μ TOSCA topology modelling the architecture of an application

and Orchestration Specification for Cloud Applications [6]. Topology nodes model the services, communication patterns, or databases in an MSA. A Service runs some business logic, e.g., a service managing users' orders in an e-commerce application. A CommunicationPattern implements message-based integration pattern [16], viz., MessageRouter and MessageBroker, which decouples the communication among two or more components. MessageBrokers are also 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, e.g., a database of orders in an e-commerce application.

Directed arcs instead model the interactions among the components in an MSA, throughout InteractsWith relationships. Such relationships can be further characterised by setting three boolean properties, viz., `c`, `t`, and `d`. The properties `c` and `t` indicate that the source node is interacting with the target node via a circuit breaker or by setting proper timeouts, respectively. The property `d` instead indicates that the endpoint of the target of the interaction is dynamically discovered (e.g., with service discovery).

Finally, nodes can be added to an Edge group. The latter specifies the application components that are publicly accessible from outside of the application, namely those components that can be directly accessed by external clients.

Example. Figure 2 displays an example of μ TOSCA topology modelling the MSA of a toy e-commerce application. The application includes four services, i.e., frontend (accessible by external clients), orders, payment, and shipping. It is then completed by two integration components, i.e., router and queue, and two databases, i.e., catalogDb and ordersDb. The frontend allows browsing the catalogue of available products, by interacting with catalog. The actual instance of catalog used to access the catalogDb is dynamically discovered by a message router implementing server-side service discovery. The frontend also allows to place orders, by interacting with orders. The latter allows to upload new product orders, which are stored in ordersDb, and which are also enqueued in the asynchronous message broker implementing the queue of orders to be shipped. A circuit breaker is set to let orders tolerate the possible failures of the queue of orders. Finally, the queue is consumed by the service shipping, which pulls orders from the queue and proceeds with their shipping.

Kubernetes

Kubernetes allows deploying and managing multi-service applications in distributed clusters. Such a deployment and management is realised by orchestrating *pods*, which constitute Kubernetes' deployment units. A pod is a deployable instance of an application service, which is shipped within a single container or in several tightly coupled containers. A pod can actually encapsulate multiple Docker containers that need to share the same resources, e.g., when a containerised service is accompanied by "sidecar" containers monitoring it or proxying its communications.

Pod instances are deployed and managed with Kubernetes *controller*s. The latter allow to spawn and manage pod instances from pod templates, which are included in *workload* resource specifications, e.g., *Deployment*s, *StatefulSet*s, and *ReplicaSet*s. The latter specify the Docker containers running in a pod, their target state, as well as the number of replicas of the pod that must be deployed. Kubernetes controllers then ensure that the specified number of replicas of a pod continue to run on a cluster, with each pod instance reaching and maintaining its target state.

Replicated pods can be accessed through Kubernetes *service*s, which define their load balancing policies. A Kubernetes service implements a message routing component, balancing requests among the pods it manages according to the specified balancing policy. Kubernetes services can be of multiple types, depending on whether they should be accessible only within the Kubernetes cluster (viz., *ClusterIP* services), or whether they should be exposed to external clients (viz., *NodePort* or *LoadBalancer* services).

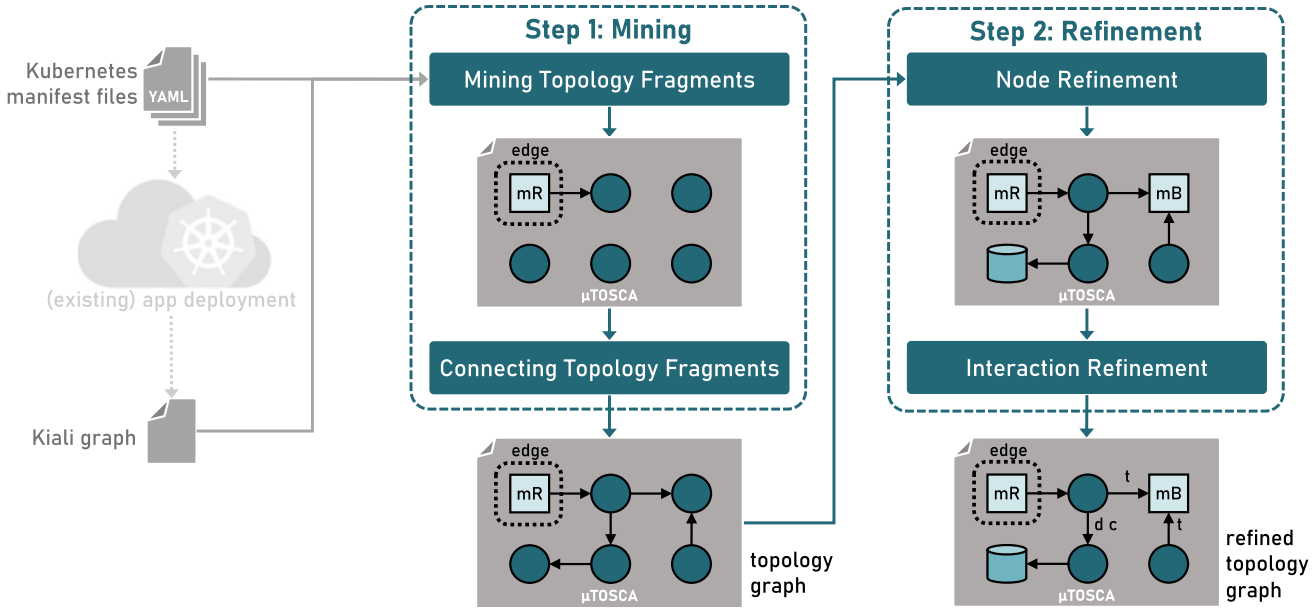


Fig. 3 Our two-steps technique for mining MSAs from their Kubernetes deployment

Istio and Kiali

Istio and Kiali are two Kubernetes-native tools for controlling and monitoring service interactions. Istio includes so-called *envoy* proxies in a Kubernetes deployment, which are deployed alongside application services to control how they interact with each other. This is done by specifying *VirtualService*s or *DestinationRule*s, which define how to route a message to its destination. This includes, e.g., indicating whether timeouts or circuit breakers are used to avoid the sender to continue waiting for an answer when the receiver has failed.

Kiali is an observability console, which comes natively integrated with Istio. It exploits Istio envoy proxies to store proxied interactions, so as to trace the interactions among deployed services. Each interaction is stored together with its metadata, including the source and target Kubernetes workloads or services, and whether the interaction successfully completed. Kiali then exploits such interactions to build different types of graphs, which enable visualising them at different abstraction levels in the Kiali dashboard, and which can be exported to JSON graph data files. In the rest of this article, we consider Kiali graph modelling monitored service interactions, viz., *service* graphs.[2]

Mining MSAs

The overall flow of our technique for mining MSAs is illustrated by the pipeline in Fig. 3, which starts from the Kubernetes manifest files specifying the application deployment and a JSON graph data file specifying the graph generated by Kiali while monitoring an existing application deployment. Such inputs are processed by a first step, called the *mining* step, which consists of two other substeps, viz., *mining topology fragment* and *connecting topology fragments*. They essentially elicit the nodes and interactions forming the target MSAs, and produce a first corresponding μ TOSCA topology graph. The graph is then passed to the *refinement* step, which also consists of two substeps, viz., *node refinement* and *interaction refinement*. The *node refinement* subset determines whether mined nodes model databases or message brokers, while the *interaction refinement* substep characterises the mined relationships by indicating whether dynamic discovery, circuit breakers, or timeouts are used in the corresponding interactions.

We hereafter illustrate the above described steps in more detail, to show how they enable building the μ TOSCA topology graph modelling a mined MSA. In doing so, we denote the mined μ TOSCA topology graphs following the notation from our previous work [4]. Namely, we represent the μ TOSCA topology graph modelling a mined MSA as a triple $A = \langle N, R, E \rangle$, where:

- N is the set of typed topology nodes, with the name and type of each node $n \in N$ denoted by $n.\mathsf{name}$ and $n.\mathsf{type}$, respectively;

[2] https://kiali.io/docs/features/topology.

