UNIVERSITÁ DEGLI STUDI DI MILANO - BICOCCA

Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

DEPLOYMENT-AWARE ANALYSIS OF MICROSERVICE-BASED APPLICATIONS

Relatori:
Francesca Arcelli Fontana
Antonio Brogi
Jacopo Soldani

Relazione della prova finale di:
Davide Rendina
Matricola 830730

Anno Accademico 2021-2022

Abstract

This thesis introduces an extension of the μTOSCA toolchain, which is a collection of tools that can mine the microservices architecture of existing applications, detect architectural smells of microservices, and refactor the architecture to resolve any identified smells. The toolchain is based on the μTOSCA modeling language, which allows the modeling of a microservices architecture as a typed-oriented graph. The extension involves introducing the detection of a new microservice smell, improving existing smell detection, and refactoring the identified smells. This is achieved by modifying an existing tool (μFreshener) and introducing a new one (μFreshener++) to the toolchain. The detection of a new architectural smell for microservices is implemented into the μFreshener tool, and required also the extension of the μTOSCA language to represent the deployment of microservices in the model. Improvements to the smell detection capabilities are achieved through the *extension* functionality of the μFreshener++ tool, which corrects and enriches the μTOSCA model by using information obtained from the Kubernetes deployment files of the analyzed system. The refactoring feature of μFreshener++ applies five refactoring techniques to solve four architectural smells, and modifies the Kubernetes deployment of the application for solving smells. The validation of μFreshener++ was performed using two existing third-party open-source projects, and the results demonstrated that it can detect the new smell, improve the smell detection of the μTOSCA toolchain and refactor the smells identified in the system.

Contents

1 Introduction

The software industry recently observed a growing popularity of microservices architecture (MSA), which enables realizing so-called *cloud-native* applications [26]. Major companies such as Amazon, LinkedIn, Netflix, Spotify, and Google indeed already adopted microservices for delivering their core businesses [19] [29].

A microservice-based application is essentially composed of multiple microservices that interact to deliver the overall application functionality. Microservice-based architectures can actually be seen as a particular extension of service-oriented architectures (SOA), characterized by a set of extended design principles [19]. Ensuring the adherence to such design principles is hence crucial for realizing the full potential and benefits of microservice-based systems.

In this perspective, Neri et al. [19] elicited four key design principles for microservices, by also outlining the architectural smells[1] that may result in violations of such principles. Neri et al. [19] also identified the architectural refactorings that are known to resolve the occurrence of microservices' architectural smells. Soldani et al. [27] then exploited the such design principles, architectural smells, and refactorings to develop the μTOSCA toolchain, which comprises tools to mine the microservices architecture of an existing application, to determine the occurrence of architectural smells therein, and to reason on whether/how to refactor the architecture to resolve the occurrence of identified smells. The toolchain is based on the μTOSCA modeling language, which is a profile of the OASIS TOSCA [20] standard that enables modeling a microservices architecture as a typed oriented graph, with nodes representing application components, and oriented arcs denoting the interactions occurring among such nodes. The μTOSCA toolchain exploits such language to describe the architecture of microservice-based applications. A μTOSCA model is indeed returned by the μMiner and μTOM, which are the tools enabling to reconstruct the microservices architecture of an existing application from its deployment in Kubernetes [14]. The obtained μTOSCA model is then the input for μFreshener, which is the tool enabling to identify a subset of the architectural smells from Neri et al. [18] in modeled applications. μFreshener also suggests possible refactorings for resolving identified smells, which can be applied to refactor the μTOSCA model, then relying on developers to concretely implement them in the application [26].

As for the analysis, we extend the μTOSCA model to enable modelling deployment-related information, and we exploit the proposed extension to enable detecting the architectural smell that may denote a violation of the *independent deployability* principle for microservices, as per its definition by Neri et al. [19]. We also exploit the information specified in the Kubernetes deployment of an application to refine the μTOSCA model (e.g., specifying that a microservice is accessible from external clients, if this is configured in the deployment). As for the refactoring, we not only enable reasoning on the

[1]An architectural smell can be observed in the architecture of an application, and it is the effect of a bad decision (though often unintentional) while architecting the application, which may negatively impact on the overall system quality [8].

refactoring of the newly supported smell, but we also enable automatically refactoring the Kubernetes deployment of an existing application to resolve all architectural smells identified therein. The proposed extensions are realized by means of a new version of μFreshener, called μFreshener++, which takes as input the μTOSCA model describing an application and the application deployment in Kubernetes, refactors them to resolve the architectural smells therein, and returns the refactored version of the μTOSCA model and Kubernetes deployment.

To validate μFreshener++, two existing, third-party, and open-source projects were used, viz., MFDemo [18] and Sock Shop [24]. We actually provided μFreshener++ with the μTOSCA models and Kubernetes deployment of both applications, also artificially injecting *independent deployability*-related smells and differences between models and deployments, so as to demonstrate that the extended analysis can detect smells on suitably refined μTOSCA models (hence avoiding false positives due to discrepancies between modeled and deployed applications, which μFreshener was instead detecting as smells). We also validated the refactoring capabilities of μFreshener++, by generating refactored deployments of both applications, and checking that the smells therein were effectively resolved.

In summary, the main contributions of this thesis are twofold:

- We introduce μFreshener++, a new version of the smell detection and refactoring tool of the μTOSCA toolchain, which can detect a wider set of smells, refine input models based on an application's deployment in Kubernetes, and automatically refactor such deployment to resolve identified smells.
- We assess the practical applicability and effectiveness of the proposed tool by means of two case studies based on two existing, third-party applications.

The rest of this thesis is organized as follows. Chapter 2 positions our contributions with respect to related work in the field. Chapter 3 provides the necessary background on container-based technologies, on the architectural smells for microservices, and on the μTOSCA toolchain. Chapter 4 provides a high-level view on the proposed tool, viz., μFreshener++ tool, whose enhanced smell detection and refactoring capabilities are described in Chapters 5 and 6, respectively. Chapter 7 presents the assessment of μFreshener based on two existing, third-party applications. Finally, Chapter 8 draws some concluding remarks and outlines possible future research directions.

2 Related Work

Various existing research works contribute to the design of microservices applications, therein including the migration from monolithic systems to microservices and the elicitation of known bad smells, design patterns, and anti-patterns for microservices. We hereafter focus on the design aspects most related to our research work, namely the definition of architectural smells for microservices (Section 2.1) and their detection/resolution (Section 2.2).

2.1 Architectural smells for microservices

Different existing studies provide catalogs and taxonomies of bad smells for microservices, by also describing their common problems and possible solutions. For instance, Waseem et al. [32] [33] conducted a study on issues in microservices systems and proposed a taxonomy. Their empirical study mixed different strategies and has been conducted by collecting data from 2 641 issues in 15 open-source systems, performing 15 interviews, and gathering answers from an online survey compiled by 150 practitioners. All information collected from these sources allowed authors to produce three different taxonomies for issues, their causes, and available solutions in microservices systems. Each taxonomy presents an organization in category and subcategories and also describes all possible types of issues (402 found), causes (228), and solutions (177). Then, the authors analyzed the results obtained and pointed out some major issues, their main causes, and how they are solved more frequently. The conducted study showed that the most found issues are related to Technical Debt, CI/CD, and Exception Handling, which are caused by General Programming Errors, Missing Features and Artifacts, and Invalid Configuration and Communication. For dealing with these issues, the authors showed that the most common strategies are Fixing, Adding, and Modifying artifacts.

Another taxonomy in literature was proposed by Taibi and Lenarduzzi [28] [29], who collected bad practices from the developers' point of view by conducting an industrial survey. In their first work they collected 256 bad practices and relative solutions and then derived, by analyzing the collected results, a catalog of eleven microservices smells. Interviews were conducted only with people with experience with microservice systems, so interviewing directly senior microservices developers, architects, project managers, and Agile coaches. They later extended this work by conducting mixed research, merging the results of industry surveys, interviews, and literature reviews, which, combined with the results of the previous work, allowed the definition of a taxonomy containing 20 anti-patterns, including eleven smells identified before. Bad practices were divided into technical and organizational, and for each, they presented a brief description of the problem and the best solution to adopt to remove it. In analyzing responses obtained, the authors also reported the experience of the practitioners who reported smells *Wrong Cuts*, *Hard-Coded Endpoints*, *Cyclic Dependencies*, and *Shared Persistence* as most harmful for the development.

Neri et al. [19] carried out a multivocal review for identifying the most-known architectural smells for microservices and their possible refactoring. They aim to organize the state-of-art and practice on the topic in a single work that can help both researchers and practitioners in their daily work with microservices. The main focus of Neri et al. [19] work was indeed to identify key design principles for microservices, together with the architectural smells that might denote violations of such principles, and the refactorings allowing to resolve such smells. Neri et al [19] identify four key design principles, viz., *independent deployability* of microservices, their *horizontal scalability*, the *isolation of failures* in the system, and *decentralization*. They then propose a taxonomy mapping including such design principles, seven architectural smells possibly denoting their violation, and 16 architectural refactorings for resolving such smells. The taxonomy maps each key design principle to the architectural smell that might violate it and each architectural smell to the architectural refactorings enabling the resolution of such smells. The study conducted by Neri et al. [19] proposed a taxonomy that reordered the results of previously cited works [28] [29] [32] [33], and was the reference point of Soldani et al. [27] for the development of the μTOSCA toolchain. The main purpose of this thesis is to present the work done for the extension of the μTOSCA toolchain, and therefore we can say that this study is fundamental as it covers its aspects of interest.

