Semi-Automated Smell Resolution in Kubernetes-Deployed Microservices

Jacopo Soldani[a], Marco Marinò and Antonio Brogi[b]
Department of Computer Science, University of Pisa, Pisa, Italy

Keywords: Microservices, Kubernetes, Architectural Smells, Architectural Refactoring.

Abstract: Microservices are getting commonplace, as their design principles enable obtaining cloud-native applications. Ensuring that applications adheres to microservices' design principles is hence crucial, and this includes resolving architectural smells possibly denoting violations of such principles. To this end, in this paper we propose a semi-automated methodology for resolving architectural smells in microservices applications deployed with Kubernetes. Our methodology indeed automatically detects architectural smells by analyzing the Kubernetes manifest files specifying an application's deployment, and it can also generate the refactoring templates for resolving such smells. We also introduce `KubeFreshener`, an open-source prototype of our methodology, which we use to assess it in practice based on a controlled experiment and a case study.

1 INTRODUCTION

Microservices gained momentum in enterprise IT, as they enable realizing so-called *cloud-native* applications (Balalaie et al., 2018; Yussupov et al., 2020). Microservices are essentially service-oriented architectures satisfying additional key design principles, e.g., ensuring microservices' independent deployability and horizontal scalability, or isolating failures (Zimmermann, 2017). Ensuring that microservices' design principles are satisfied is hence crucial for a microservices application to truly deliver their promises (Taibi and Lenarduzzi, 2018; Herrera et al., 2023).

(Neri et al., 2020) singled out architectural smells, which could possibly denote violations of microservices' key design principles. An architectural smell is the observable symptom of a bad (though unintentional) decision while designing an application, which may negatively impact on its adherence to design principles (Garcia et al., 2009), and which can be resolved through refactoring. For this reason, (Neri et al., 2020) also elicited the architectural refactorings that enable resolving the occurrence of such smells.

In this paper, we propose a "black-box" methodology to resolve the occurrence of four architectural smells from (Neri et al., 2020) in microservices applications. Our methodology is "black-box" in the sense that it abstracts from the internals of containerized microservices, which are typically polyglot (Soldani et al., 2018). We rather automatically detect architectural smells in microservices applications by analyzing the manifest files specifying their deployment in Kubernetes, the de-facto standard for microservices deployment (Indrasiri and Siriwardena, 2018). We also show how to generate templates for resolving the detected architectural smells, by refactoring the manifest files specifying their deployment. The resolution templates specify the refactorings needed for resolving the detected smells, which should be completed by suitably adapting an application's microservices and their deployment to ensure that the application's functionalities are preserved. For this reason, we say that our methodology enables *semi-automatically* resolving architectural smells in microservices applications.

To assess the practical applicability of our methodology, we introduce `KubeFreshener`, an open-source prototypical implementation that we use to run experiments. We indeed discuss the application of `KubeFreshener` in a controlled experiment based on a toy application deployment, showing that it can effectively detect injected architectural smells and generate their resolution templates. We also illustrate a case study applying `KubeFreshener` to an existing, third-party application, in which we detect architectural smells and generate the templates of the refactorings allowing to resolve them. In the case study, we also discuss how an existing smell can lead to a possible violation of a microservices' key design principle, while the same does not hold in the refactored application deployment (even if some further adaptation would be needed to

[a] https://orcid.org/0000-0002-2435-3543
[b] https://orcid.org/0000-0003-2048-2468

Soldani, J., Marinò, M. and Brogi, A.
Semi-Automated Smell Resolution in Kubernetes-Deployed Microservices.
DOI: 10.5220/0011845500003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 34-45
ISBN: 978-989-758-650-7; ISSN: 2184-5042

suitably complete the refactoring).

In summary, the main contributions of this paper are the following:

(i) We propose a semi-automated methodology to resolve smells in microservices applications deployed with Kubernetes,

(ii) we introduce `KubeFreshener`, a prototypical implementation of the proposed methodology, and

(iii) we evaluate our methodology by exploiting `KubeFreshener` to run a controlled experiment and a case study.

The paper is organized as follows. Section 2 provides background on microservices' architectural smells and refactorings. Section 3 introduces our semi-automated smell resolution methodology, whose prototype and evaluation are presented in Sections 4 and 5, respectively. Finally, Sections 6 and 7 discuss related work and draw some concluding remarks, respectively.

2 BACKGROUND

We hereafter recap four microservices' architectural smells from (Neri et al., 2020), together with the refactoring allowing to resolve their occurrence.

Multiple Services per Deployment Unit. (Neri et al., 2020) consider containers (such as Docker containers) as the deployment units for microservices, and explain how placing multiple microservices in the same container would constitute a microservices' architectural smell. One such placement would indeed violate the principle of *independent deployability* of microservices: if two microservices would be shipped within the same container, spawning/destroying such container would necessarily result in deploying/undeploying both microservices at the same time. More generally, by placing two microservices in the same deployment unit (whether this being a Docker container or a Kubernetes pod), such microservices would operationally depend one another, as it would not be possible to launch a new instance of one of the two, without also launching an instance of the other.[1]

The *multiple services per deployment unit* smell can be resolved by refactoring the deployment so that each microservice is deployed with a different deployment unit. This means that each microservice should be packaged in a different container, and that it should run in a different pod, if using Kubernetes.

[1]We here generalize the smell called *multiple services in one container* in (Neri et al., 2020) to deployment units, as pods (and not containers) are Kubernetes' deployment units.

Endpoint-based Service Interaction. This architectural smell occurs when a microservice invokes a specific instance of another microservice, e.g., since no load balancer is used to handle the requests arriving to the instances of the invoked microservice. If this is the case, the *horizontal scalability* of the invoked microservice is compromised: when scaling out such microservice by adding new replicas, the newly introduced replicas would not be reached by the invokers, hence only wasting resources (Neri et al., 2020).