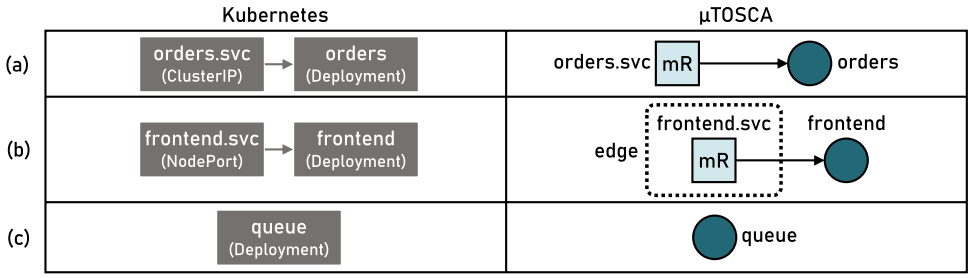


Fig. 4 Examples of topology fragments mined from Kubernetes services and workloads.

- $R \subseteq N \times N \times 2^P$ is the set of node relationships, with $P = \{\mathsf{c}, \mathsf{d}, \mathsf{t}\}$ being the set of properties that can be used to characterise a relationship, viz., (c) circuit breaker set, (d) dynamic discovery used, or (t) timeout set;
- $E \subseteq N$ is the subset of nodes in the Edge group, namely those nodes that are publicly accessible to external clients.

Step 1: Mining

The mining step processes the available inputs to determine the nodes and interactions forming the target MSA. Firstly, μ TOSCA topology fragments are mined from the Kubernetes manifest files, by essentially mapping Kubernetes entities to μ TOSCA nodes. The topology graph is then completed by connecting mined topology fragments based on the runtime interactions contained in the Kiali graph.

Mining Topology Fragments. Topology fragments are extracted from the Kubernetes manifest files by first mapping the workloads and services specified therein to μ TOSCA nodes. Each Kubernetes workload specifies the pod configuration for a component of the target MSA, by indicating the Docker container from which it runs and its target configuration (Sect. "Kubernetes"). Therefore, each Kubernetes workload is mapped to a μ TOSCA node of type Service. The type may change in the refinement step, if the workload is used to deploy an integration component or database.

A Kubernetes service instead implements a message routing component balancing the traffic sent to the replicas of the pod they manage, specified by a Kubernetes workload (Sect. "Kubernetes"). They are hence mapped to μ TOSCA nodes of types MessageRouter, which are directly specified to interact with the Service node corresponding to the workload they manage. In addition, if a Kubernetes service is a *NodePort* or *LoadBalancer*, its corresponding MessageRouter node is placed within the Edge group. This reflects the fact that *NodePort* or *LoadBalancer* services can be invoked by external clients.

More formally, the *mining topology fragment* substep generates a graph $A = \langle N, R, E \rangle$ by processing the Kubernetes manifest files as follows:

- Each Kubernetes workload w is modelled by a node $n \in N$ such that n.name is the name of w and n.type = Service.
- Each Kubernetes service s is modelled by a node $n \in N$ such that n.name is the name of s and n.type = MessageRouter. If the Kubernetes service s is a *NodePort* or *LoadBalancer*, the node n is also added to E.
- Each pairing of a Kubernetes service s with a workload w (managed by s) is modelled by a directed arc $\langle n, m, \emptyset \rangle \in R$, with $n \in N$ being the node modelling s and $m \in N$ being the node modelling w.

Example. Figure 4 illustrates three examples of application of our topology fragment mining. In case (a), a Kubernetes *ClusterIP* service orders.svc manages the *Deployment* workload running the service orders. By applying our node mining technique, we obtain a MessageRouter node modelling the Kubernetes service, which InteractsWith the Service node modelling the workload. Case (b) is similar, with the only difference that the MessageRouter node is placed in the Edge group, since Kubernetes *NodePort* services are exposed to external clients. Finally, case (c) considers a *Deployment* workload used to deploy a message queue, without any Kubernetes service balacing its load. In this case, we obtain a singleton Service node. Case (c) also provides an example of μ TOSCA node that may be typed as Service only temporarily: the type of queue might be changed to AsynchronousMessageBroker, if it actually implements an asynchronous message broker (Sect. "Step 2: Refinement").

Connecting Topology Fragments. The topology fragments obtained from the *mining topology fragments* substep are then interconnected to model the runtime interactions occurring among the components they model. This is done by parsing the Kiali graph, which explicitly models the component interactions that were monitored in a former deployment of the application, e.g., in its production deployment.

A monitored interaction is represented as an *edge* in the Kiali graph, which connects the node corresponding

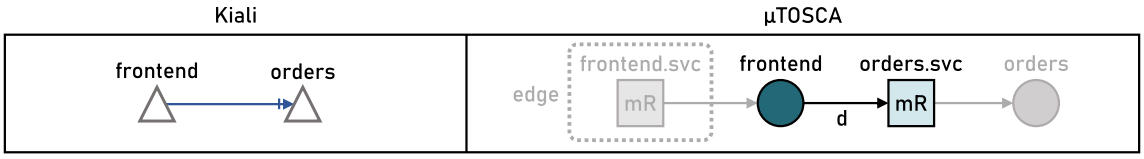


Fig. 5 Examples of InteractsWith relationship mined from the Kiali graph

Table 1 Official Docker images of software implementing a database or message broker

Databases	Message brokers
`cassandra`, `db2`, `iris`, `mariadb`, `mongo`, `mysql`, `neo4j`, `oracle`, `postgres`, `redis`, `sqlite`	`activemq`, `kafka`, `mosquito`, `nats`, `rabbitmq`

to the the Kubernetes workload that started the interaction to the Kubernetes service that was invoked.[3] Each *edge* is hence mapped to an InteractsWith relationship connecting the Service node modelling the starting workload to the MessageRouter node modelling the target Kubernetes service. In addition, the mined InteractsWith relationship are directly specified as enacting dynamic discovery, given that Kubernetes prescribes to invoke services based on their name and to rely on Kubernetes' native DNS to resolve the address of the actual host to contact.

More formally, the *connecting topology fragments* extends the graph $A = \langle N, R, E \rangle$ obtained with *mining topology fragments* substep, by generating a new graph $A' = \langle N, R', E \rangle$. The extended graph A' includes all formerly mined relationships, viz., $R \subseteq R'$, plus a new relationship modelling each monitored interaction. The interaction from a workload w to a Kubernetes service s (denoted by an arc in the Kiali graph) is modelled by a directed arc $\langle n, m, \{\mathsf{d}\} \rangle \in R'$, with $n \in N$ being the node modelling w and $m \in N$ being the node modelling s. The label d is instead used to denote that the modelled interaction enacts dynamic discovery, as described above.

Example. Figure 5 illustrates an example of mined InteractsWith relationship, which connects two of the topology fragments in Fig. 4. The Kiali graph specifies that the Kubernetes workload running the frontend service interacted with the Kubernetes service managing the replicated orders service. This is modelled by including an InteractsWith relationship connecting the corresponding μTOSCA nodes, viz., the Service frontend and the MessageRouter orders.svc.

[3] Kiali unifies a Kubernetes service with the Kubernetes worklaod it manages, assuming that Kubernetes services are used to enact server-side service discovery, as recommended by Kubernetes documentation (https://kubernetes.io/docs/concepts/services-networking/service).

Step 2: Refinement

The *mining* step produces a "draft" of the μ TOSCA topology modelling the target MSA, in which all nodes and interactions are recognised, but associated with default types and properties. The objective of this step to suitably characterise the mined nodes and relationships.

Node Refinement. After the *mining* step, nodes are associated with either one of two types: MessageRouter or Service. Whilst nodes types as MessageRouter are truly routing messages in the Kubernetes deployment of an MSA (being them obtained from Kubernetes services), Service is used as the default type for all other nodes. Nodes initially typed as Service s may however implement other components than those running some business logic, namely databases or asynchronous message brokers. The objective of the *node refinement* substep is hence to identify such nodes and assign them with the corresponding μTOSCA type, viz., Database and AsynchronousMessageBroker.