Three different papers were presented with the same goal of proposing a catalog or taxonomy of possible problems (viz., smells, issues, or bad practice) for microservice architectures, applying different strategies to achieve the result, such as interviews [29] [33], surveys [29] [33], literature reviews [19] [29], or systematic analysis of GitHub issues [33]. Although the source of the information was different, applied collection techniques were not similar between papers, and the types of the problem considered were different, the authors achieved similar results. Whether they are architectural problems, bad practices, or issues, they are still all problems related to similar errors in the application. To give an example, the presence of hardcoded IP is a problem that is presented by all three papers in different forms, just as it can also be for the presence of databases used by more than one microservice, etc. Thus, it is possible to point out the goodness and robustness of the results obtained by the authors because, although they started from different data collections and sources and considered different aspects related to the system's problems, they still obtained similar results that highlighted the same problems in different forms.

2.2 Detecting and resolving smells in microservices applications

Multiple solutions have been proposed to automate the detection of smells in microservices applications. For instance, Pigazzini et al. [21] proposed an extension of an existing architectural-smell detection tool, viz., Arcan [4], by implementing the detection of three architectural smells for microservices, namely *Shared Persistence*, *Hard-Coded Endpoints*, and *Cyclic Dependency*. Arcan represents the system as a call graph, built by analyzing configuration files and source code, that is then used for detecting the *Cyclic Dependency* smell, while the other two smells are detected directly by analyzing the microservices source code.

Bacchiega et al. [2] presented the tool Aroma (*Automatic Recovery of Microservices Architecture*), which can perform microservice smell detection. Aroma exploits dynamic analysis to build a graph representing the system under analysis and then launches its detector for searching the presence of three architectural smells for microservices, viz., *No API Gateway*, *Shared Persistence*, and *Cyclic Dependency.*

Another tool that supports developers in performing smells detection in microservice systems is MSANose, presented by Walker et al. [31]. The tool uses a static analysis approach for carrying out smell detection, by producing graphs for representing the application under analysis, built starting from byte-code or source code, and then by analyzing them for performing the smells detection. MSANode is able to detect up to eleven bad smells in an application, among those elicited by Taibi et al. [29], but it can also identify in the system under analysis other types of issues, such as faults, bugs, performance problems, service decomposition problems. It is interesting to point out that among the smells recognized defined in Taibi et al. [29] and detected by MSANose are also *Shared Persistence*, *No API Gateway*, and *Hard-Coded Endpoints*, which are all also defined by Neri et al. [19] and therefore also considered in this thesis.

Hübener et al. [11] instead enable visualizing the architecture of a microservice system and detecting anti-patterns therein. This is done dynamically by analyzing a running system by using Apache Kafka[2] and by collecting invocations of microservices in the system, for then building a graph that describes the architecture of the application as microservices and their interactions. Hübener et al. [11] then compute various metrics for analyzing the reconstructed graph and, based on the computed values, they can detect the presence of *Mega-services*, *Nano-services*, *Bottleneck-services*, *Ambiguous-services*, and *Cycle-dependent-services.*

Udara et al. [6] developed a tool for finding anti-patterns, able to detect *The Knot*, *Nano Service*, *Service Chain*, *Bottleneck Service*, and *Cyclic Dependency.* The key idea of this project is to represent the entire system as a graph, generated through dynamic analysis using traces, and then use graph algorithms to compute metrics that can suggest the presence of microservice anti-patterns. At the end of the execution, all metrics calculated are outlined through bar charts and, if the value is greater than a defined threshold, then the developer can see that the anti-pattern is present for one or more microservices.

Al Maruf et al. [17] provide another approach for reconstructing microservices architecture and detecting anti-patterns, which consists in analyzing service-mesh logs, which contain all information about communication between components of the system. In this way, it is possible to build a Service Dependency Graph (SDG) representing the system, which is then analyzed for performing anti-pattern detection. By analyzing the SDG, it is possible to make the detection of six anti-patterns [23] [3], namely *Absolute Importance of the Service*, *Absolute Dependence of the Service*, *Cyclic Dependency*, *Bottleneck*

[2]`https://kafka.apache.org/`

Service, *Shared Persistency*, and *API Versioning*. Following these concepts, authors also presented the prototype tool *istio-log-parser*, able to analyze Istio service-mesh logs, calculate metrics, and detect smells and anti-patterns.

Some of the tools presented above differ from the ones we propose as they focus more on detecting anti-patterns than architectural smells, even if some of them can recognize some of the architectural smells presented by Neri et al. [19]. In particular, the *No API Gateway* smell is detected by two of the proposed tools [2][31], while two other tools [21][31] can detect *Endpoint-based Service Interactions* (even if called differently, viz., *Hard-Coded Endpoints*). As for *Shared Persistence*, in the work described above it is presented both as an architectural smell and an anti-pattern and identified by 4 different tools [2][17][21][31]. Although the presented tools share some of the smells considered in this thesis [19], all of them differ from the one presented in this thesis as they focus *only* on smell detection, while we instead aim at proposing a solution that can enable identifying *and* resolving such architectural smells.

In this perspective, the solution presented by Soldani et al. [27] is a step closer to the one we are looking for. They indeed introduce the so-called μTOSCA toolchain, which can mine the architecture of a microservice system, perform smell detection of four smells among those defined by Neri et al. [19], and suggest available refactoring for smells found. Every tool of the chain works with a representation of the system modeled using μTOSCA, a customized version of the OASIS TOSCA standard, designed for representing microservices systems as a topology graph. The first tool of the chain is μMiner, responsible to analyze the application and produce in the output the description of the system modeled through μTOSCA, combining both static and dynamic analysis to perform architecture mining. The μTOSCA description of the system can then be passed as input to μFreshener, the second tool of the chain, capable of analyzing the topology graph representing the model and detecting smells present in the system, and applying refactoring techniques on the μTOSCA model.

In this thesis, we enhance the capabilities of the μTOSCA toolchain, by introducing an alternative tool to μFreshener, which can not only detect and resolve smells on the μTOSCA topology graph but also semi-automate the implementation of the proposed refactorings by automatically generating the necessary updates to the application deployment in Kubernetes. Also, the proposed tool will cover the architectural smell related to a key design principle that was not yet covered by the μTOSCA toolchain, viz., the *independent deployability* of microservices.

3 Background

This chapter provides the necessary background on the technologies used in this thesis, namely, containers (Section 3.1) and Kubernetes (Section 3.2). This chapter also recaps how to model microservices applications with μTOSCA (Section 3.3), and the architectural smells from Neri et al. [19] that are considered in this thesis (Section 3.4). Finally presents the μTOSCA toolchain (Section 3.5) introduced by Soldani et al. [27].

3.1 Container

A container is a lightweight, standalone package that encapsulates a complete runtime environment, including an application and its dependencies - so libraries, binaries, and any additional configuration files [30].

Containers can be used in any application, but they are frequently associated with microservice systems. In these applications, each microservice can be developed in a separate container, and later be executed independently from the others. In addition, each container has its codebase, so it can be assigned to a team and developed separately from the others, a key principle of microservice development.

A single container can be run by using some tools, i.e. Docker,[3] but for running an entire application this may not be enough, because managing a system deployed in that way can be in some cases difficult and some feature can miss. In the production environment, a microservice application is often run using a container orchestration platform, such as Kubernetes. These platforms allow developers to deploy a complete application by running several containers and managing them, offering also different additional capabilities to the system, i.e., service discovery between microservices.

3.2 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized applications [14]. By packaging each microservice in a different container, and then using Kubernetes for deploying the entire system, it is possible to run and manage microservice applications.

When an application is deployed using Kubernetes, a cluster is obtained, that consists of a set of worker nodes and a control plane, which respectively run the container of the application and manage worker nodes and the whole cluster.

By defining Kubernetes objects (also called resources), it is possible to configure the desired state of the cluster. The control plane, during the execution, always checks if the desired state is reached and, in case the actual state is different from that, operates for reaching it.

[3]https://docs.docker.com/

Kubernetes allows the deployment of multiple resources, that cover various aspects and offer additional functionalities to the application, but in this chapter, only the ones relevant to this work are presented. In addition to Kubernetes native resources, Istio service mesh resources are introduced, which allows additional traffic management features to be added to the cluster.

3.2.1 Pod

A Pod is the smallest unit of computing that can be deployed in a Kubernetes cluster [16], which can be described as an object defining one or more containers running with shared storage, network resources, and configuration for the execution of containers. A Pod in Kubernetes is ephemeral, so everything related to its execution is "lost" when the Pod is deleted. When a Pod is deployed it is scheduled on a node and an IP address is assigned to it, but during its execution, it can fail and then restart, be scheduled on another node, etc., so the IP address assigned can change.

A Pod is normally not reachable from messages coming from outside the cluster, but two properties allow external clients to reach it directly. The first one is *hostNetwork*, which indicates that the Pod shares the same network as the node, so its ports are exposed directly to the host node address. The second is *hostPort*, which is used for exposing a single port on the host node. Both proprieties are important for this study because are used during the extension and refactoring phases.

Typically, pods are not deployed directly within the cluster but are managed by other deployed resources offering additional capabilities. For instance, ReplicaSet, Deployment, and StatefulSet resources can provide Pod replication, declarative updates, and management of pods with persistent storage, respectively. In this thesis, the term Pod is used in a broad sense to refer to both the Pod resource and any of the resources mentioned earlier that define or manage it.