An *endpoint-based service interaction* smell can be resolved by introducing a message router, which handles the requests sent to a microservice, e.g., by balancing the load among its replicated instances.

No API Gateway. When an application lacks an API gateway, external clients must necessarily invoke its microservices directly. The result is a situation similar to that of *endpoint-based service interaction*s, with invokers this time being the external clients (rather than other microservices from the same application). If this is the case, the *horizontal scalability* of invoked microservices is compromised, similarly to what happens for the case of *endpoint-based service interaction*s.

A *no API gateway* smell can be resolved by introducing a message router working as API gateway for the affected microservice. The introduced gateway handles the requests of external clients by suitably routing them to the affected microservice's instances.

Wobbly Service Interaction. The interaction between two microservices is "wobbly" when a failure of the invoked microservice can trigger a cascading failure in the invoker, hence compromising the principle of *isolation of failures* of microservices. This typically happens when a microservice invokes the functionalities offered by another microservice, without any solution for handling the possibility of the invoked service to fail or be unresponsive.

A *wobbly service interaction* can be resolved by introducing timeouts or circuit breakers to handle the failure of invoked microservices. Timeouts are a simple yet effective solution for a microservice to stop waiting for an answer from another microservice, when the latter is unresponsive, e.g., since it failed or due to network issues. Circuit breakers instead wrap the invocations from a microservice to another. The circuit breaker is initially "closed", meaning that it just forwards the invocations to the wrapped microservice, and monitors its execution to detect and count failing invocations. Once the frequency of failures reaches a given threshold, the circuit breaker trips and "opens" the circuit. All further calls to the wrapped microservice will "safely fail", as the circuit breaker will immediately return an error message to the callers.

3 METHODOLOGY

The overall idea is to process the manifest files specifying the deployment of a microservices application in Kubernetes, so as to detect occurrences of architectural smells and generate their resolution templates. We hereafter illustrate how to do it for the architectural smells from (Neri et al., 2020) recapped in Section 2, separately. For each architectural smell, we actually first recap its *related Kubernetes objects*, to then illustrate how to exploit such objects to enact *smell detection* and *resolution template generation*.

3.1 Multiple Services per Deployment Unit

The *multiple services per deployment unit* happens when different microservices are shipped within the same deployment unit (Section 2). If this happens, such microservices operationally depend one another, and it would not be possible to launch a new instance of one of them, without also launching an instance of the others (Neri et al., 2020).

3.1.1 Related Kubernetes Objects

Pods constitute the smallest deployment units that can be spawned and managed by Kubernetes (Kubernetes, 2022). They can be created directly with manifests of type `Pod`, or – more typically – from workload resources specified in manifests of type `Deployment` (Poulton, 2022).

Essentially, a pod is a group of one or more containers, with a specification for how to run them (Figure 1). When running multiple containers, they are always co-located and co-scheduled, and run in a shared context, e.g., with shared storage and network resources. For this reasons, and following the guidelines given in the documentation of Kubernetes (Kubernetes, 2022), co-located containers should be used for initialization purposes, through so-called *init containers*, or when they constitute a single cohesive unit of service, with one "main container" running the service and a set of supporting containers implementing the *sidecar*, *ambassador*, or *adapter* patterns (Richardson, 2018).

Figure 1 provides an example of `Deployment`, for a service named `catalogue`, which runs from the main container named `catalogue`. The figure also includes an init container for loading the catalogue itself, and an example of supporting container, viz., a sidecar `dynatrace` container deployed to monitor `catalogue`. At the same time, the `Deployment` in Figure 1 includes another container than `catalogue`, viz., `users`. The latter is not an init container, nor identified as an implementation of a sidecar, ambassador, or adapter (according to the above listed criteria). The container `users` should hence be considered an occurrence of the *multiple services per deployment unit* smell.

```
kind: Deployment
metadata:
  name: catalogue
  labels:
    service: catalogue
spec:
  template:
    initContainers:
    - name: catalogue-loader
      image: example/catalogue-loader
    containers:
    - name: catalogue
      image: example/catalogue
    - name: dynatrace
      image: dynatrace/oneagent
    - name: users
      image: example/users
      resources:
        requests:
          memory: 50Mi
  metadata:
    labels:
      service: catalogue
  selector:
    matchLabels:
      service: catalogue
  replicas: 2
```

Figure 1: Example of pod specification, through a manifest of type `Deployment`.

3.1.2 Smell Detection

The overall idea is to ignore init, sidecar, ambassador, and adapter containers when looking for occurrences of the *multiple services per deployment unit* smell. Their use is intentional and intended to suitably run the service running in the "main container" of the pod (Kubernetes, 2022), with the main container identified as the first container that is neither an init container nor an implementation of a sidecar, ambassador, or adapter. The same does not hold for any other container deployed alongside the main container, which may be running another service, hence witnessing a possible occurrence of the *multiple services per deployment unit* smell. An example of this is the `users` container in Figure 1, which is deployed alongside the main `catalogue` container, while not being an init, sidecar, ambassador, or adapter container.

The above idea is realized by considering each manifest file of type `Pod` or `Deployment`, whose `initContainers` field is directly skipped. Then, each container in the `containers` section, other than the main container, is processed as follows: if the container is an implementation of a sidecar, ambassador, or adapter pattern, it is ignored. This is identified by checking whether the properties `name` or `image` of the container include the keywords *sidecar*, *ambassador*, or *adapter*, or whether they include the name or Docker image of

Table 1: Examples of software distributions known to implement sidecar, ambassador, or adapter patterns.