Database and message brokers can be seen as "passive" components: despite they reply when being invoked by other components, they are not proactively invoking other components [4]. They hence appear as "sink nodes" in the mined μ TOSCA topology graph, meaning that they are targeted by InteractsWith relationships, whilst no such relationship outgoes from them. This intuition enables reducing the number of nodes to be processed for possible refinement: the *node refinement* substep indeed focuses on the Service nodes being sink nodes, and determines whether they should be rather typed as Database or AsynchronousMessageBroker. This is essentially done by looking at the Docker image running in the corresponding Kubernetes workload: if such image is one of the official Docker images for databases or message brokers (Table 1), then the node's type is changed to Database or AsynchronousMessageBroker, respectively. Otherwise, the node continues to be a Service.

More formally, the *node refinement* substep updates the μ TOSCA topology graph $A = \langle N, R, E \rangle$ obtained from the

(a)
```
kind: VirtualService
spec:
  hosts: [ orders ]
  http:
    - route: [ destination: { host: orders } ]
      timeout: 0.5s
```

(b)
```
kind: DestinationRule
spec:
  host: catalog
  trafficPolicy:
    conectionPool:
      tcp: { maxConnections: 1 }
      http: { http1MaxPendingRequests: 1, maxRequestsPerConnection: 1 }
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 1
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

Fig. 6 Examples of **a** timeouts and **b** circuit breakers defined with Istio

mining step, by generating a new graph $A' = \langle N', R, E \rangle$. The updated graph includes all formerly mined nodes, but the sink nodes of type Service, viz.,

$$\{n \in N \mid \exists \langle n, \cdot, \cdot \rangle \in R \vee n.\mathsf{type} \neq \mathsf{Service}\} \subseteq N'.$$

It then refines the sink nodes of type Service as described above, when possible. More precisely, $\forall n \in N \,.\, \nexists \langle n, \cdot, \cdot \rangle \in R \wedge n.\mathsf{type} = \mathsf{Service}$:

- If the Docker image of the workload modelled by n is an official Docker image for databases, n is replaced by $n' \in N'$, with $n'.\mathsf{name} = n.\mathsf{name}$ and $n.\mathsf{type} = \mathsf{Database}$;
- If the Docker image of the workload modelled by n is an official Docker image for message brokers, n is replaced by $n' \in N'$, with $n'.\mathsf{name} = n.\mathsf{name}$ and $n.\mathsf{type} = \mathsf{AsynchronousMessageBroker}$;
- In any other case, n cannot be refined and is kept also in N', viz., $n \in N'$.

As a result, there might be nodes typed as Service s, despite they are implementing some databases or message brokers by means of unofficial Docker images. If this is the case, the application developer can refine the generated μ TOSCA representation of the target MSA by suitably changing their types. To support this, our technique foresees its implementations to not only feature the fully *automated* mode described above, but also an *interactive* mode prompting developers when a sink node may implement something different from a Service. This would enable them to explicitly indicate whether such node should be typed as Service, Database, or AsynchronousMessageBroker.

Interaction Refinement. The *interaction refinement* substep is intended to characterise mined interactions by associating them with other properties than the default property d included during the *mining* step. More precisely, it associates each mined InteractsWith relationship with properties c and t, depending on whether a circuit breaker or a timeout is used during the corresponding interactions. This is done by inspecting the Istio traffic management rules defined for the service targeted by each mined relationship.

Istio traffic management rules are defined in *VirtualService*s and *DestinationRule*s (Sect. "Istio and Kiali"). *VirtualService*s allow explicitly setting a `timeout` field to indicate the maximum amount of time after which the interaction with the target service is considered to have failed (Fig. 6a). *DestinationRule*s instead feature a field `outlierDetection` that allows defining circuit breaking policies, by setting the maximum number of tolerated consecutive errors before the circuit breaker trips, as well as the amount of time it remains tripped (Fig. 6b).

The *interaction refinement* substep hence checks whether *VirtualService*s or *DestinationRule*s are defined for the target of each interaction. To avoid unnecessarily browsing the input Kubernetes manifest files, it relies on the metadata included in the Kiali graph. Kiali indeed already determines whether the target of an interaction is reached through a *VirtualService* or through a *DestinationRule* defining some circuit breaking policy. If this is the case, Kiali associates the

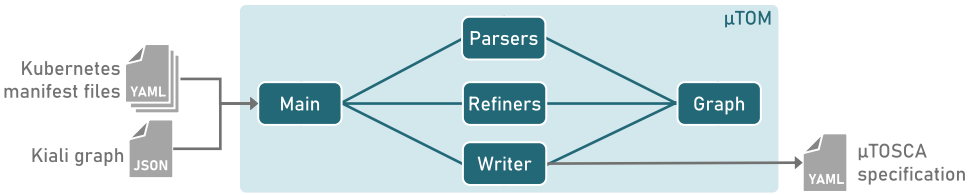


Fig. 7 Architecture of μ TOM

service targeted by a monitored interaction with properties `hasVS` or `hasCB`, respectively. Therefore, if the property `hasVS` is set for a service in the Kiali graph corresponding to the target of a mined InteractsWith relationship, the *interaction refinement* substep looks for the corresponding *VirtualService* in the Kubernetes manifest files. It then checks whether such *VirtualService* sets some `timeout` (similarly to Fig. 6a). If this is the case, the property t is set for the corresponding InteractsWith relationship, to model that a timeout is used therein.

Similarly, if the service in the Kiali graph corresponding to the target of a mined InteractsWith relationship has the property `hasCB` set, the *interaction refinement* substep looks for the corresponding *DestinationRule* in the Kubernetes manifest files. The *interaction refinement* substep then checks whether such *DestinationRule* sets a circuit breaking policy through the `outlierDetection` field (similarly to Fig. 6b). If this is the case, the property c is set for the corresponding InteractsWith relationship, to model that a circuit breaker is used therein.

More formally, the *interaction refinement* substep updates the μ TOSCA topology graph $A = \langle N, R, E \rangle$ obtained from the *node refinement* substep, by generating a new graph $A' = \langle N, R', E \rangle$. The updated graph A' refines the interactions modelled by R in A by updating their properties as described above. More precisely, $\forall \langle n, m, p \rangle \in R$:

- If there is a *VirtualService* setting a `timeout` to m, $\langle n, m, p \rangle$ is replaced by $\langle n, m, p' \rangle \in R'$, with $p \subseteq p' \wedge \{\mathsf{t}\} \in p'$;
- If there is a *DestinationRule* setting an `outlierDetection` circuit breaking policy to m, $\langle n, m, p \rangle$ is replaced by $\langle n, m, p' \rangle \in R'$, with $p \subseteq p' \wedge \{\mathsf{c}\} \in p'$;
- In any other case, $\langle n, m, p \rangle$ is preserved as is, viz., $\langle n, m, p \rangle \in R'$.

Prototype

To assess the feasibility of our mining technique, we have developed μ TOM (μ TOSCA *Offline Miner*), an open source prototype tool implemented in Java.[4] μ TOM provides a command-line interface that automatically generates the μ TOSCA specification of an MSA, given the Kubernetes manifest files specifying the corresponding application deployment and a Kiali graph obtained from an existing deployment.

μ TOM consists of the five components shown in Fig. 7. Main implements the command-line interface offered by μ TOM and coordinates the other components for enacting our two-steps mining technique. It first invokes Parsers, which resembles the Java classes implementing the logic for running the *mining* step (Sect. "Step 1: Mining") by parsing the input Kubernetes manifest files and Kiali graph. The mined MSA is represented by istantiating the object model provided by the Graph component, and returned to Main. The latter then invokes Refiners, which resembles the Java classes implementing the logic of the refinement step (Sect. "Step 2: Refinement"). This results in updating the instance of the Graph object model, which is refined by updating the types associated with mined nodes and by characterizing mined relationships. The refined instance is returned to Main, which passes it to Writer. The latter implements the logic for processing the obtained instance of the Graph object model and marshalling it to a μ TOSCA specification in YAML, which constitutes the output of μ TOM.

μ TOM can be run by issuing the command:

```
java -jar microTOM-1.0.jar WORKDIR [-i]
```

where "`microTOM-1.0.jar`" is the executable file JAR file obtained from μ TOM 's sources. "`WORKDIR`" is instead the path to a directory containing the Kubernetes manifest files and the Kiali graph to be passed as input to μ TOM, and where μ TOM will also store the generated μ TOSCA file. Finally, the option "`-i`" enables activating the *interactive* refinement mode foreseen in Sect. ("Step 2: Refinement"): when "`-i`" is set, the user is prompted with the nodes that remain assigned with type Service even if their interactions are such that they may implement some

[4] The sources of μ TOM are publicly available on GitHub (https://github.com/di-unipi-socc/microTOM) and on Software Heritage (https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/di-unipi-socc/microTOM).

different component, and she is asked to confirm or update their type. By default, μ TOM however runs the fully *automated* mode.

Case Studies

To assess our approach, we exploited μ TOM to mine the MSA of four open source, third-party applications, namely *Sock Shop* [10], *Online Boutique* [11], *Robot Shop* [12], and *Book Info* [13]. We actually compared the MSA mined by μ TOM (in its fully *automated* mode) with that declared in the online available documentation of the considered applications, as well as with that mined by μ Miner [5], the state-of-the-art tool for mining the μ TOSCA specification of an MSA [4]. As a result, we observed that μ TOM effectively mined the MSA of the considered applications (as per what declared in their documentation), and that it generated more informative μ TOSCA specifications if compared with μ Miner. For instance, μ TOM identified the use of timeouts and circuit breakers in mined interactions, which were not instead detected by μ Miner.

We hereafter report on the mining of the MSAs of *Sock Shop* (Sect. "Sock Shop"), *Online Boutique* (Sect. "Online Boutique"), *Robot Shop* (Sect. "Robot Shop"), and *Book Info* (Sect. "Book Info"). To enable repeating our assessment, we published a dump of all the necessary inputs online.[5]

Sock Shop

Sock Shop [10] is a microservice-based application simulating an e-commerce website selling socks. According to its documentation, the MSA of *Sock Shop* is that shown in Fig. 8a, and the application does not include failure handling mechanisms like timeouts or circuit breakers [10].

We run both μ Miner and μ TOM on the publicly available Kubernetes deployment of *Sock Shop* to automatically generate a μ TOSCA representation of the MSA of *Sock Shop*. In both cases, we exploited the load script available on *Sock Shop*'s GitHub repository[6] to load the application. More precisely, we used the load script to load the deployed *Sock Shop* instance during the dynamic mining step of μ Miner [14]. When using μ TOM, instead, we deployed an instance of the application and used the load script to generate workload. We then downloaded the Kiali graph obtained from the loaded deployment, which we then provided as input to μ TOM.

Figure 8b illustrates the μ TOSCA representation of the MSA of *Sock Shop* generated by μ Miner. By looking at the figure, we can observe that μ Miner successfully recognised all *Sock Shop*'s components, all the interactions occurring among them, and that front-end is the only service accessible by external clients. μ Miner also successfully typed the message routing components as MessageRouter s, rabbitmq as a AsynchronousMessageBroker, and carts-db and order-db as Database s. μ Miner instead wrongly typed user-db and catalogue-db, which are represented as Service s despite they actually are databases. The reason for this is that μ Miner types a component as a Database if it runs from an official Docker image for databases, which is not the case for user-db and catalogue-db.

The same limitation is shared by the newly proposed μ TOM, as μ TOM also types a component as a Database if it runs from an official Docker image for databases (Sect. "Step 2: Refinement"). This can be observed by looking at the μ TOSCA representation of the MSA of *Sock Shop* mined by μ TOM, which is shown in Fig. 8c, and where user-db and catalogue-db are wrongly typed as Service s. As one can also observe, the μ TOSCA representation generated by μ TOM (Fig. 8c) is the same as that generated by μ Miner (Fig. 8b), meaning that μ TOM is as good as μ Miner in mining the MSA of *Sock Shop*. The advantage of μ TOM with respect to μ Miner is however that μ TOM did not require us to run *Sock Shop* in a suitably configured testing environment, as we were instead able to run it as is in a Kubernetes cluster.

Online Boutique

Online Boutique is a demo microservice-based application developed by Google [11]. It provides another example of e-commerce application, still without timeouts or circuit breakers handling possible service failures. According to its online available documentation [11], the MSA of *Online Boutique* is that in Fig. 9. It is worth noting that, being it a demo application, it natively includes a loadgenerator component generating workload to the application frontend when the application is deployed.

We run both μ Miner and μ TOM on the publicly available Kubernetes deployment of *Online Boutique* to automatically generate a μ TOSCA representation of its MSA. Fig. 9b and c provide the μ TOSCA representations of the MSA mined by μ Miner and μ TOM, respectively. Both tools successfully elicited all the components forming *Online Boutique* and all interactions occurring among them. They also successfully typed all components, therein included recognizing that cache is a Database.