3.2.2 Service and Ingress

A Kubernetes Service [16] can be described as an abstraction defining a set of logically connected pods and a policy for accessing them. It is possible to state that a Service acts as a message router in the cluster for a set of pods, providing them service discovery and load balancing capabilities. Using Kubernetes services, when a Pod P_A needs to access functionalities offered by P_B, does not use directly the P_B IP address but sends a message to the Kubernetes Service exposing it by using its hostname and port, allowing service discovery. The Service receives the request on a port and routes it to one of the instances manage that can handle the received message at that moment, enabling load balancing.

For making a Kubernetes Service to expose a certain Pod, two conditions must be satisfied. A Pod can be enriched with one or more labels, and a Service for exposing a Pod has to define in its selector field at least one of the labels defined by the Pod. Each Kubernetes Service exposes ports, and the Service must also define a port that routes traffic to one of those exposed by the Pod. If a Service *my-service* defines port 80 which redirects traffic to port 8080 of Pod *my-pod*, then for contacting *my-pod* it is necessary to send a request to *my-service:80*.

A Service can also be exposed outside the cluster, and this is possible by using types. A Service can be of four different types, but only two types are meant for this study, ClusterIP and NodePort, because are the ones used for refactoring activities. ClusterIP is the default type and makes it only reachable within the cluster while, on the other hand, a NodePort Service exposes each of its ports on the host node.

For exposing a Service to the outside, an Ingress resource can also be used, which is an API object that manages external access to the services in a cluster [16], exposing HTTP and HTTPS routes from outside the cluster to Kubernetes services within it. For having an Ingress working for the cluster, an *Ingress controller* running in the cluster is required, such as Nginx.[4] After running a controller, Ingress resources can be deployed, and each resource defines one or more rules that create the routes exposed from the outside to services.

3.2.3 Istio

Istio is an open source *service mesh* that layers transparently into existing distributed applications. In the microservice context, a service mesh is a dedicated infrastructure layer of an application, that allows to transparently add capabilities like observability, traffic management, and security, without modifying microservices for adding code [13].

An Istio service mesh is composed of two main components, namely the data plane and the control plane. The data plane is a set of Envoy[5] proxies, where each of them is deployed as a sidecar for every Kubernetes Service defined in the cluster, and it controls network communication between microservices. The control plane is the object which manages the proxies, so take the configuration and dynamically inject and configure proxies in the cluster. For configuring them, Istio provides some resources, defined through Kubernetes API, that can be deployed in the cluster. The architecture of the service mesh is outlined in Figure 3.1.

Among the capabilities that Istio can offer, the most interesting for this thesis is traffic management, and its related resources *VirtualService*, *DestinationRule*, and *Gateway*. The first two resources are the key building block of Istio's traffic routing functionality. A *VirtualService* allows the configuration of how requests are routed to a service, while *DestinationRule* configures what happens to traffic directed to that destination. With the *VirtualService* resource, a timeout can be defined for the communication, in a way that after a specified amount of time without a response from the Pod, the proxy responds with an error message. On the other hand, using a *DestinationRule* resource can configure a circuit breaker for communication. A circuit breaker is a component that can be in two different states, opened or closed. In the opened state, it forwards all received messages to

[4]`https://kubernetes.github.io/ingress-nginx/`

[5]`https://www.envoyproxy.io/`

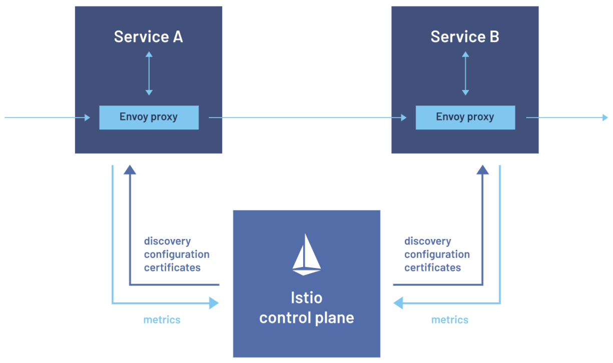


Figure 3.1: Representation of Istio architecture [13].

the microservice, monitoring their execution and recording failures. When failures exceed a selected threshold, the circuit breaker changes state to open, and the proxy responds with a message error to each request received. The timeout can also be defined using a *DestinationRule* resource, but the most common way for implementing them is using a *VirtualService*, as explained earlier. The third interesting resource is the *Gateway*, which is similar to the presented KubernetesIngress.

We introduced these resources as they are fundamental both for model extension and smell refactoring, to search or define timeouts, circuit breakers, and gateways for the application.

3.3 Modeling microservices architetures

μTOSCA was introduced by Soldani et al. [27] for enabling the modeling of the architecture of microservices applications, and then process it using the μTOSCA toolchain. It builds on top of the *Topology and Orchestration Specification for Cloud Applications* (Section 2.3.1), also called TOSCA. Essentially, μTOSCA (Section 2.3.2) provides a type system to specify microservice-based architectures as typed topology graphs in TOSCA. Intuitively, the nodes in a TOSCA topology graph model the services, integration components (e.g., API gateways, load balancers, or message queues), and databases forming a microservice-based application, while the arcs indicate the runtime interactions occurring among them.

3.3.1 TOSCA

TOSCA [20] is an OASIS standard that allows specifying the architecture of a cloud application by using a YAML-based modeling language. In TOSCA cloud applications are described using a *service template*, which is a composition of a *topology template* and *types* needed to build it. The *topology template* is a representation of a topology graph, that models a cloud system as components (nodes) and their interactions (edges). The TOSCA metamodel is outlined in Figure 3.2.

In the topology graph, an application-specific instance of a component is defined us-

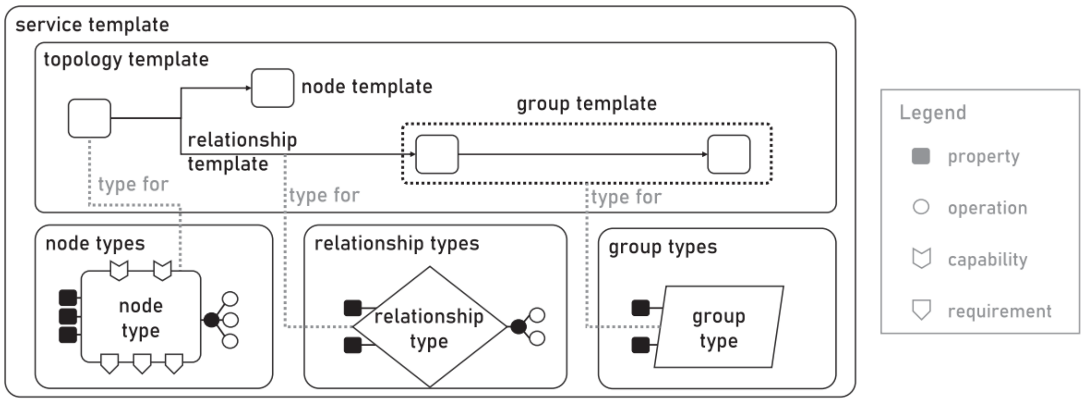


Figure 3.2: Representation of the TOSCA metamodel [27]

ing a *node_ template*, represented by a node in the graph. Each node has assigned a *node_ type*, that represents the kind of component it represents, and describes its observable properties, management operations, requirements for instantiating it, and capabilities offered to other components. Each template is present only one time in the graph, while the same type can be assigned to several nodes. For better understanding, an instance of a microservice running in the application it is represented using a *node_ template* with "microservice" *node_ type*.

Regarding relationships, an edge between two nodes of the graph is modeled using a *relationship_ template*, representing a specific interaction. Each relationship has a *relationship_ type*, defining its type and indicating operations for managing it and its observing properties. For example, a relationship of *deployedOn* type can indicate that one component is deployed on another.

In TOSCA nodes can be also grouped, by using the *group_ template*, an instance of logically grouped nodes, which are grouped for the reason specified by the *group_ type* assigned.

Is worth pointing out that TOSCA types support inheritance, so it is possible to define hierarchies of types. For example, there can be a *node_ type* datastore, indicating the general characteristic of a data store in the application, and then this could be specialized with NoSQL Datastore type, adding attributes belonging only to NoSQL datastores.

3.3.2 μTOSCA

μTOSCA [27] is an OASIS TOSCA customization for modeling microservice-based architecture applications as a TOSCA topology graph, which defines its hierarchy of types outlined by Figure 3.3. The Service type represents a component that executes some business logic, so a microservice of the application. With *CommunicationPattern* type it is possible modeling a component that breaks down the communication between a sender and a receiver, representing communication patterns defined by Hohpe and Wolf [9]. Two types in the hierarchy are a specialization of this type, representing MessageRouter and MessageBroker communication pattern. The first describes a component able to perform

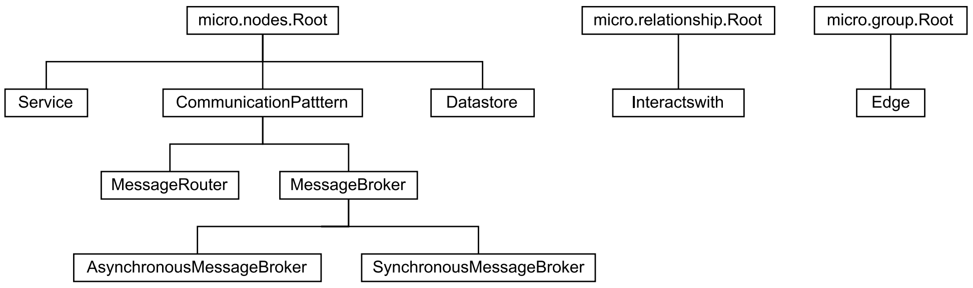


Figure 3.3: Hierarchy of types defined by the μTOSCA model [27]

some kind of message routing activities in the application. A MessageBroker, on the other hand, implements the Message Broker pattern, which is further specialized into *AsynchronousMessageBroker* and *SynchronousMessageBroker*. The last *node_ type* present in the hierarchy is Datastore, a direct child of the root node, representing a component able to store and make accessible data in a microservice application. Nodes can be clustered in groups, and in particular μTOSCA defines the Edge group type, which clusters all nodes accessible by external clients from outside the application.