Software Name (Docker Image)
Datadog (datadog/agent), Dynatrace (dynatrace/oneagent), Hitch (hitch), Logstash (logstash), OAuth2 Proxy (bitnami/oauth2-proxy), Prometheus (bitnami/prometheus)

software distributions known to implement a sidecar, ambassador, or adapter pattern (Table 1). In any other case, since the container may run another service than the main container, it is considered to denote a *multiple services per deployment unit* smell.

Going back to the example in Figure 1, catalogue is identified as the main container, as we ignore catalogue-loader (init container) and dynatrace (which appears in Table 1). Instead, the users container does not satisfy any of the conditions for being ignored, and it is hence considered an occurrence of the *multiple services per deployment unit* smell.

3.1.3 Resolution Template Generation

The detected occurrences of the *multiple services per deployment unit* smell can be resolved by implementing the only known refactoring, namely by keeping only one service in the pod and moving the others to different pods. This is done by keeping the service running in the main container in the pod, together with any init, sidecar, ambassador, or adapter containers therein. Each other container possibly running a service is instead migrated to a different Deployment, whose metadata indicate that it runs a service named as the container's name, and which preserves the original configuration of the container itself.

Figure 2 provides an example of what above, by displaying the Deployments obtained by refactoring that in Figure 1 to resolve the *multiple services per deployment unit* smell due to users. The latter is removed from the Deployment of catalogue (Figure 2a) and moved to a new Deployment (Figure 2b). The name of the new Deployment is the same as that of the migrated container, and it preserves its original configuration. It indeed indicates the same requests on resources, and it is set to be replicated twice.

3.2 Endpoint-Based Service Interaction

The *endpoint-based service interaction* smell occurs when a microservices directly invokes another microservice's instance, e.g., with no intermediate component balancing the requests arriving to the replicated instances of the invoked service (Neri et al., 2020).

```
kind: Deployment
metadata:
  name: catalogue
  labels:
    service: catalogue
spec:
  template:
    initContainers:
    - name: catalogue-loader
      image: example/catalogue-loader
    containers:
    - name: catalogue
      image: example/catalogue
    - name: dynatrace
      image: dynatrace/oneagent
  metadata:
    labels:
      service: catalogue
  selector:
    matchLabels:
      service: catalogue
  replicas: 2
```

(a)

```
kind: Deployment
metadata:
  name: users
  labels:
    service: users
spec:
  template:
    containers:
    - name: users
      image: example/users
      resources:
        requests :
          memory: 50Mi
  metadata:
    labels:
      service: users
  selector:
    matchLabels:
      service: users
  replicas: 2
```

(b)

Figure 2: Refactoring templates for resolving the *multiple services per deployment unit* smell in Figure 1.

3.2.1 Related Kubernetes Objects

Pods are dinamically spawned/destroyed to match the desired state for a deployed microservices application. Each pod gets its own IP address, which can be fixed, however also meaning that different instances of the same microservice will get different IP addresses (Kubernetes, 2022).

Kubernetes Services are message routing components allowing to decouple invokers and invoked microservices. They indeed provide an abstract way to expose the multiple replicas of the invoked microservice's pod with the same hostname. When receiving a request sent to a microservice's hostname, a Kubernetes Service forwards such request to one of the corresponding pod's instances (Kubernetes, 2022).

The association between a Kubernetes Service and the pods it handles is specified through the selector fields in both manifest files. An example of this is given in the manifest file in Figure 3, which displays a Kubernetes Service for handling the traffic sent to the catalogue microservice (Fig-

```
kind: Service
metadata:
  name: catalogue-service
spec:
  type: ClusterIP
  selector:
    service: catalogue
```

Figure 3: Example of Kubernetes `Service`.

ure 2a). The hostname of the target microservice is set to `catalogue-service`, in the `metadata` field of the manifest in the figure. The association `Service`-`Deployment` is instead specified in the `selector` fields of the manifest files in Figure 2a and Figure 3. Finally, the manifest file in Figure 3 indicates that the type of the Kubernetes `Service` is `ClusterIP`, which means that it is reachable only from within the cluster where the application is deployed (Kubernetes, 2022).

3.2.2 Smell Detection

When processing the manifest files specifying an application's deployment, we check whether there exists at least one Kubernetes `Service` associated with each `Pod` or `Deployment` specifying the deployment of a microservice that can be invoked by other microservices.[2] A microservice with no associated Kubernetes `Service` is considered an occurrence of the *endpoint-based service interaction* smell. Indeed, without a Kubernetes `Service` routing the requests to the possible replicas of the pod running such microservice, its instances' endpoints should necessarily be directly contacted by the invoking microservices.

For instance, suppose that we process the manifest files in Figures 2 and 3, and that all the microservices deployed with such manifest files may get invoked by other microservices. We would detect that there is no Kubernetes `Service` handling the requests sent to the `users` microservice, whose `Deployment` specifies that such microservice is replicated twice (Figure 2b). Any other microservice invoking `users` should therefore directly invoke any of the two replicas, hence possibly witnessing the occurrence of an *endpoint-based service interaction* smell on `users`.

3.2.3 Resolution Template Generation

The occurrence of an *endpoint-based service interaction* smell can be resolved by introducing a Kubernetes `Service` to handle the requests sent to the replicas of the pod running the affected microservice. This is done by creating a new manifest file specifying the newly introduced Kubernetes `Service` of type `ClusterIP`, to ensure that the affected microservice can still be accessible only from within the cluster where the application is running. The newly introduced Kubernetes `Service` is then associated with the workload resource specifying the deployment of the affected microservice, either reusing the `selector` already specified therein, or by adding a new `selector` in both manifest files.

[2]We assume the list of invoked microservices to be provided as input, e.g., as described in Section 4.

```
kind: Service
metadata:
  name: users-service
spec:
  type: ClusterIP
  selector:
    service: users
```

Figure 4: Refactoring template for resolving the *endpoint-based service interaction* smell affecting `users`.