[5] The inputs for repeating the assessment described in this section are publicly available on GitHub (https://github.com/di-unipi-socc/microTOM/tree/main/data/examples) and on Software Heritage (https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/di-unipi-socc/microTOM &path=data/examples).

[6] https://github.com/microservices-demo/load-test.

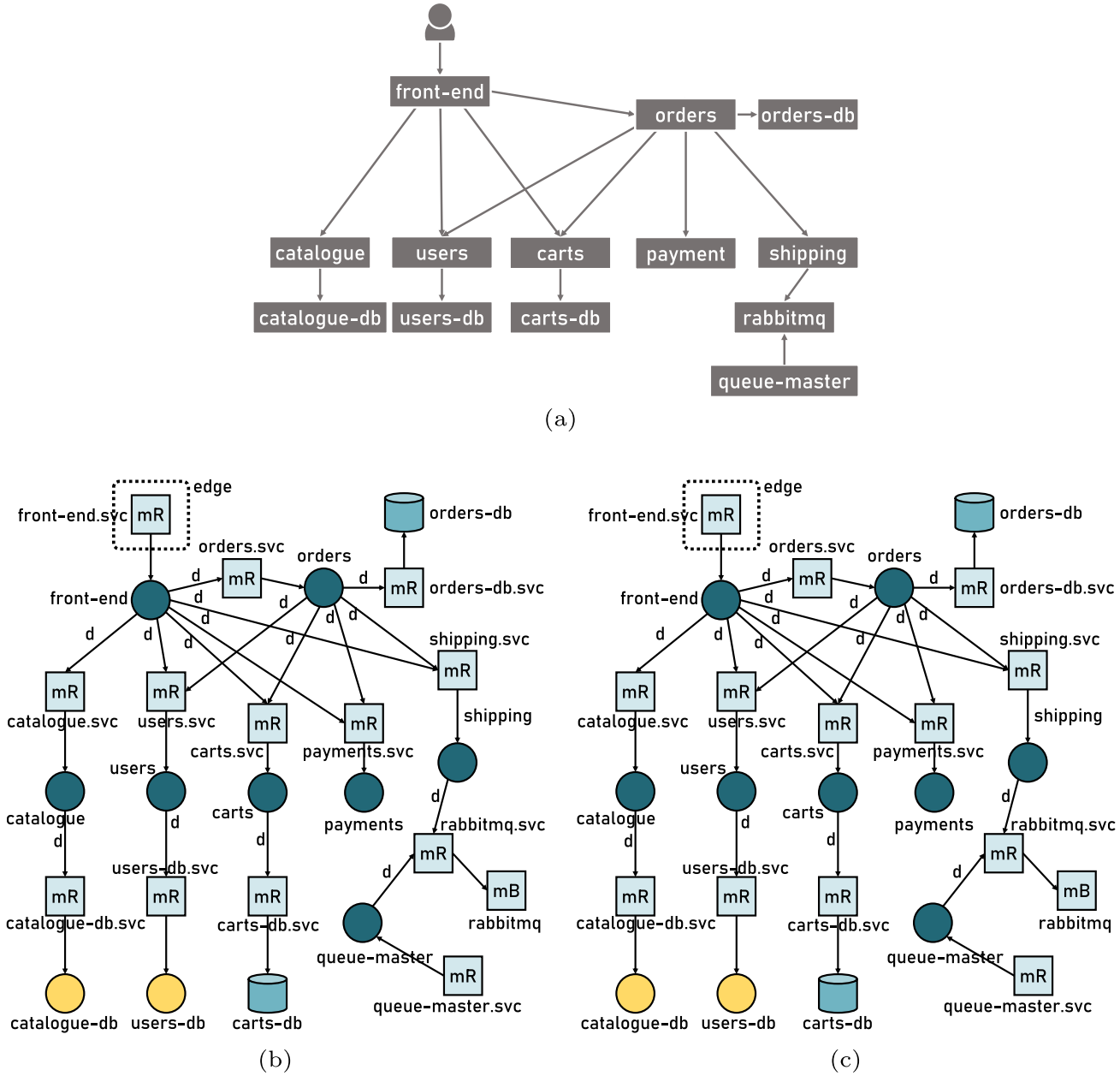


Fig. 8 MSAs of *Sock Shop* **a** taken from its documentation and mined with **b** μ Miner and **c** μ TOM. Issues in the mined MSAs are highlighted in yellow

The only difference between the mined MSAs resides in the message routing components handling the requests arriving to frontend. μ Miner "splits" the Kubernetes service used to handle the requests sent to frontend into two message routing components, viz., frontend-default.svc handling the internal traffic sent by the loadgenerator and frontend-external.svc handling that potentially arriving from external clients. μ TOM instead recognises that such Kubernetes service is actually one message routing component, handling the requests arriving by both the loadgenerator and external clients.

In summary, μ TOM was as good as μ Miner in recognizing the components and interactions forming *Online Boutique*. μ TOM actually outperformed μ Miner by avoiding to split the Kubernetes service handling the traffic arriving to frontend into multiple message routing components.

Robot Shop

Robot Shop [12] is a microservice-based application simulating an e-commerce website selling robots. Similarly to the two formerly considered applications, *Robot Shop* does not

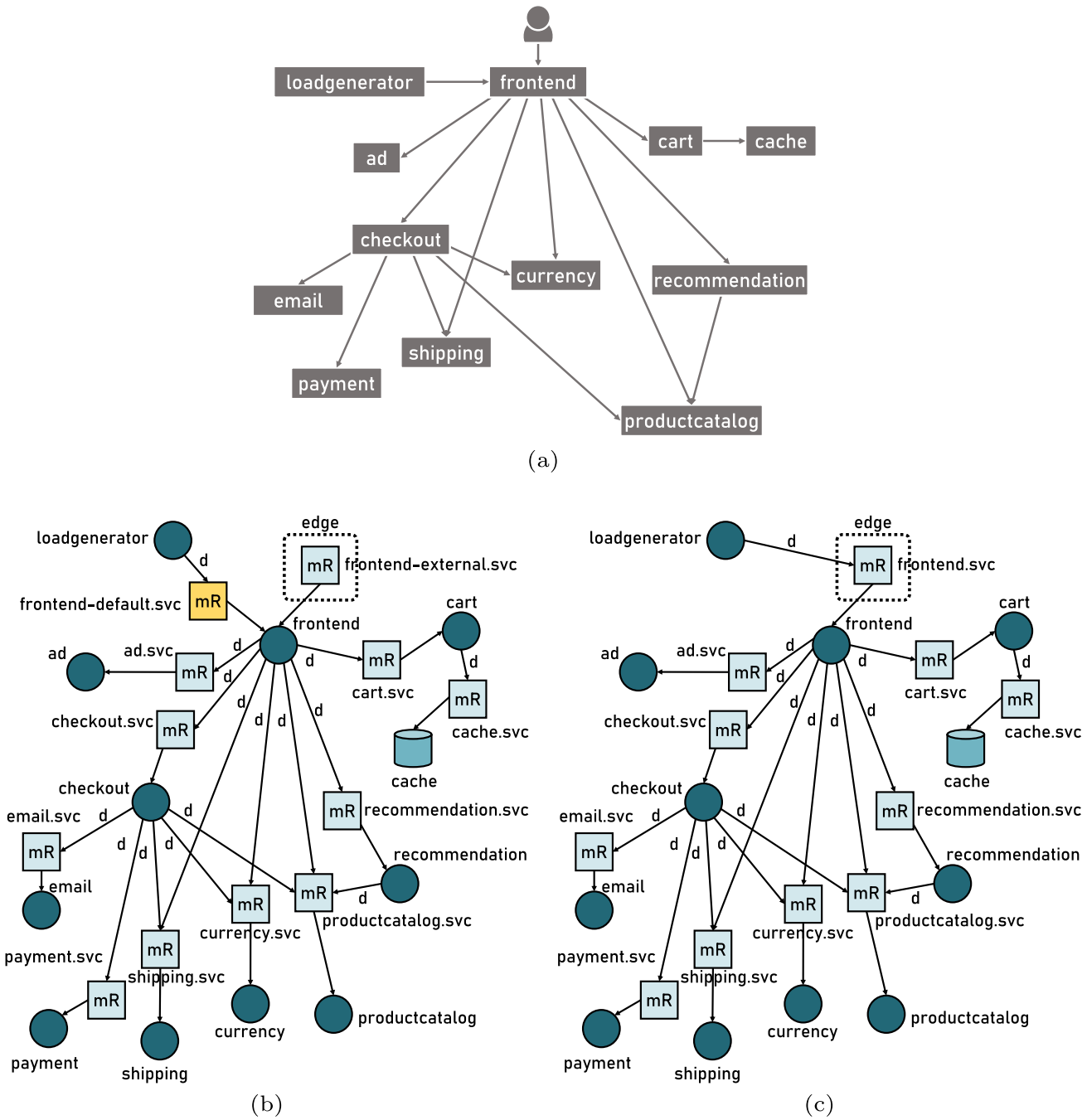


Fig. 9 MSAs of *Online Boutique* **a** taken from its documentation and mined with **b** μ Miner and **c** μ TOM. Issues in the MSA mined by μ Miner are highlighted in yellow

include failure handling mechanisms like timeouts or circuit breakers. Its MSA is documented to be as shown in Fig. 10a.

We run both μ Miner and μ TOM on the publicly available Kubernetes deployment of *Robot Shop* to automatically generate a μ TOSCA representation of the MSA of *Robot Shop*. In both cases we exploited the *Robot Shop*'s load component to generate workload for the application and monitor the runtime interactions among its components. In the case of μ Miner, we used the load directly in its dynamic mining step [14]. In the case of μ TOM, we instead deployed the application, used the load component to generate workload, and then downloaded the Kiali graph from the running deployment, which we then provided as input to μ TOM itself. The μ TOSCA representations generated by μ Miner and μ TOM are shown in Fig. 10b and c, respectively.

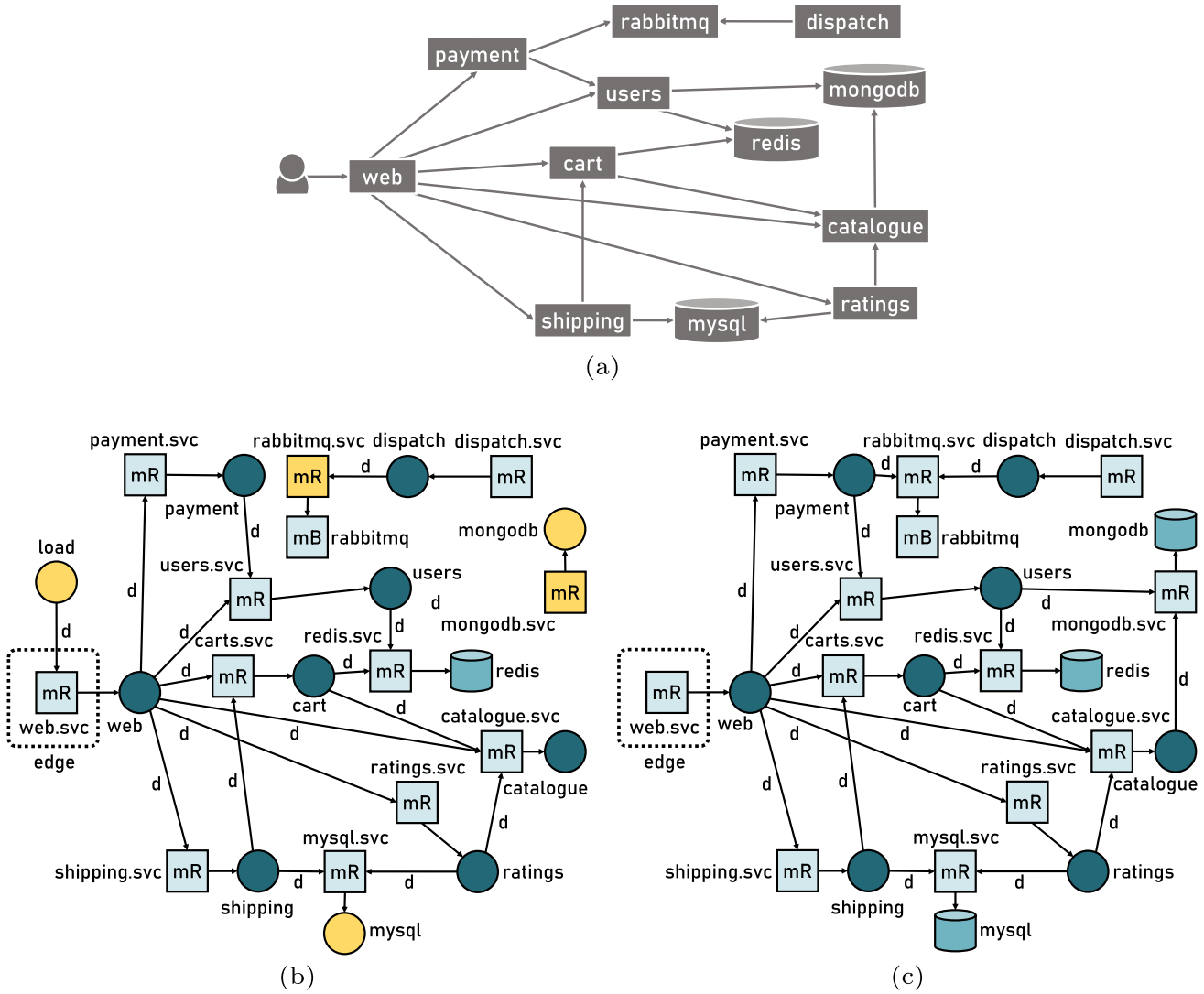


Fig. 10 MSAs of *Robot Shop* **a** taken from its documentation and mined with **b** μ Miner and **c** μ TOM. Issues in the MSA mined by μ Miner are highlighted in yellow

By looking at Fig. 10, we can observe that both μ Miner and μ TOM successfully identified all components forming the MSA of *Robot Shop*. Given that they are deployed as Kubernetes workloads managed by Kubernetes services, each mined node is proxied by a MessageRouter node implementing the corresponding Kubernetes service. At the same time, we can observe that there are some issues in the nodes mined by μ Miner, viz., (i) mongodb and mysql are not recognised to be Database s, but rather typed as Service s, and (ii) the load component used to generated workload is included in the mined MSA, even if it is not truly part of the MSA of *Robot Shop*. The same does not hold for μ TOM, which successfully identifies mongodb and mysql as Database s, and which does not include the load component in the mined MSA.

In addition, while both μ Miner and μ TOM effectively characterise the mined InteractsWith relationships, two relationships are missing in the MSA mined by μ Miner. The latter does not include the interactions from payment and catalog to the Kubernetes services managing rabbitmq and mongodb, respectively. As a result, the portions including rabbitmq and mongodb result to be disconnected from the rest of the MSA in the topology mined by μ Miner. The same does not hold for the μ TOSCA topology mined by μ TOM, which successfully identifies all the interactions in the MSA of *Robot Shop*.

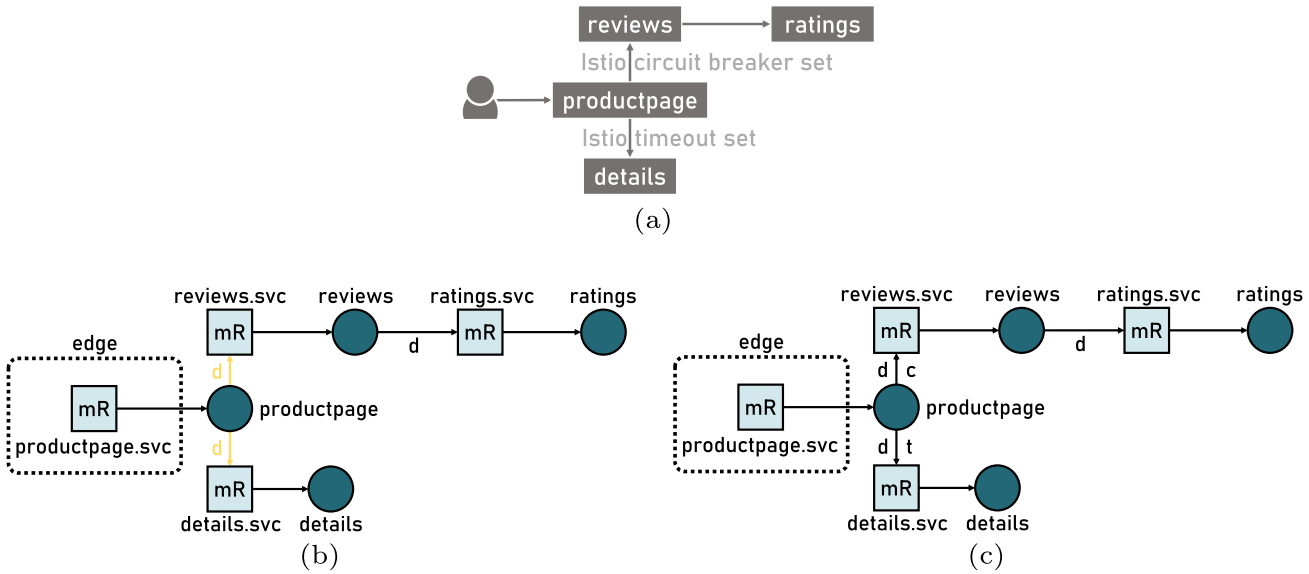
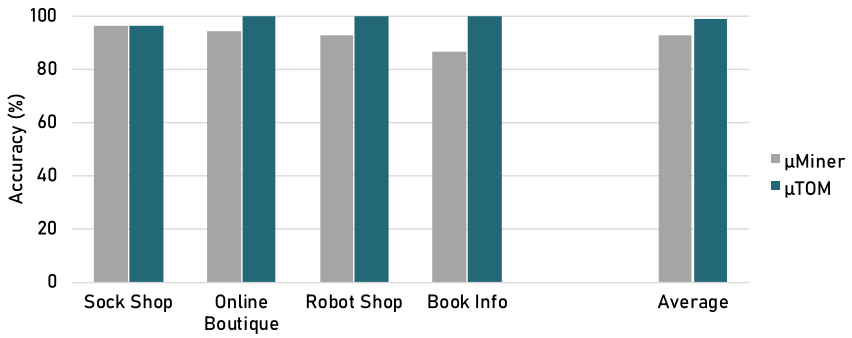


Fig. 11 MSAs of *Book Info* **a** taken from its documentation and mined with **b** μ Miner and **c** μ TOM. Issues in the MSA mined by μ Miner are highlighted in yellow

Fig. 12 **a** Table and **b** bar plot of the accuracy of μ Miner and μ TOM in identifying and characterising the nodes and interactions in the applications considered in our case studies. In the table, the highest accuracy in each column is bolded

	Sock Shop	*Online Boutique*	*Robot Shop*	*Book Info*	Average
μMiner	**96.36**	94.55	92.86	86.67	92.61
μTOM	**96.36**	**100.00**	**100.00**	**100.00**	**99.09**

(a)



(b)

Book Info

Book Info [13] is a microservice-based application developed to play with Istio. It consists of the four services in Fig. 11a. We instrumented its Kubernetes deployment by exploiting Istio to set a timeout in the interactions between productpage and details, and a circuit breaker in that between productpage and reviews. This enabled us to show that μ TOM outperforms μ Miner in determining whether timeouts or circuit breakers are used in interactions.