μTOSCA currently features only one type of relationship that can be defined in a topology graph, namely the InteractsWith relationship type, which enables specifying a generic runtime interaction between a source and a target component. This relationship can also be enriched with three different observable properties, describing the type of communication between connected components. The first is *dynamic_ discovery*, which represents that in an interaction from m_A to m_B, m_A uses a service discovery mechanism for finding the location of m_B. Another property available is *timeout*, indicating that a timeout is defined for the interaction. The last property available is *circuit_ breaker*, which indicates that the circuit breaker pattern [9] is applied to the represented interaction.

Figure 3.4 shows an example of μTOSCA topology graph, specifying the architecture of an e-commerce application, which includes three different microservices, namely *orders*, *shipping*, and *warehouse*. The entry point of the application is represented by the

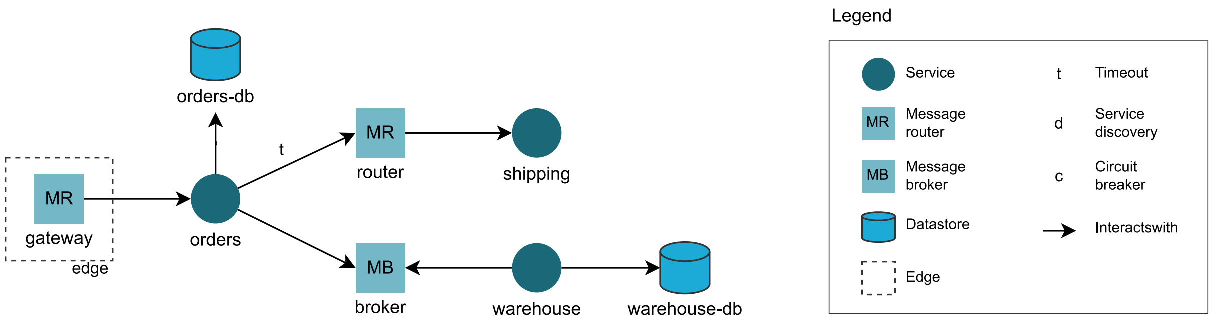


Figure 3.4: Example application modelled using μTOSCA

MessageRouter *gateway*, which is a member of the Edge group for modeling the fact that it is accessible by external clients. The gateway routes messages to the *orders* microservice, which interacts with three different components, its Datastore *orders-db*, a MessageBroker, and a MessageRouter. The *router* exposes the microservice *shipping* and a timeout is also defined in the interaction from orders to the router. The *broker* is instead used for communicating with *warehouse* microservice, which in turn interacts with the datastore *warehouse-db*.

3.4 Architectural Smells for Microservices

This section introduces a key concept for this thesis, the *architectural smells for microservices*. These smells indicate violations of microservice design principles introduced by Neri et al. [19] and are the main focus of the tools presented in the following chapters. This thesis will use various names to refer to them, namely *architectural smells for microservices*, *architectural smells*, *smells*, or *microservice smells*. In detail, this section discusses five smells that violate four design principles, and each pair of violated smell-principle is enumerated in Table 3.1.

Design Principle	**Architectural Smell**
Decentralisation	Shared Persistence
Horizontal Scalability	No API Gateway
Horizontal Scalability	Endpoint-based Service Interaction
Independent Deployability	Multiple Services in One Container
Isolation of Failures	Wobbly Service Interaction

Table 3.1: List of microservices design principle with smells that violate them

This section follows with a detailed explanation of all the smells considered, with their definition and the explanation of the problems introduces in the system.

3.4.1 Endpoint-based Service Interactions

An Endpoint-based Service Interaction smells occurs between two microservices m_A and m_B when m_A directly invokes an instance of m_B for using its functionalities, without passing through any service discovery mechanism or component [19]. In a microservices system, an endpoint-based interaction can so be present when m_A, for consuming functionalities offered by m_B, has the IP address hard-coded in its code-base and uses this address for sending the request.

This situation violates the *Horizontal Scalability* principle of microservices, which advocates that all replicas of a microservice m should be reachable by the microservices invoking it [19], and this leads to problems related to microservice scaling described in Figure 3.5. When m_A invokes directly m_B and this is scaled by adding some new instances, messages from m_A will always reach only the "original" invoked instance of m_B, resulting in having running instances not used and causing so a waste of resources. In this case, one instance A_1 of the microservice m_A communicates directly with one instance B_1 of m_B, for example by knowing its IP address. The microservice m_B needs

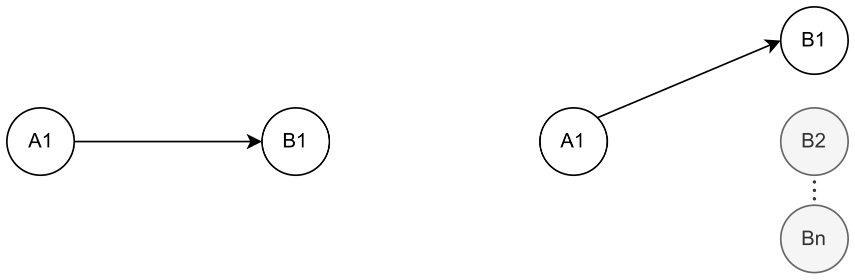


Figure 3.5: Representation of the problem caused by the smell on horizontal scalability of the system

then to be scaled for increasing its instances, because having only one running is no longer enough for handling the load of requests, so new instances $B_2..B_N$ are deployed. Since A_1 contacts directly B_1 using its IP address, newly deployed instances $B_2..B_N$ will be never contacted, causing the waste of resources.

3.4.2 No API Gateway

The *No API Gateway* smell arises in the application when a microservice instance is directly accessible by the client from outside the system, so, when messages from external clients reach directly microservices instances instead of passing through a gateway [19]. This smell is similar to *Endpoint-based Service Interactions* because, in both cases, a microservice instance is directly accessed. The only difference between them is that while *Endpoint-based Service Interactions* smells focus on interactions between components inside the application, *No API Gateway* smell is related to messages coming from outside the application, thus, it occurs at the edge of the application instead of in communication between microservices.

Both smells violate the same *Horizontal Scalability* principle and, since the problem is similar, even the issues encountered are the same as discussed before, detailed in Figure 3.6 for this smell. In the example, a user directly invokes one instance of a microservice

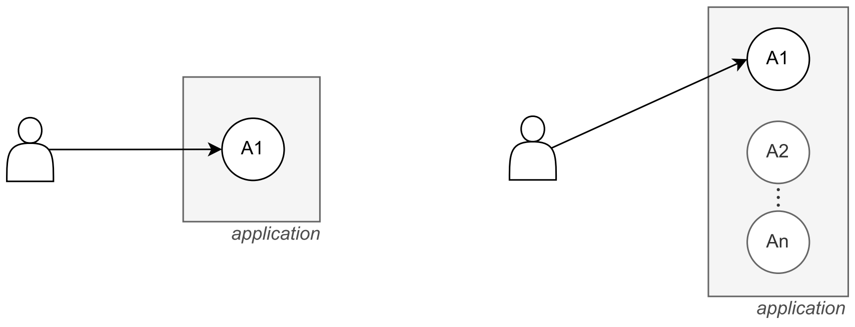


Figure 3.6: Representation of problems caused by the presence of the No Api Gateway smell in the system

A_1, i.e., by knowing its IP address. When A_1 is not able anymore to handle all its requests, there is the necessity of scaling it out, so new instances $A_2..A_n$ are deployed.

Even if the new instances are running and available, the user continues using only the microservice A_1, because it contacts this instance directly and doesn't even know that others are available. Here the problem related to *Horizontal Scalability* occurs, which causes a waste of resources.

3.4.3 Multiple Services in One Pod

The *Multiple Services in One Pod* smell occurs when a Pod executes more than two microservices [19], so if it deploys two or more container that runs business logic. Considering only containers that execute business logic is very important, since in Kubernetes can be applied several patterns [15] that deploy two or more containers within the same Pod, and this cannot be considered as smell.

Having more than one microservice defined by the same pod violates the *Independent Deployability* principle [19], which advocates that in a microservice-based application, each microservice should be operationally independent of the others, so it should be deployed and undeployed independently. Containers provided ad ideal tool for adhering to this principle, because with this technology is possible to define small packages that contain everything needed for executing the software, so code, dependencies, resources, etc. Each container can be run independently from the others and even easily scaled by running multiple instances of it. Considering Kubernetes, if two or more containers are defined by the same Pod, they are not independent of each other, violating the *Independent Deployability* principle. This can cause some problems during the execution of the system, because if two microservices are not independent they must be executed and stopped together, restarting the Pod for a failure in one of them causes the restart of both containers, it is not possible to scale them separately, etc.