The above refactoring is exemplified in Figure 4, which shows the Kubernetes `Service` introduced as template for resolving the smell on `users`. Given that the `Deployment` of `users` (Figure 2b) already specifies a `selector`, we reuse the same `selector` in the newly introduced Kubernetes `Service`.

3.3 No API Gateway

The *no API gateway* smell occurs whenever the clients of a microservices directly invoke any of its microservices. This does not mean that there is no API gateway set for the whole application, but rather that no API gateway is used to handle the external requests sent to the affected microservices (Neri et al., 2020).

3.3.1 Related Kubernetes Objects

The `Deployment` or `Pod` specifying the deployment of a microservice allow specifying that the pod where it runs can be directly accessed from external clients. This can be done by setting the properties `hostNetwork` and `hostPort` (Kubernetes, 2022). The property `hostNetwork` can be set for the whole pod, allowing all its containers to be accessibile from all the network interfaces of the host where the pod runs. Instead, the property `hostPort` can be set on each container running in a pod, to expose it on a given port of the host where it runs. Both properties are however considered *privileged operations*, which should be used to enable specific network plugins, rather than to expose microservices outside of the cluster where they run (VMWare, 2020).

Pods should better be exposed by exploiting the Kubernetes objects devoted to that purpose. For instance, Kubernetes `Services` allow to handle the traffic arriving to pods' replicas not only from other microservices running in the same cluster, but also from external clients. Indeed, by specifying Kubernetes

Table 2: Examples of software distributions known to implement API gateways.

Software Name (Docker Image)
Apache APISIX (apache/apisix), Envoy (envoy/envoyproxy), Kong (kong/kong-gateway), NGINX (nginx), Traefik (traefik)

```
kind: Deployment
metadata:
  name: payments
  labels:
    service: payments
spec:
  template:
    containers:
    - name: payments
      image: example/payments
      ports:
        containerPort: 80
        hostPort: 4040
  metadata:
    labels:
      service: payments
  selector:
    matchLabels:
      service: payments
```

Figure 5: Example of *no API gateway* smell.

Services of type NodePort or LoadBalancer, such Services get accessible from outside of the cluster where they run. A Kubernetes Service of type NodePort is exposed on a given port of the host where the Service runs, while those of type LoadBalancer are exposed externally by using a cloud provider's load balancer (Kubernetes, 2022). Another possibility to expose microservices externally, yet through Kubernetes Services is to define Ingress nodes redirecting external requests to such Services (Poulton, 2022).

3.3.2 Smell Detection

No API gateway smells are detected by analyzing each Pod or Deployment specifying how to deploy microservice. If such Pod or Deployment specifies either of the hostNetwork and hostPort properties, the deployed component is exposed outside of the cluster where it runs. This is not a problem if the component is implementing an API gatweay itself. This is identified by checking whether the properties name or image of the main container include the keyword *gateway*, or whether they include the name or Docker image of software distributions typically used to implement API gateways (Table 2). In any other case, since the exposed container may run a microservice that is directly accessible by external clients, it is considered to denote a *no API gateway* smell.

Consider, for instance, the manifest file in Figure 5, which specifies the Deployment of the payments microservice. The property hostPort of the main container is set to expose payments on port 4040. External clients can hence directly invoke payments, without passing through any API gateway (even if one is set for payments itself). This, together with the fact that the main container does not satisfy the above listed criteria for identifying implementations of API gateways, makes payments to be considered as affected by a *no API gateway* smell.

3.3.3 Resolution Template Generation

A no API gateway smell due to the use of hostNetwork or hostPort can be resolved by removing them, however considering that they were specified to expose a microservice externally, which is something to be preserved. The refactoring template hence also consists of introducing a "basic API gateway", namely a message routing component for handling external requests and forwarding them to the affected microservice (Neri et al., 2020). For this reason, by default, we introduce a Kubernetes Service of type NodePort,[3] which is accessible by external client on a given port of the host where it runs. The port mapping is the same as that indicated in the original configuration of the affected microservice, if available. Otherwise, the default choice is to use port 8080, as a placeholder for later adapting the port mapping to better fit the application's setting. A possible alternative is introducing a Kubernetes ClusterIP Service, and configuring an Ingress node to listen on the port where the microservice was exposed (or on port 8080, as a placeholder) and to redirect incoming requests to the newly introduced Kubernetes Service.

Figure 6 provides an example of the default resolution template generation. The figure displays the updated Deployment of payments and a new Kubernetes Service, obtained by refactoring its original Deployment in Figure 5. The Deployment is updated by removing hostPort, and it is associated with the newly introduced Kubernetes Service by reusing the already available selector. Such Service is of type NodePort and provides the same port mapping as in the original configuration of payments, being it accessible on port 4040. It implements a basic API gateway, meaning that it implements the message routing mechanism needed to handle the requests sent to payments, by suitably routing them to any pod instantiated from the Deployment in Figure 6a.

3.4 Wobbly Service Interaction

A *wobbly service interaction* occurs whenever a microservice invokes another without mechanisms for

[3]The association between the newly introduced Kubernetes Service and the workload resource is realized via selectors, similarly to the case of endpoint-based service interactions (Section 3.2).

```
kind: Deployment
metadata:
  name: payments
  labels:
    service: payments
spec:
  template:
    containers:
    - name: payments
      image: example/payments
      ports:
        containerPort: 80
  metadata:
    labels:
      service: payments
  selector:
    matchLabels:
      service: payments
```

(a)

```
kind: Service
metadata:
  name: payments-service
spec:
  type: NodePort
  selector:
    service: payments
  ports:   containers:
    - name: payments
      image: example/payments
      ports:
        - port: 80
          targetPort: 80
          nodePort: 4040
```

(b)

Figure 6: Refactored manifest files, obtained by resolving the *no API gateway* smell in Figure 5.

tolerating the failure of the invoked microservice, such as, e.g., timeouts or circuit breakers (Section 2).