The above can be readily observed by looking at the μ TOSCA representations of the MSA of *Book Info* generated by μ Miner and μ TOM, which are shown in Fig. 11b and c, respectively. Whilst both μ Miner and μ TOM effectively mined all components and interactions forming *Book Info*, only μ TOM successfully detected the timeout and circuit breaker used in the InteractsWith relationships outgoing from productpage.

Discussion

Figure 12 shows the accuracy of μ Miner and μ TOM in mining the MSAs of the applications considered in our case studies. The accuracy is measured using the classical formula:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Table 2 Inputs, requirements, and type of run for μ Miner and μ TOM

	Inputs	Requirements	Run
μ Miner	Kubernetes manifest files Kubernetes cluster load generator*	No traffic encryption root privileges on cluster	Online
μ TOM	Kubernetes manifest files Kiali graph	Istio and Kiali enabled	Offline

*If not available

where:

- Successfully recognised components and interactions are classified as *true positives* (*TP*),
- Wrongly typed components and wrongly characterised interactions are classified as *false positives* (*FP*), and
- Missing components/interactions are classified as *false negatives* (*FN*).

It is worth noting that, as per the above definition, *false positives* (*FP*) also include those components/interactions that were mined even if they were not present in the original MSA of a considered application. Also, counting *true negatives* (*TN*) is not meaningful for the task of mining the MSA of an existing application, as they would correspond in correctly identifying the components that are *not* part of the target MSA. *TN* was hence set to zero when computing the accuracy of μ TOM and μ Miner in each of the considered cases.

The numbers in Fig. 12 show that μ TOM achieved 100% accuracy when mining the MSAs of *Online Boutique*, *Robot Shop*, and *Book Info*, as it successfully identified and characterised all the components/interactions forming their MSAs. It hence outperformed μ Miner in mining the MSAs of such applications: the accuracy of μ Miner when mining their MSAs was indeed significantly below 100%, especially in the cases of *Robot Shop* and *Book Info*. The same did not hold for the case of *Sock Shop*, where μ TOM and μ Miner achieved the same accuracy, as they both did not recognise that two of the mined components were Database s (Sect. "Sock Shop"). However, the numbers in the table, also considering the average values of accuracy (rightmost column), show that μ TOM outperformed μ Miner in the four considered case studies.

Other than effectiveness, it is also worth commenting on what we did to let μ Miner and μ TOM mine the MSAs of the applications considered in our assessment. In both cases, we set up a Kubernetes cluster where to deploy the considered applications and used load generators to simulate end user requests, by devising ad-hoc scripts for the applications coming without a load generator, viz., *Sock Shop* and *Book Info*. The application deployments were however configured differently. In the case of μ Miner, we configured the security of the cluster-based deployments ad-hoc, to ensure that no traffic encryption was enforced, and that we could run μ Miner on the master node of the Kubernetes cluster with root privileges. In the case of μ TOM, we instead only enabled Istio and Kiali on the clusters where the considered applications were deployed. While the Kubernetes cluster setup and load generation are mandatory in general for μ Miner (Table 2), they were required by μ TOM only since we were considering third-party applications, whose existing deployments were not accessible for us. The above considerations are only giving first insights on the usability and applicability of μ TOM and μ Miner: a thorough empirical evaluation is however needed, and planned as part of our future work.

Related Work

Several solutions have been proposed for mining the MSAs of existing applications. The closest to μ TOM is μ Miner, which we presented in our previous work [14]. μ Miner is indeed the only existing solution mining a μ TOSCA representation of an application's MSA from its Kubernetes deployment. μ Miner runs the application deployment on a devoted cluster, and it sniffs the packets exchanged among the deployed application components to mine services and service interactions. The enacted sniffing requires μ Miner to run with root privileges on the cluster, and the application to not encrypt any of the messages exchanged among deployed components, which can of course happen only in a testing environment. The deployed application must also be loaded to stress all possible service interactions, to allow μ Miner to monitor them. In short, μ Miner requires to run the target microservice-based application in a suitably configured testing environment, which limits its applicability, e.g., not allowing it to consider an existing deployment of the application, like its production deployment. The same does not hold for our technique, which works with existing Kubernetes application deployments, including production deployments. In addition, as we have shown in Sect. ("Case Studies"), our technique outperforms μ Miner in the quality of mined MSAs.

Other approaches worth mentioning are those presented in [17–21]. They introduce different techniques, which differ from ours in the mining approach and in the generated representation of mined MSAs. As for the latter, we generate a representation of the mined MSA where components

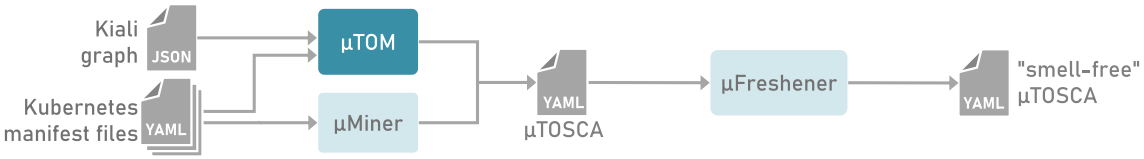