The problem is outlined in Figure 3.7, where is represented by a Pod P_1 that executes C_1 and C_2, with C_1 that is used by another client while C_2 is not used by anyone. When

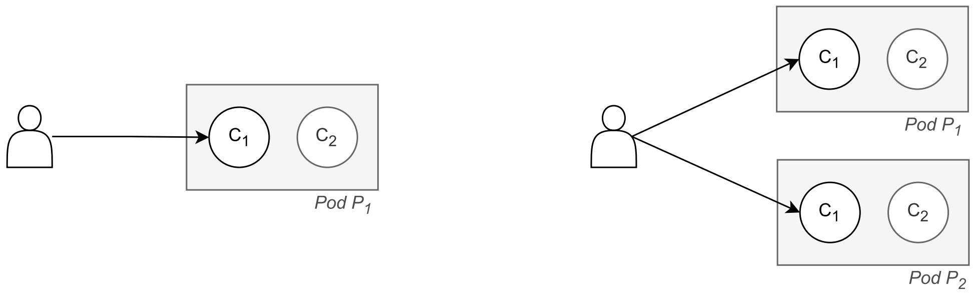


Figure 3.7: Representation of the independent deployability problem

another instance of C_1 is required, for scaling it out must be deployed the whole Pod P_2, including both C_1 and C_2. The client starts to use C_1 from both containers, but the number of unused instances rises from 1 to 2, because both C_2 containers are not used by anyone. In this case, it was not necessary to ass another C_2 instance, but it could not be done otherwise because the two microservices are not independent.

Multiple Services in One Container This section introduced the smell MS but Neri et al. [19] defined the smell *Multiple Smell in One Container*, which defines the smell when two microservices are packaged in the same container and thus have no independent deployment.

The situation just described could not be detected by this tool, since the analysis performed does not include a check of the contents of the container. However, it was possible to apply the same principle to the deployment Kubernetes to detect a similar smell. In Kubernetes, the smallest deployable unit is the Pod, and typically the "one-container-per-pod" model is applied, where a Pod is the wrapper of a single container (which is assumed to define only one microservice) [16]. If a Pod defines more than one container, this may indicate that multiple microservices are being deployed together in that Pod. In this case, a similar problem as above is present, where the two microservices are not deployed independently, and therefore the detection of this smell was considered as an "alternative" to that defined by Neri et al. [19].

3.4.4 Wobbly Service Interaction

We can define an interaction between two microservices m_A and m_B (with m_A that invokes m_B) as "wobbly" when a failure of the invoked microservice m_B results in triggering an error also in the microservice m_A [19]. This problem is present when the microservice m_A consumes directly functionalities offered by m_B, and the invoker microservice does not handle the possibility of m_B failure. In case of m_B failure, even m_A fails, and this can cause cascade defeats among all microservices of the application having a big impact on the running application.

The isolation of failure principle advocates that a microservices application should be designed in a way that each microservice can tolerate failures of any invocation to the microservice it depends on [19]. This principle is violated by this smell because it lets errors propagate among all components of the system, and so the presence of this smell causes problems related to microservice availability. This issue is highlighted in Figure 3.8, where two microservices m_A and m_B communicate directly with a wobbly interaction. During its execution, m_B fails, i.e., because of an unhandled exception, and then m_A

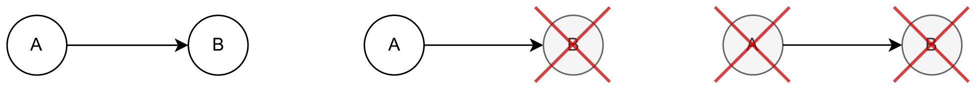


Figure 3.8: The image points out the problem that can arise in presence of wobbly interactions between services

sends a request to it. Microservice m_A then awaits m_B response, but will potentially be able to remain in an infinite wait since m_B has failed, so the error is propagated from m_B to m_A.

3.4.5 Shared Persistence

In a microservices system, data have to be split across multiple small databases instead of having a big centralized datastore [19], in a way that each microservice uses a dedicated database containing only its data. There must therefore be a 1:1 mapping between a microservice and its related database, following the *Decentralisation principle*, which advocates that the business logic of an application should be fully decentralized and distributed among its microservices, each of which should own its domain logic [19]. When in an application more than one microservice access and manages the same datastore, this principle gets violated, causing the *Shared Persistence* smell.

3.5 The μTOSCA Toolchain

The μTOSCA toolchain [27] is a set of tools presented by the Service-Oriented Cloud Computing (SOCC)[6] research group at the University of Pisa. These tools operate in the context of microservice-based applications and enable mining of their architecture, detection of architectural smells for microservices, and application of refactoring on the μTOSCA model describing the application. The μTOSCA toolchain (Figure 3.9) includes the tools μMiner and μTOM, enabling mining, and μFreshener, enabling smell detection and refactoring of the μTOSCA model. For better interoperability among the tools in the chain, a shared data model is valuable. Therefore, every tool in the chain operates on a μTOSCA model that defines the software being analyzed.

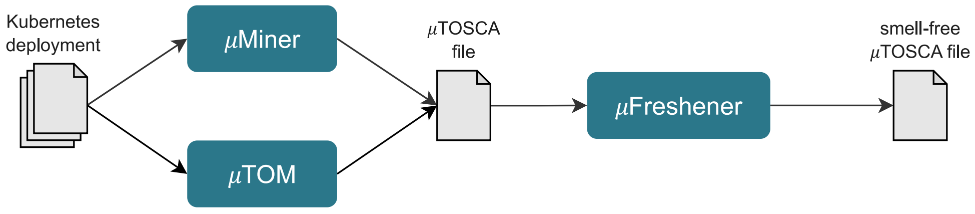


Figure 3.9: Representation of the μTOSCA toolchain

This section describes the toolchain by presenting the functionalities of the three tools mentioned. In particular, it describes the tools μMiner and μTOM for architectural mining (Section 3.5.1), analyzes the μFreshener tool presenting its features and architecture (Section 3.5.2), and details detection strategies implemented for smell detection (Section 3.5.3).

3.5.1 Architecture mining: μMiner & μTOM

Manually modeling an entire application's architecture using μTOSCA models can be a time-consuming, tedious, and error-prone task. To help developers define a μTOSCA model that represents a system, a simple and quick-to-use solution is needed, that is

[6]`https://di-unipi-socc.github.io/`

capable of generating the topology graph automatically and with as few errors as possible. To address this problem, the SOCC research group has introduced two different tools, μMiner [27] and *μTOSCA Offline Miner* (μTOM) [25], which use different strategies to retrieve the architecture of microservice-based applications.

μMiner performs mining by combining both static and dynamic analysis, and thus requires the target microservice-based application to run in a specially configured test environment. The analysis performed consists of three different steps. The first is the *static mining* one, which starts from Kubernetes deployment files and creates corresponding nodes in the μTOSCA topology graph. The second step, *dynamic step*, consists in applying dynamic mining techniques to dynamically retrieve information about components interactions. Finally, a *refining step* is performed to correct the generated node starting from the information collected in the previous step.

The approach taken by μTOM, on the other hand, differs in that it can only generate the μTOSCA model through static analysis. This tool so does not require the analyzed application to be running, but it only needs to be configured to use Istio[7] and Kiali,[8] two tools for proxying services and monitoring their interactions. To perform the mining, the tool simultaneously processes the deployment of Kubernetes, the Istio-based proxying of its services and the graph generated by Kiali, which contains the interactions between the components to determine the architecture of the analyzed system.

Although the strategies used are different, in both cases the tools are able to start from the Kubernetes files describing the system deployment and generate its representation through the μTOSCA modeling language. This generated μTOSCA model is then used by subsequent tool in the toolchain, μFreshener, to perform smell detection and model refactoring.

3.5.2 Smell detection: μFreshener

μFreshener [27] is the tool within the μTOSCA toolchain that performs smell detection and refactoring by analyzing the μTOSCA system definition. Regarding detection, the tool recognizes four different architectural smells for microservices [19], namely *Endpoint-based Service Interactions*, *No API Gateway*, *Shared Persistence*, and *Wobbly Service Interaction*. In addition to performing smell detection, this tool can perform refactoring on the μTOSCA model by applying several solving techniques [19]. This gives the developer the "smell-free" μTOSCA description of the system but smell are not really solved in the application, so he has to modify the system for removing smells, using the produced model as reference.

This tool can be divided into two main components, μFreshener and μFreshener-core. All the main functionalities needed for the tool are implemented by μFreshener-core, which is a Python project that defines functionalities such as import and export of the μTOSCA model, smell detection, etc. μFreshener, on the other hand, is a client

[7]https://istio.io/
[8]https://kiali.io/

server project that functions mainly as a wrapper for the functionality implemented by μFreshener-core, and provides a GUI for using the system.

μFreshener-core is divided into several modules, but the most significant for this work are Importer, Model, Exporter, and Analyzer. The Importer deals with parsing the μTOSCA definition from files, and representing the system through the data structures defined by the Model. At the end of the tool execution, the Exporter is able to save the Model data structures to disk. The last module presented is the Analyzer, which implements the strategies used for smell detection in different detectors. This module is particularly significant for this work for two main reasons. Firstly, we modified it to introduce the detection of the new smell *Multiple Services in One Pod*. Secondly, the μFreshener++ tool presented in this thesis strictly uses this module to reuse detection functionalities and avoid reimplementing them from scratch.

3.5.3 μFreshener detection strategies

The purpose of this section is to present the detection strategies used by μFreshener to identify the four architectural smells for microservices that are considered in this study, namely *Endpoint-based Service Interactions*, *No API Gateway*, *Wobbly Service Interaction*, and *Shared Persistence* [19].