3.4.1 Related Kubernetes Objects

Timeouts and circuit breakers can be set by managing the traffic sent to a pod with Istio, a Kubernetes-native traffic management system. Istio's `VirtualServices` allow explicitly setting a `timeout` to indicate the maximum amount of time after which the interaction with a microservice is considered to have failed (Istio, 2022). For instance, the `VirtualService` in Figure 7a sets a `timeout` of `0.5s` for requests sent to `payments` (Figure 6a).

Instead, Istio's `DestinationRules` allow setting an `outlierDetection` policy, through which circuit breakers can be set by indicating This is done by setting the maximum number of tolerated consecutive errors before the circuit breaker trips (Istio, 2022). For instance, the `DestinationRule` in Figure 7b sets a circuit breaker for `users` (Figure 2b), which trips after two consecutive errors are returned by `users`.

3.4.2 Smell Detection

We focus on detecting the lack of timeouts and circuit breakers *only* in the manifest files specifying the deployment of a microservices application. Similarly to the detection of *endpoint-based service interac-*

```
kind: VirtualService
spec:
  hosts:
   - payments
  http:
   - route:
       - destination:
           host: payments
     timeout: 0.5s
```

(a)

```
kind: DestinationRule
spec:
  host: users
  trafficPolicy:
    outlierDetection:
      consecutive5xxErrors: 2
```

(b)

Figure 7: Examples of (a) timeouts and (b) circuit breakers defined with Istio.

*tion*s, we start from an input list of invokable microservices, and we check whether the `Deployment` or `Pod` specifying their deployment are associated with `VirtualServices` or `DestinationRules` setting timeouts or circuit breakers, respectively. Each microservice whose pod deployment is not associated to one such `VirtualService` or `DestinationRule` is considered as affected by the *wobbly service interaction* smell, as the microservices sending it requests would not tolerate its failure – unless timeouts or circuit breakers are set in their source code, which are not considered by our "black-box" approach.

For instance, suppose that we process the manifest files in Figures 2 to 7, and that all the microservices deployed with such manifest files may get invoked by other microservices. There is a `VirtualService` setting a timeout for `payments` (Figure 7a) and a `DestinationRule` setting a circuit breaker for `users` (Figure 7b). The same does not hold for `catalogue`, for which there is no timeout or circuit breaker set in the manifest files specifying its deployment in Kubernetes. This means that the microservices invoking `catalogue` may not tolerate its failure, hence possibly failing in cascade – unless they have fault tolerance mechanisms set in their source code. For this reason, and given that we apply a "black-box" approach by focusing on what we can observe from the Kubernetes manifest files, we would detect a *wobbly service interaction* smell on `catalogue`.

3.4.3 Resolution Template Generation

A *wobbly service interaction* affecting a microservice can be resolved by introducing a `VirtualService` setting a timeout for the requests sent to such microservice, or by introducing a `DestinationRule` setting a circuit breaker for such requests. This can be done by essentially mirroring the configuration in the manifest files in Figure 7. The `VirtualService` resolution approach is applied by default, by introducing

```
kind: VirtualService
spec:
  hosts:
   - catalogue
  http:
   - route:
       -  destination:
            host: catalogue
     timeout: 0.5s
```

Figure 8: Refactoring template for resolving the *wobbly service interaction* smell affecting `catalogue`.

a manifest file specifying a `VirtualService`, whose `hosts` and `destination` fields indicate that it handles the requests sent to the affected microservice, with a `timeout` set to `0.5s`. Alternatively, the `DestinationRule` resolution approach can be applied by introducing manifest file specifying a `DestinationRule`, whose `host` field indicates that it handles the requests sent to the affected microservice, with a `trafficPolicy` configuring a circuit breaker that trips after two consecutive errors.

Figure 8 displays the manifest file introduced as resolution template for the *wobbly service interaction* smell affecting the `catalogue` microservice. The manifest file specifies a `VirtualService` that handles the requests sent to `catalogue`, which is listed within the `host` field, and which is indicated as the `destination` of the `route` of the `VirtualService` itself. The manifest file also sets a `timeout` of `0.5s` for such a `route`, hence setting a timeout for all requests sent to the targeted microservice (Istio, 2022).

3.5 Summary and Discussion

Sections 3.1 to 3.4 illustrate how to automatically detect architectural smells by analyzing the manifest files specifying the deployment of a microservices application in Kubernetes. Our methodology is "black-box", as we analyze *only* the information specified in Kubernetes, by abstracting from the internals of the containerized microservices forming an application. This enables supporting polyglot microservices applications, by focusing on their deployment Kubernetes, which is the de-facto standard for microservices deployment (Indrasiri and Siriwardena, 2018).

At the same time, an architectural smell automatically identified by our methodology does not necessarily mean that some microservices' design principle is truly violated. For instance, we may detect a *multiple services per deployment* smell on a pod with two containers even if one actually implements a sidecar, but its name does not include the keyword *sidecar* or it does not run from a known implementation of sidecars. We may also detect a *wobbly service interaction* smell affecting a microservice, but the microservices

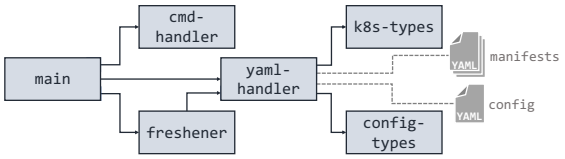

Figure 9: The architecture of `KubeFreshener`.

invoking it may implement some timeout or circuit breaker directly within their source code, which we cannot see. This is however in line with the definition of architectural smell themselves: they are only possible *symptoms* of bad design decisions, which may possibly result in a violation of the design principles of microservices in our case (Neri et al., 2020). Anyhow, to reduce the rate of false postives, the prototypical implementation of our methodology (Section 4) already enables to specify which smells to ignore on which microservices, e.g., since such microservices are known to implement the necessary countermeasures in their source code.