Fig. 13 Updated μ TOSCA toolchain. Existing tools are in light blue, while the newly introduced tool is darker

are distinguished among services, integration components, and databases. This is intended to enable checking whether the mined MSA is affected by some architectural smells, by giving the mined MSA to smell detection tools like μ Freshener [5]. The same is not supported by the approaches presented in [17–21], which do not distinguish the type of mined components.

As for the enacted mining approach, the approaches presented in [17–19] reconstruct the MSA of an application by statically analysing the source code of its components. They hence follow a "white-box" approach, assuming the availability of the source code of the components forming an MSA. Our mining technique instead works also in "black-box" scenarios, viz., when the source code of application components is not available. We indeed only require the manifest files specifying its deployment in Kubernetes and the runtime interactions monitored among its components. In addition, while our mining solution can be fully automated, those presented in [18, 19] require developers to manually intervene while mining an MSA.

Similar considerations apply to the approaches presented in [20, 21], which also employ a white-box, semi-automated technique to mine an MSA from the source code of its components. Such technique is semi-automated since it relies on developers to manually refine the obtained MSA by removing the infrastructure facilities (e.g., service discovery components) used to let application components interoperate. At the same time, the approaches presented in [20, 21] are a step closer to ours, given that they enrich the mined MSA by relying on runtime monitored interactions. Our technique hence differs from what proposed in [20, 21], since it can fully automate the mining of an MSA, and since it can work in black-box scenarios, i.e., when the source code of some application components is not available.

Finally, it is worth relating our mining technique with existing systems for monitoring Kubernetes-based application deployments. For instance, Kiali [9], KubeView [22], and WeaveScope [23] are three open source tools for monitoring and visualising the structure of applications deployed with Kubernetes. They differ from our technique mainly since their goal is to enable visualising the deployed Kubernetes objects (e.g., workloads and services) and their interactions. Our solution instead generates a machine-readable representation of an MSA, whose components are distinguished among services, integration components, and databases forming an MSA, and where component interactions are characterised by indicating whether client-side service discovery, timeouts, or circuit breakers are used therein.

Similar considerations apply to Instana [24], another tool for visualising applications deployed with Kubernetes. Instana [24] is however closer to our mining technique in the generated representation, given that it distinguishes the deployed components among services and databases. Our mining technique goes beyond this, by recognising whether deployed components are implementing message routing/brokering patterns, and whether service discovery, timeouts, or circuit breakers are used in component interactions. Additionally, while Instana [24] is a commercial and subscription-based tool, an open source implementation of our technique is publicly available on GitHub.

Conclusions

We have presented a technique for mining MSAs from their Kubernetes deployment. Our technique also inputs the component interactions monitored in a former deployment with Kiali, and it process all such inputs offline. As a result, it automatically generates a representation of the mined MSA in μ TOSCA, a microservice-oriented profile of the TOSCA standard.

We have also presented μ TOM, a prototype implementation of our mining technique. μ TOM plugs into the μ TOSCA toolchain [4], as shown in Fig. 13. It actually provides an offline alternative to μ Miner [14] to generate μ TOSCA representations of MSAs, which can still be processed by μ Freshener [7] to identify and resolve the architectural smells therein. μ TOM showed to outperform μ Miner in generating more informative representations of mined MSAs, without requiring to run the target application in a suitably configured testing environment, but rather by processing the information monitored with Kubernetes-native monitoring in former application deployments, e.g., production deployments. If such information is not available, e.g., since Kubernetes-native monitoring is not enabled, one could anyhow still use μ Miner to mine the MSA of an application.

We anyhow plan to further enhance the mining capabilities of μ TOM and, more generally, of our mining technique. For instance, we plan to enhance the detection of the type of mined components, which currently detects message brokers or databases when they run from official Docker images of software distributions known to implement such components. The type of component run by a Docker image may be detected by exploiting machine learning techniques, e.g., similary to what done in [25] to predict the popularity of Docker images, or by inspecting them with approaches like that proposed in DockerFinder [26].

We also plan to enable μ TOSCA to model security aspects of MSAs and our mining technique to elicit them from their Kubernetes deployment, e.g., whether service interactions are encrypted, whether microservices are directly accessible by external clients, or which access rights are given to microservices. This would enable analysing mined MSAs to also, e.g., identify and resolve microservices' security smells [27].