An *Endpoint-based Service Interactions* smell occurs between two microservices m_A and m_B when m_A directly invokes an instance of m_B for using its functionalities, without passing through any service discovery mechanism or component [19]. In the μTOSCA model, interactions between two microservices are modeled through the *InteractsWith* relation, and direct interaction between two service nodes indicates that represented microservices are able to interact directly. Among possible proprieties, an interaction can be enriched with *service_ discovery*, which represents the uses of a service discovery mechanism in the interaction in order to find the location of the invoked service. The presence of an *InteractsWith* relation between two service nodes not enriched with the service_discovery property indicates that the sender contacts directly the receiver microservice, i.e., by knowing directly his location (IP address). This case represents an endpoint-based interaction, which is the smell μFreshener had to find in the μTOSCA model. To detect these kinds of interactions, the Analyser starts by checking all relations present in the model, and in particular, only *InteractsWith* edges between two Service nodes are considered. If the relation has not had the property *service_ discovery* set, then the smell is detected. Figure 3.10 represents the smell present in the μTOSCA model, where is present an interaction from a Service node to another Service not enriched with the *service_ discovery* property.

The *No Api Gateway* smell arises in the application when a microservice instance is directly accessible by the client from outside the system, so, when messages from external clients reach directly microservices instances instead of passing through a gateway [19]. Considering the μTOSCA model, a node member of the edge group indicates that the

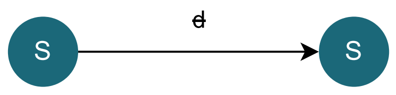


Figure 3.10: Representation of the *Endpoint-based Service Interactions* smell in the μTOSCA topology graph

represented component is directly accessible by external clients, and Gateways or equivalent components are represented through a message router node member of the edge. If the node in the edge, however, is not a message router but for example a service, this may indicate that the microservice m_A represented by this node is directly reachable by external clients, without passing through any other component, i.e., a gateway. Finding these nodes is the strategy of μFreshener for detecting the presence of this smell. In fact, the tool performs No Api Gateway detection based on the analysis of the node members of the Edge group, controlling the type of each node belonging to the group and, if the node is a service or a message broker, the smell is detected. Figure 3.11 outline this situation, showing the two cases when the smell is found in the model.

MB
edge
S
edge

Figure 3.11: Representation of the *No API Gateway* smell in the μTOSCA topology graph

An interaction between two microservices m_A and m_B (with m_A that invokes m_B) is "wobbly" when a failure of the invoked microservice m_B results in triggering an error also in the microservice m_A [19]. Considering μTOSCA, an interaction between two components m_A and m_B is modeled through *InteractsWith* relationship, which can be enriched with timeout or circuit breaker properties among other possibilities. An interaction enriched with *timeout* allows fault isolation because, in case of an m_B failure, m_A receives an error when timeout ends and is able to handle it easily, avoiding the propagation of error in the entire system. Circuit_breaker property, on the other hand, indicates the presence of a circuit breaker in the relation which handles communication between m_A and m_B. So, if m_A tries to send a message to m_B but this is too overloaded or failed, the circuit breaker is opened and responds with an error message, that is then managed by m_A. Even in this case, the presence of the circuit breaker allows fault isolation. In both cases, are defined mechanism for preventing the error to be propagated. Considering that, we can affirm that errors are handled in an interactions enriched with *timeout* or *circuit_ breaker* properties, and this is the key idea of the analyzer for searching this smell in the system. For this detection, only interactions between two Service nodes or from a Service to a MessageBroker are considered. The properties defined by these interactions are analyzed to check if the edge is enriched with at least one of *timeout* or *circuit_ breaker* properties. If the interaction does not define any of these, it means that errors are not handled and so the smell *Wobbly Service Interaction* is detected. The situation described is outlined in Figure 3.12, which shows the two cases where the smell is found.

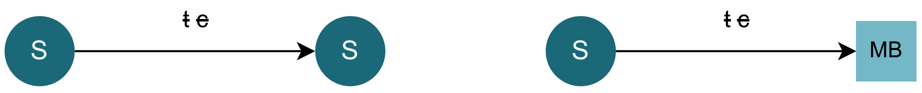


Figure 3.12: Representation of the *Wobbly Service Interaction* smell in the μTOSCA topology graph

Finally, the Shared Persistent smell arises in the application when more than one microservice share the same datastore system [19]. A database is represented in μTOSCA using the datastore node and, for indicating that a microservice m_A access the database d_B, an InteractsWith edge is added to the graph from service node m_A to d_B. Since an incoming interaction to the datastore represents a microservice that uses it, the number of incoming InteractsWith relations from service nodes is equal to the number of microservices that uses the database represented. In order to not violate the principle, a datastore node should have exactly one interaction incoming from a service, representing so that only one microservices accesses it. The μFreshener++ tool uses this idea for finding the smell, by searching datastore nodes used by more than one service counting the number of incoming Interactswith relations, and detecting the *Shared Persistence* smell where the count is greater than one.

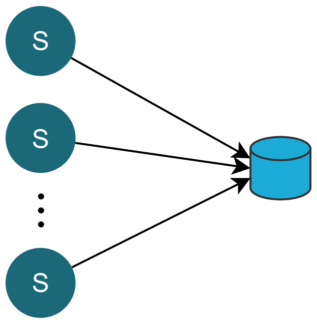


Figure 3.13: Representation of the *Wobbly Service Interaction* smell in the μTOSCA topology graph

3.6 Summary

The contents of this chapter offer the necessary background to comprehend this thesis. The first concepts we introduced, namely containers and the Kubernetes orchestrator, serve as building blocks for microservice-based applications. In regards to Kubernetes, all resources considered in this thesis have been described, including those defined by the Istio service mesh. Next, we described the modeling language μTOSCA [19] for describing microservice architectures. This language is used by all the tools of the μTOSCA toolchain and is essential for comprehending the entire thesis. Finally, we introduced the μTOSCA toolchain, describing all the tools that make it up and the detection strategies adopted for individuating smells. This step is crucial, as this thesis presents an extension of the μTOSCA toolchain that involves modifying the previously presented tools and adding a new tool (μFreshener++) to the toolchain.

4 μFreshener++: A Bird-Eye View

One of the main advantages of developing a toolchain is that it can be extended easily by introducing a new tool that provides additional functionality. As outlined in earlier chapters, the μTOSCA toolchain delivers capabilities for mining architectures, performing smell detection, and refactoring μTOSCA models. To expand the μTOSCA toolchain, we developed the μFreshener++ tool, which includes two new features: correcting and enhancing μTOSCA models (extension feature), and automatically refactoring to remove architectural smells (refactoring feature).

The introduction of the new tool implies an update in the μTOSCA toolchain, and its new version is illustrated in Figure 4.1. The μFreshener++ tool takes as input the

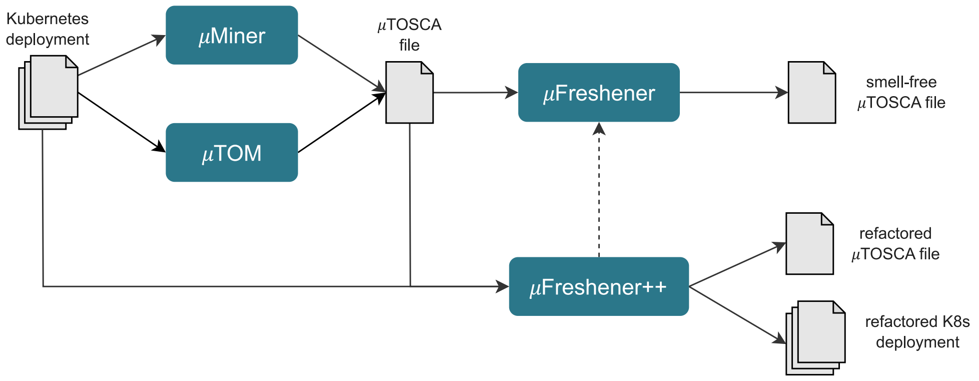


Figure 4.1: The new μTOSCA toolchain after the introduction of μFreshener++, where the dashed edge indicates the use of μFreshener functionalities

μTOSCA model that describes the system architecture and its corresponding Kubernetes deployment, which are processed together to produce as output the refactored Kubernetes deployment and the refactored μTOSCA model. The μTOSCA model can be created either manually or by using the mining tools available in the μTOSCA toolchain, such as μMiner and μTOM. To identify the smells present in the system, μFreshener++ uses μFreshener to perform smell detection on the μTOSCA model. By leveraging μFreshener, it is not necessary to develop all of the detection strategies from scratch since the required functionalities can be reused.

This chapter presents the main functionalities of μFreshener++ and its workflow (Section 4.1), and describes the design of μFreshener++ detailing its architecture (Section 4.2).

4.1 Workflow

The main functionalities of μFreshener++ are twofold: extending the μTOSCA model and refactoring architectural smells. The extension feature involves analyzing the μTOSCA description of a system and its Kubernetes deployment files simultaneously to modify the μTOSCA model, correcting errors and adding components that are not currently represented. On the other hand, refactoring consists of applying different strategies to correct the smells found in the system by directly modifying the Kubernetes deployment.