Similarly, the proposed resolution templates automatically adapt the deployment of a microservices application in Kubernetes. The proposed refactoring exploits Kuberntes elements to introduce what needed, but the same refactoring could be achieved by suitably adapting the source code of affected microservices, or using other extensions of Kubernetes. Also, while the update/generation of Kubernetes manifest files can be fully automated, the overall resolution approach is semi-automated. The microservices application and its deployment in Kubernetes may indeed need some further adaptation to continue working. For instance, a microservice may need to be updated to invoke the Kubernetes `Service` introduced to resolve an *endpoint-based service interaction* smell, or init containers may need to be added to the `Deployments` introduced to resolve *multiple services per deployment unit* smells.

4 PROTOTYPE

We hereafter introduce `KubeFreshener`, a Rust implementation of the semi-automated smell resolution methodology described in Section 3. `KubeFreshener` is open-source and publicly available on GitHub.[4]

4.1 Architecture of `KubeFreshener`

`KubeFreshener` is structured in six Rust modules, as displayed in Figure 9. The main module implements a command-line interface to run the proposed semi-automated smell resolution by coordinating the other modules. In particular, it first invokes the cmd-handler

[4]https://github.com/di-unipi-socc/kube-freshener.

to process the command-line inputs, which include the analysis to run and the options to customize such analysis. The main module then invokes the yaml-handler to parse the YAML manifests specifying the deployment of an application in Kubernetes, which are transformed in Rust objects by relying on the object model defined by k8s-types. The yaml-handler also parses the YAML config file specifying the analysis' configuration, e.g., which smells to ignore on which services, yet instantiating Rust objects by relying on the object model defined by k8s-types. The obtained objects are returned to the main module, which passes them to the freshener. The latter processes such objects by implementing the semi-automated smell identification described in Section 3, returning a report on identified smells to the main module. If the analysis is also specified to generate the refactoring templates, the freshener also updates the objects representing the application deployment in Kubernetes accordingly. The main module finally invokes the yaml-handler to streamline the (possibly updated) objects representing the application deployment back to YAML-based Kubernetes manifest files, and it outputs the results of the smell analysis on the command-line.

4.2 Using KubeFreshener

After cloning its GitHub repository, KubeFreshener should be configured by placing the manifest files specifying the Kubernetes deployment to be analyzed in a new subfolder called manifests. The run of KubeFreshener can be further configured by editing a file config.yaml, which enables specifying (i) the list of invoked_services and (ii) the list ignore_smells indicating which architectural smells should not be checked on which microservices. The list (i) enables checking for *endpoint-based* and *wobbly service interactions* only those microservices that are actually invoked by other microservices, as described in Sections 3.2 and 3.4. Instead, the list (ii) enables reducing the rate of false positives, by allowing to indicate, e.g., which microservices are known to implement what needed to avoid architectural smells in their source code (as discussed in Section 3.5)

Once the configuration is completed, KubeFreshener can be launched by issuing the command

```
$ cargo run analyze.
```

This will run KubeFreshener in detection mode only, meaning that it will output only the list of architectural smells that have been automatically identified on the considered deployment, if any (Figure 10). By specifying option "-s", instead, KubeFreshener will also generate the templates for the needed refactorings, by adapting the manifest files in the folder manifests as described in Section 3.

```
! [Multiple containers per unit]
(*) payments is deployed alongside search-
engine, which may not be a sidecar
Hint: deploy payments and search-engine
separately

! [No API Gateway]
(*) payments has HostNetwork set, but it
may not implement any message routing
Hint: unset HostNetwork on payments.

! [Endpoint-based interaction]
(*) Service catalogue is invoked by other
microservices, but there's no associated
Kubernetes Service
Hint: add a Kubernetes Service handling
the requests sent to catalogue

! [Wobbly Interaction]
(*) Service named catalogue is reached by
another service without any circuit breaker
or timeout.
Hint: solve it by adding a circuit breaker
and/or a timeout in between.
```

Figure 10: Example of output returned by KubeFreshener.

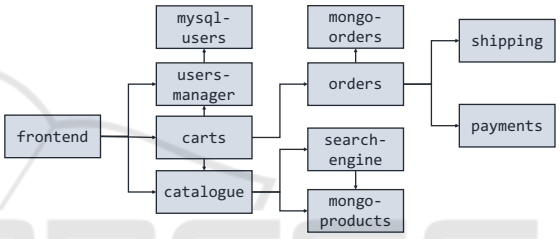

Figure 11: Architecture of the toy microservices application in our controlled experiment. Boxes denote services, while oriented arcs denote service interactions.

5 EVALUATION

To assess the practical applicability of our semi-automated smell resolution methodology, we applied KubeFreshener in a controlled experiment (Section 5.1) and a case study (Section 5.2).

5.1 Controlled Experiment

The objective of our controlled experiment was to test whether our methodology can effectively detect the considered architectural smells and generate their resolution templates. We hence devised a Kubernetes deployment for the toy microservices application in Figure 11. The deployment was set to include each of the four architectural smells considered in Section 3. The microservices catalogue and search-engine were included in the same Deployment to inject a *multiple services per deployment unit* smell. A *no API gateway* smell was instead injected on payments by setting

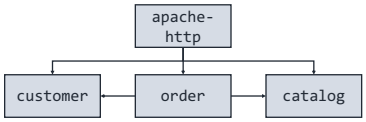

Figure 12: Architecture of the microservices application in our case study.

hostNetwork to true in its Deployment. Finally, we injected *endpoint-based* and *wobbly service interaction* smell on catalogue, which was not associated with a Kubernetes Service routing incoming requests among its possible replicas, nor with any Istio-based timeout or circuit breaker.