Finally, we plan to enable our technique to work with other technologies than Kubernetes, Istio, and Kiali. For instance, we plan to include support for manifest files specifying the deployment of a microservice-based application with Docker Compose/Swarm. We also plan to support processing the interactions monitored with other tracing tools, e.g., Jaeger [28] or Zipkin [29].

Funding Open access funding provided by Università di Pisa within the CRUI-CARE Agreement. The authors have no relevant financial or non-financial interests to disclose.

Data availability No data was used for the research described in the article.

Declarations

Conflict of Interest The authors also do not have any competing/conflict of interests relevant to content presented in this article.

References

1. Kratzke N, Quint P-C. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. J Syst Softw. 2017;126:1–16. https://doi.org/10.1016/j.jss.2017.01.001.
2. Soldani J, Tamburri DA, Van Den Heuvel W-J. The pains and gains of microservices: a systematic grey literature review. J Syst Softw. 2018;146:215–32. https://doi.org/10.1016/j.jss.2018.09.082.
3. Zimmermann O. Microservices tenets. Computer Sci: Res Dev. 2017;32(3–4):301–10. https://doi.org/10.1007/s00450-016-0337-0.
4. Soldani J, Muntoni G, Neri D, Brogi A. The μTOSCA toolchain: mining, analyzing, and refactoring microservice-based architectures. Softw: Pract Exp. 2021;51(7):1591–621. https://doi.org/10.1002/spe.2974.
5. Brogi A, Neri D, Soldani J. Freshening the air in microservices: Resolving architectural smells via refactoring. In: Yangui, S., (eds.) Service-Oriented Computing–ICSOC 2019 Workshops, 2020;pp. 17–29. Springer, Cham. https://doi.org/10.1007/978-3-030-45989-5_2
6. OASIS: TOSCA Simple Profile in YAML. v1.3, https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.pdf 2020.
7. Neri D, Soldani J, Zimmermann O, Brogi A. Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Softw-Intensive Cyber-Phys Syst. 2020;35(1):3–15. https://doi.org/10.1007/s00450-019-00407-8.
8. The Istio Authors: Istio. https://istio.io 2022.
9. The Kiali Authors: Kiali. https://kiali.io 2022.
10. Weaveworks, Container Solutions: Sock Shop. https://microservices-demo.github.io 2021.
11. Google Cloud: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo 2021.
12. Instana: Robot Shop. https://github.com/instana/robot-shop 2021.
13. The Istio Authors: Book Info. https://github.com/istio/istio/tree/master/samples/bookinfo 2021.
14. Muntoni G, Soldani J, Brogi A. Mining the architecture of microservice-based applications from their kubernetes deployment. In: Zirpins, C., (eds.) Advances in Service-Oriented and Cloud Computing, pp. 103–115. Springer, Cham 2021. https://doi.org/10.1007/978-3-030-71906-7_9
15. Soldani J, Khalili J, Brogi A. Offline mining of microservice-based architectures. In: Proceedings of the 12th International Conference on Cloud Computing and Services Science–CLOSER,, pp. 63–73. SciTePress, Setúbal, Portugal 2022. https://doi.org/10.5220/0011061000003200. INSTICC.
16. Hohpe G, Woolf B. Enterprise integration patterns: designing, building, and deploying messaging solutions. USA: Addison-Wesley; 2003.
17. Ma S, Fan C, Chuang Y, Lee W, Lee S, Hsueh N. Using service dependency graph to analyze and test microservices. In: Reisman, S., (eds.) 2018 IEEE 42nd Annual Computer Software and Applications Conference, pp. 81–86 2018. https://doi.org/10.1109/COMPSAC.2018.10207
18. Rademacher F, Sachweh S, Zündorf A. A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan, S., (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 311–326. Springer, Cham 2020. https://doi.org/10.1007/978-3-030-49418-6_21
19. Alshuqayran N, Ali N, Evans R. Towards micro service architecture recovery: An empirical study. In: Gorton, I., (eds.) 2018 IEEE International Conference on Software Architecture, 2018; pp. 47–4709. https://doi.org/10.1109/ICSA.2018.00014
20. Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. Towards recovering the software architecture of microservice-based systems. In: Malavolta, I., Capilla, R. (eds.) 2017 IEEE International Conference on Software Architecture Workshops, 2017;pp. 46–53 . https://doi.org/10.1109/ICSAW.2017.48

21. 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. In: Malavolta, I., Capilla, R. (eds.) 2017 IEEE International Conference on Software Architecture Workshops, 2017;pp. 298–302. https://doi.org/10.1109/ICSAW.2017.9
22. Coleman B. KubeView. https://github.com/benc-uk/kubeview 2021.
23. Weaveworks: WeaveScope. https://www.weave.works/oss/scope 2021.
24. Instana: Instana. https://www.instana.com 2021.
25. Guidotti R, Soldani J, Neri D, Brogi A, Pedreschi D. Helping your Docker images to spread based on explainable models. In: Brefeld, U., (eds.) Machine Learning and Knowledge Discovery in Databases, pp. 205–221. Springer, Cham 2019. https://doi.org/10.1007/978-3-030-10997-4_13
26. Brogi A, Neri D, Soldani J. Dockerfinder: Multi-attribute search of docker images. In: 2017 IEEE International Conference on Cloud Engineering (IC2E), 2017; pp. 273–278. https://doi.org/10.1109/IC2E.2017.41
27. Ponce F, Soldani J, Astudillo H, Brogi A. Smells and refactorings for microservices security: a multivocal literature review. J Syst Softw. 2022;192: 111393. https://doi.org/10.1016/j.jss.2022.111393.
28. The Jaeger Authors: Jaeger. https://www.jaegertracing.io 2021.
29. OpenZipkin: Zipkin. https://zipkin.io 2021.