The workflow of the μFreshener++ tool is illustrated in Figure 4.2, which shows its main modules and the data flow between them. The tool consists of six main modules, five of which are part of the μFreshener++ architecture (Importer, Extender, Solver, Report, Exporter), while the sixth module (Analyzer) is part of the μFreshener tool. It should be noted that the Importer and Exporter modules of μFreshener++ use the corresponding modules of μFreshener to read and save the μTOSCA model, but this has been slightly modified in the figure for better readability. The execution of μFreshener++ begins

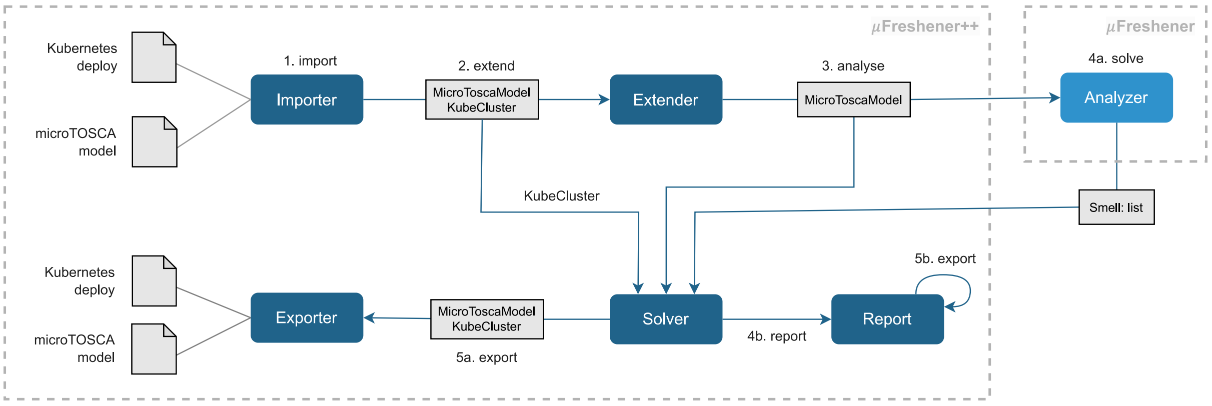


Figure 4.2: Basic workflow of the μFreshener++ tool

with the import procedure, which reads the μTOSCA definition (using the μFreshener Importer module) and the Kubernetes deployment files (using the μFreshener++ Importer) from the disk and represents them within the program as MicroToscaModel and KubeCluster objects. These objects are then processed by the Extender module, which performs the previously described extension task and creates an extended MicroToscaModel. The Analyzer module, which is part of the μTOSCA tool and reused by μFreshener++, detects smells on the extended model and outputs a list of instances found. This list is passed to the Solver, which processes it along with the KubeCluster and the extended MicroToscaModel to output their refactored versions. The Solver also uses the Report module to generate a summary of the actions performed, which is saved to the disk using internal methods. Finally, the MicroToscaModel and KubeCluster are exported to the disk using the Exporter modules of both μFreshener and μFreshener++ projects.

Multiple executions The workflow presented above has been simplified to focus only on the basic steps that the tool performs. Specifically, it was described that after the Extender generates the extended model, it passes it to the Analyzer, which performs the detection, and then the list of smells found is passed as input by the Solver for applying refactoring techniques. However, the execution is more complicated than that. Each time the Solver applies refactoring, it increments an internal counter to keep track of how many refactorings were successfully applied in that single run. At the end of the Solver execution this counter is checked and if it is greater than zero, the process restarts from the Analyzer to run the detection and refactoring phases again. This loop ends when the number of smells solved is zero, and the execution no longer modifies the system. This strategy was adopted because in some cases a single execution cannot resolve all the smells present. For example, when two nodes of type Service communicate directly with each other, the first run of the Solver inserts a MessageRouter between them that resolves the EB smell. Although this smell has been solved, there is still a "wobbly" interaction between the Service and the MessageRouter because there is no timeout or circuit breaker defined in the communication. Without running the detection again, this smell is never discovered and refactored. Thus, it is necessary to perform these steps more than once to resolve as many smells as possible.

4.2 Architecture

After presenting the workflow of μFreshener++, its architecture is analyzed, detailing each module that makes up the tool and the relationships between them. The architecture is analyzed in Figure 4.3, where can be observed that the tool consists of eight different modules that cooperate for performing extension and refactoring tasks. The diagram presented also includes four modules of the μFreshener tool since they are fundamental for carrying out μFreshener++ tasks.

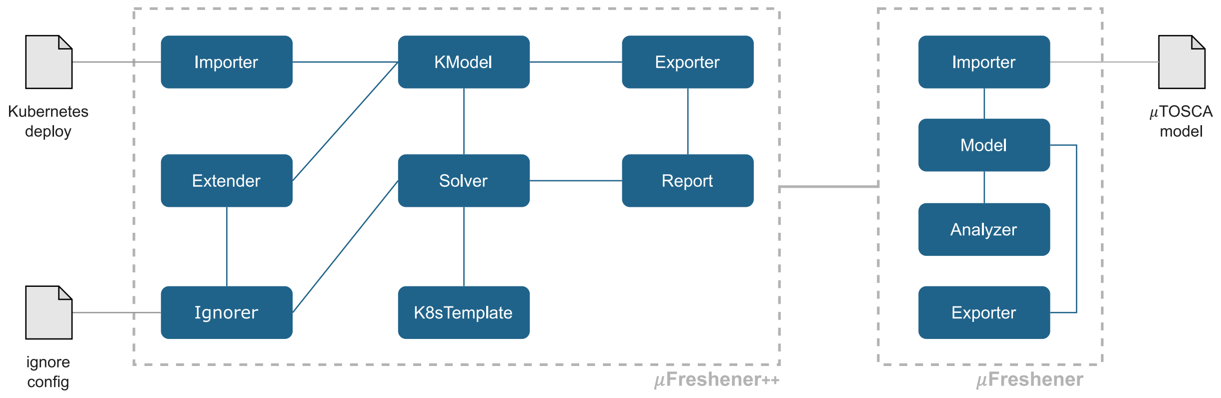


Figure 4.3: Architecture of μFreshener++ tool described as modules and relations among them

4.2.1 Importer

The μFreshener++ Importer is the first module executed by the tool and is responsible for parsing the Kubernetes resources from deployment files and creating data structures representing them. The module reads all Kubernetes YAML deployment files from a specified folder and, for each resource found, creates a *KubeObject* (KModel module) that represents it inside the program.

The Importer adopts the *Strategy* design pattern [7], so it is present an interface that defines a generic Kubernetes importer, and each concrete strategy is implemented in a separate class. So far, only a *YMLImporter* has been developed, but the idea in adopting this pattern was to be as flexible as possible in terms of different ways to build the Kubernetes cluster representation.

4.2.2 KModel

The KModel module defines the classes for representing Kubernetes resources, and the generated KubeObject hierarchy is shown in Figure 4.4. Each KubeObject internally represents the retrieved YAML description through a Python map and then exposes its data through defined properties. The root of the hierarchy is the KubeObject class, which defines the basic properties of each Kubernetes resource, i.e. the name, and has four child classes:KubeWorkload, KubeNetworking, KubeContainer, andKubeIstio.

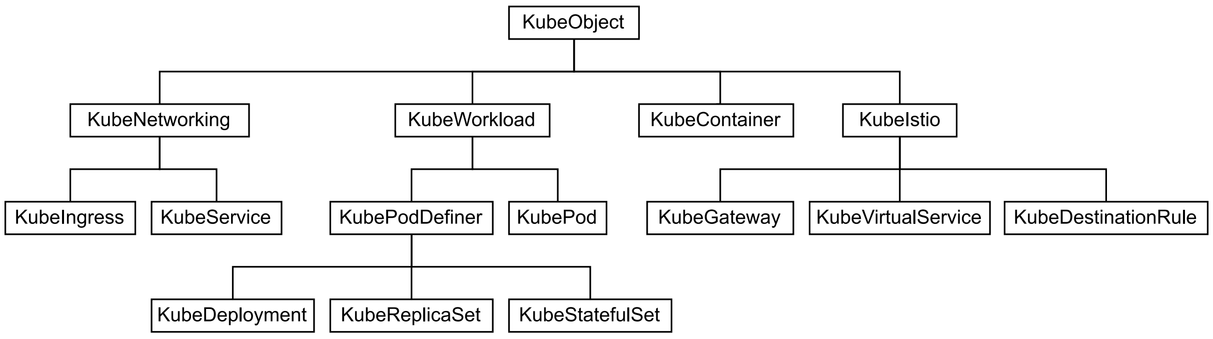


Figure 4.4: KubeObject hierarchy

KubeWorkload is the superclass for each resource that defines a Pod, and it in turn has two subclasses defined, KubePod and KubePodDefiner. KubePod represents a Pod resource, while KubePodDefiner is a resource that manages a Pod such as Kubernetes Deployment, ReplicaSet, and StatefulSet (each represented by the relative "Kube" subclasses).

KubeNetworking represents all objects related to network activities, i.e. routing messages, and has two subclasses, KubeService and KubeIngress, representing the corresponding Kubernetes resources.

A container is not a Kubernetes resource, but KubeContainer is also represented as a KubeObject in this hierarchy because μFreshener++ often uses container information to perform its activities, and representing them using objects simplifies analysis. A KubeContainer is created only when it is extracted from the defining KubeWorkload resource and is not parsed by the importer like other objects.

Finally, KubeIstio is the superclass for each defining Istio resource, i.e. Gateway, VirtualService, and DestinationRule. The KModel module also defines the KubeCluster class, which holds at runtime all created KubeObject and provides useful methods for retrieving them, i.e., retrieving all KubePods exposed by a particular KubeService.