We configured KubeFreshener to process the specified deployment, while also indicating the list of microservices that are invoked by other microservices. We then run KubeFreshener, which successfully detected all the four injected smells, as displayed in Figure 10. Finally, we tested the smell resolution, observing that KubeFreshener was capable of generating the smell resolution templates, which are available alongside the input manifest files on GitHub.[5]

5.2 Case Study

The objective of our case study was to assess the applicability of our methodology to a third-party microservices application. We therefore configured KubeFreshener to process an existing demo application from (Wolff, 2016), whose Kubernetes deployment is publicly available on GitHub.[6] Given the application's documented architecture (Figure 12), we also configured KubeFreshener to consider that the microservices customer, order, and catalog are invoked by another microservice. We then run KubeFreshener, which identified three *wobbly service interaction* smells, as shown in Figure 13.

We further experimented on the identified smells, by artificially injecting a deterministic failure of catalog. The source code of catalog was indeed updated to never reply when invoked, to check whether apache-http was failing in cascade. We run the updated deployment, and invoked apache-http to stimulate its interaction with catalog. As a result, we observed that apache-http never replied to our invocations, hence witnessing a cascading failure due to the failure injected in catalog.

We hence re-run KubeFreshener to automatically resolve the identified *wobbly service interaction* smells. The refactored manifest files – which are publicly

[5]https://github.com/di-unipi-socc/kube-freshener/tree/main/data/examples/controlled-exp.

[6]https://github.com/ewolff/microservice-kubernetes.

```
! [Wobbly Interaction]
(*) Service named customer is reached by
another service without any circuit breaker
or timeout.
Hint: solve it by adding a circuit breaker
and/or a timeout in between.

! [Wobbly Interaction]
(*) Service named order is reached by
another service without any circuit breaker
or timeout.
Hint: solve it by adding a circuit breaker
and/or a timeout in between.

! [Wobbly Interaction]
(*) Service named catalog is reached by
another service without any circuit breaker
or timeout.
Hint: solve it by adding a circuit breaker
and/or a timeout in between.
```

Figure 13: Output of KubeFreshener in our case study.

available on GitHub[7] – include three new VirtualServices setting timeouts for customer, order, and catalog as described in Section 3.4. Such manifest files were used "as-is" to repeat the above described experiment. As a result, we observed that apache-http came back replying when interacting with catalog, even if the latter did not reply to its invocations.

At the same time, apache-http returned a 500 error when catalog was not replying. This witnesses why our smell resolution is semi-automated: while the updated deployment may resolve the smell-related issue, the application would require the refactoring to be completed to tolerate the failures of catalog. In this particular case, for instance, we would also need to update apache-http to better handle the situation when the newly introduced timeouts expire.

6 RELATED WORK

Various existing approaches contribute to resolving architectural smells in microservices. The closest to ours is perhaps that in (Soldani et al., 2021), which models the architecture of a microservices application in the OASIS standard TOSCA (OASIS, 2020) and enact a model-based analysis to detect and reason on how to refactor the architectural smells in (Neri et al., 2020). (Soldani et al., 2021) also allows reconstructing the TOSCA modeling of a microservices application from its Kubernetes deployment. At the same time, the TOSCA modeling abstacts from the actual deployment of services in pods, hence not allowing to resolve the *multiple services per pod* smell (as we do). Also, our approach differs from that in (Soldani et al., 2021), since we can automatically generate the refactoring templates for resolving identified smells, if any.

[7]https://github.com/di-unipi-socc/kube-freshener/tree/main/data:examples/case-study.

(Pigazzini et al., 2020) instead automatically identifies three architectural smells from (Taibi and Lenarduzzi, 2018), viz., cyclic dependencies, hardcoded endpoints, and shared persistence. This is done by adapting Arcan (Fontana et al., 2017), which statically analyzes the Java sources of given software projects, to detect the above three smells in microservices. Hence, (Pigazzini et al., 2020) differs from our methodology since it targets Java-based microservices, while our "black-box" analysis enables applying it to polyglot microservices. Also, we try to go beyond just detecting architectural smells, by automatically generating refactoring templates for resolving detected smells.

Other approaches worth mentioning are (Balalaie et al., 2018), (Haselböck et al., 2017), and (Ponce et al., 2022a), which all focus on enabling to design "smell-free" applications. (Balalaie et al., 2018) and (Haselböck et al., 2017) actually support the migrations of applications to microservices, by providing patterns and decision models for such task, respectively. (Ponce et al., 2022a) instead proposes a trade-off analysis for resolving security smells in microservices applications, with such smells taken from (Ponce et al., 2022b). We further support the design of "smell-free" microservices applications, as we can automatically detect architectural smells from their Kubernetes deployment, while also generating refactoring templates to resolve detected smells.

Finally, it is worth relating our contribution to existing solutions for identifying and resolving architectural smells in classical service-oriented applications. (Garcia et al., 2009) and (Sanchez et al., 2015) detect smells in the design of a single service, given its specification. (Arcelli et al., 2019), (Fontana et al., 2017), and (Vidal et al., 2015) instead provide programming language-specific analyses to resolve the smells contained in the source code of a single service. We instead focus on resolving architectural smells in a whole application, by considering the interactions occurring among its microservices and their deployment. Also, by enacting a "black-box" analysis, we natively work with polyglot microservices. The above approaches are anyhow complementary to ours: they can be used to analyze and refactor the internals of a microservice, while ours can be used to resolve the architectural smells of an overall microservices application.

In summary, to the best of our knowledge, ours is the first solution for semi-automatically resolving architectural smells in polyglot microservices applications, by analyzing and generating refactoring templates of their deployment in Kubernetes.

7 CONCLUSIONS

We illustrated a semi-automated solution for resolving smells in microservices applications. The main contributions of our paper are actually threefold, viz., (i) a methodology that automatically detects smell occurrences by analyzing the manifest files specifying an application's deployment in Kubernetes, which also generates the templates of the necessary refactorings by suitably adapting such manifest files, (ii) `KubeFreshener`, a prototypical implementation of our methodology, and (iii) the application of `KubeFreshener` to a controlled experiment and a case study, with the goal of assessing our methodology's practical applicability.

As pointed out in Section 3.5, an architectural smell automatically identified by our methodology does not necessarily mean that some microservices' design principle is truly violated, which is the reason why `KubeFreshener` enables specifying which smells to ignore on which microservices. For instance, we may detect that a given microservice is affected by a *wobbly service interaction*, as no timeout or circuit breaker is set in the Kubernetes deployment, but the microservices invoking it may actually implement timeouts/circuit breakers in their source code. For future work, we plan to enhance the support for such a kind of situations, by integrating our methodology with other approaches considering different inputs, e.g., (Soldani et al., 2021) or (Pigazzini et al., 2020), to automatically determine which architectural smells could be ignored on which microservices.

We also plan to enhance the usability of `KubeFreshener`, moving from the current textual input/output to a graphical support, e.g., displaying the application deployment in Kubernetes, and highlighting detected smells and refactoring templates. More broadly, we plan to enhance the smell detection capabilities of our methodology by supporting a wider set of architectural smells, such as, e.g., those in (Carrasco et al., 2018) or (Taibi and Lenarduzzi, 2018), or enabling to detect other types of smells, such as, e.g., the microservices' security smells in (Ponce et al., 2022b). We also plan to enhance the smell resolution capabilities of our methodology, by (i) supporting standards like the Service Mesh Interface (Cloud Native Computing Foundation, 2022) to generalize the refactoring templates, (ii) fully automating the refactoring when possible, and (iii) enabling to reason on whether to apply a refactoring, e.g., with a trade-off analysis like that in (Ponce et al., 2022a).

ACKNOWLEDGEMENTS

This work was partly supported by the project *hOlistic Sustainable Management of distributed softWARE systems* (OSMWARE, UNIPI PRA_2022_64), funded by the University of Pisa, Italy.

REFERENCES

Arcelli, D., Cortellessa, V., and Pompeo, D. D. (2019). Automating performance antipattern detection and software refactoring in UML models. In Wang, X. et al., editors, *2019 International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, pages 639–643. IEEE Computer Society.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., and Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042.

Carrasco, A., Bladel, B. v., and Demeyer, S. (2018). Migrating towards microservices: Migration and architecture smells. In Ouni, A. et al., editors, *Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018*, pages 1–6. Association for Computing Machinery.

Cloud Native Computing Foundation (2022). Service mesh interface. https://smi-spec.io.

Fontana, F. A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., and Di Nitto, E. (2017). Arcan: A tool for architectural smells detection. In Malavolta, I. and Capilla, R., editors, *2017 IEEE International Conference on Software Architecture Workshops, ICSA 2017 Workshops*, pages 282–285. IEEE Computer Society.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In Winter, A. et al., editors, *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, CSMR 2009*, pages 255–258, USA. IEEE Computer Society.

Haselböck, S., Weinreich, R., and Buchgeher, G. (2017). Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In Lopes, A. and de Lemos, R., editors, *Software Architecture, ECSA 2017*, pages 155–170, Cham. Springer International Publishing.

Herrera, J., Berrocal, J., Forti, S., Brogi, A., and Murilo, J. (2023). Continuous qos-aware adaptation of cloud-iot application placements. *Computing*.

Indrasiri, K. and Siriwardena, P. (2018). *Microservices for the Enterprise*. Apress Berkeley, CA, 1 edition.

Istio (2022). Documentation. https://istio.io/latest/docs/.

Kubernetes (2022). Documentation. https://kubernetes.io/docs/home/.

Neri, D., Soldani, J., Zimmermann, O., and Brogi, A. (2020). Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):3–15.

OASIS (2020). TOSCA Simple Profile in YAML, version 1.3. OASIS Standard.

Pigazzini, I., Fontana, F. A., Lenarduzzi, V., and Taibi, D. (2020). Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt, TechDebt 2020*, page 92–97. Association for Computing Machinery.

Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022a). Should microservice security smells stay or be refactored? towards a trade-off analysis. In Gerostathopoulos, I. et al., editors, *Software Architecture*, pages 131–139, Cham. Springer International Publishing.

Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022b). Smells and refactorings for microservices security: A multivocal literature review. *Journal of Systems and Software*, 192:111393.

Poulton, N. (2022). *The Kubernetes Book*. Independently published, 2022 edition.

Richardson, C. (2018). *Microservices Patterns*. Manning Publications, 1 edition.

Sanchez, A., Barbosa, L. S., and Madeira, A. (2015). Modelling and verifying smell-free architectures with the archery language. In Canal, C. and Idani, A., editors, *Software Engineering and Formal Methods, SEFM 2015*, pages 147–163, Cham. Springer International Publishing.

Soldani, J., Muntoni, G., Neri, D., and Brogi, A. (2021). The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621.

Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232.

Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). JSpIRIT: A flexible tool for the analysis of code smells. In Marín, B. and Soto, R., editors, *34th International Conference of the Chilean Computer Science Society, SCCC 2015*, pages 1–6. IEEE Computer Society.

VMWare (2020). 3 common kubernetes security mistakes and how to avoid them. White Paper.

Wolff, E. (2016). *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 1 edition.

Yussupov, V., Breitenbcher, U., Krieger, C., Leymann, F., Soldani, J., and Wurster, M. (2020). Pattern-based modelling, integration, and deployment of microservice architectures. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 40–50. IEEE Computer Society.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310.