To create a KubeObject starting from a dictionary representing a resource, this module also provides a factory class that parses it and instantiates the correct object for its representation. This way, the Importer has only to create the map from the extracted resource file and does not have to worry about what objects it represents since this task is assigned to the factory.

A hierarchy of objects is useful in representing Kubernetes resources because it simplifies the definition of common properties for objects. For example, each Kubernetes resource has assigned a *name* and a *namespace*, so they are defined only once in KubeObject for each class that represents a Kubernetes resource. On the other hand, only objects of type KubeWorkload define containers, so the property container is defined only by the KubeWorkload class. Each workload must then override this property to retrieve the definition of containers from the internal map and create the representing objects.

4.2.3 Extender

The *Extender* implements the extension functionality of the tool, that performs the enrichment of the μTOSCA model based on the information contained in the Kubernetes deployment files. Also in this case, the Strategy [7] pattern was applied, which defines an Extender interface that must be implemented by each concrete implementation. So far, only one *KubeExtender* has been developed, which performs the extension by exploiting the Kubernetes resources.

The KubeExtender takes as input a MicroToscaModel and a *KubeCluster* object and processes them by performing several controls to enrich the model. The MicroToscaModel class is defined in μFreshener Model module and is the object which represents the μTOSCA topology graph. Implementing all extender controls in a single class could have led to extensibility, testability, and maintenance problems since the result would have been a Large Class [5]. To avoid that, each control was implemented in a separate *Worker* class, which is then instantiated and executed by the KubeExtender to apply necessary changes to the μTOSCA model.

4.2.4 Solver & K8s_Template

The Solver implements the automatic refactoring for the removal of architectural smells for microservices from the application. Each refactoring technique is applied by the *Solver*, which creates, modifies, or deletes existing Kubernetes resources for applying the refactoring. The design of the Solver is the same as that of the Extender module. To avoid having a single Large Class [5] that implements all refactorings, the logic was split into different *Refactoring* classes, and each of them implements only one technique. Also, in this case, the Strategy[7] pattern was applied.

The Solver has a *solve* method that accepts as input a list of smells represented by Smell classes defined by μFreshener-core, to allow code reuse and interoperability between tools. Each smell type is associated with a list of refactoring that can be applied, and the Solver attempts to apply each of these techniques until the result of the execution is positive and the smell is thus solved. Each Refactoring class takes the smell as input, modifies the Kubernetes resources, applies the same changes to the μTOSCA model, and generates a new line in the report describing the change applied to the Kubernetes deployment.

To generate Kubernetes resources the component uses functions in the K8s_template module, that can generate a Kubernetes resource starting from different templates. It is important to outline this module had been designed to generate unique strings for names, and they also always contain a reference to the resource modified, exposed, etc.

4.2.5 Report

The *Report* class is responsible for representing and exporting a generic report and is used by μFreshener++ to generate the log of the operation performed during the refactoring phase. The main purpose of the report is to show the developer the changes introduced by the refactoring and to present which smells were corrected and how which problems could not be corrected and what actions if any, the developer needs to take on the system to complete the refactoring.

To make the whole system flexible, several interfaces have been introduced in this module to facilitate the definition of new reports and output formats. A Report is composed of one or more ReportRow, and then only the concrete implementation of the row decides what and how it is displayed. To export a Report, another abstraction called ReportExported is defined, and the concrete implementations determine different ways to display it. So far, only the RefactoringReport has been implemented, which represents the report of the refactoring phase described earlier, and can be exported to CSV using the dedicated exporter.

4.2.6 Ignorer

μFreshener++ also provides the ability to ignore the detection, extension, and refactoring on some specific nodes, through the functionality provided by the *Ignorer* module. This can be useful, if the tool does not apply refactorings correctly, or if the developer knows the presence of the problem but knows that it will be removed in a future version.

A JSON configuration file is used to configure the Ignorer, which is read and loaded by the Ignorer object when it is created and must match the JSON schema defined in the project's repository. After loading the configuration, it is possible to check if some actions on a node are ignored by a special method.

An example of Ignore configuration can be observed in Listing 1, where two different ignore rules are defined. In particular, the first one ignores all smells and workers on the Service node "container.cart.def", so smell detection and model extension are not executed on this node. The other rule ignores two refactorings on the "orders.default.svc" node. The repository contains an example that enumerates all possible values for *ig-*

nore_smell, *ignore_worker*, and *ignore_refactoring*.

```
{
  "rules": [
    {
      "node": {
        "name": "container.cart.default",
        "type": "micro.nodes.Service"
      },
      "ignore_worker": [
        "all"
      ],
      "ignore_smell": [
        "all"
      ]
    },
    {
      "node": {
        "name": "orders.default.svc",
        "type": "micro.nodes.MessageRouter"
      },
      "ignore_refactoring": [
        "add_api_gateway",
        "add_circuit_breaker"
      ]
    }
  ]
}
```

Listing 1: Example of ignore_config file

4.2.7 Exporter

At the end of the μFreshener++ execution, all data structures must be saved to disk, and this task is performed by the Extender module, which exports the MicroToscaModel object using μFreshener functions and the KubeCluster using internal methods. The Strategy pattern [7] was also applied to the Exporter, and its design is the same as that of the *Importer* to maintain consistency between the modules.

The Exporter performs several operations to ensure that the changes introduced are clear and have as little impact as possible. The folder structure containing the deployment files is maintained, so as not to confuse the user and to restore the familiar structure. Each file is so exported to its original subfolder, and all newly created resources are stored in their folder. For example, a resource is saved under the folder ~*/deployment/*frontend/file.yaml file and, after the μFreshener++ execution, it is exported to disk at the ∼/*out*/frontend/file.yaml location, keeping the *frontend* folder in the path. The file names and the positions of the fields in the YAML files are also preserved, for the same reasons as described before.

4.3 Summary

This chapter offered an overview of the μFreshener++ tool that has been developed for the μTOSCA toolchain extension. We started by presenting the role of μFreshener++ in the toolchain and its relationship with the μFreshener tool (which is utilized by μFreshener++ to detect smell). Following this, we presented the tool workflow that involves its primary modules. This helped us to introduce the two main functionalities of μFreshener++ in the subsequent chapters, namely extension (Chapter 5) and refactoring (Chapter 6). Finally, we discussed the architecture of μFreshener++ and its component modules. This information provides a general overview of the developed tool, which is essential to contextualize the next chapters.

5 Revisiting Microservices Smell Detection in μTOSCA

This chapter describes the revision we conducted on the μTOSCA toolchain for detecting architectural smells, which is a key component of this thesis. The revision involved two main activities, namely the detection of a new smell and the development of the extension functionality in μFreshener++. The detection strategy for identifying the *Multiple Services in One Pod* smell (Section 5.1) involved updating the μTOSCA definition and implementing a new detection algorithm in the μFreshener tool. The extension functionality (Section 5.2) involves correcting and enriching the μTOSCA model. This feature is explained in detail by presenting all the workers developed and their impact on smell detection.

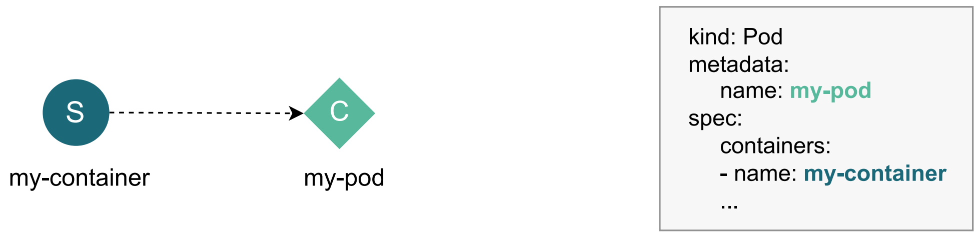
5.1 Detection of a new Microservice Smell

The implementation of the new detector for *Multiple Services in One Pod* smell can be described in two steps. The first activity consists in updating the μTOSCA definition (Section 5.1.1) to add new components to the μTOSCA modeling language that can model microservices deployment. After their introduction, the μFreshener tool was updated for implementing the new detector (Section 5.1.2) for *Multiple Services in One Pod*, which uses the newly introduced components to find out microservices deployed together.

5.1.1 μTOSCA definition extension

To work on the detection of the *Multiple Services in One Pod* smell, we had to enhance the components of the μTOSCA language by introducing some new components. The first component introduced is the Compute node, which represents a component on which are deployed one or more microservices. For connecting a node with the Compute on which the represented component is deployed, the DeployedOn relation is also introduced. Since in the μTOSCA model a Service node (or Datastore, MessageBroker) represents a container running a microservice, and each container in Kubernetes is deployed by a Pod resource, we can state that a Compute node is the representation of a Pod (or Deployment, ReplicaSet, etc.) that defines one or more containers. So, the DeployedOn relationship in the μTOSCA graph always starts from a Service, Datastore or MessageBroker, and targets the Compute representing the Kubernetes object that defines it.

An example of the newly introduced components can be observed in Figure 5.1 which represents, both in the μTOSCA model and as Kubernetes resources, a Kubernetes Pod deploying a container. Modeling this situation in μTOSCA requires a Service node (*my-container*) and a Compute node (*my-pod*) connected with a DeployedOn relationship *my-container* $\longrightarrow$ *my-pod*, which indicates that the container is deployed on that Pod.



```
kind: Pod
metadata:
    name: my-pod
spec:
    containers:
    - name: my-container
    ...
```

Figure 5.1: Example of the newly introduced μTOSCA component. Compute nodes are visually represented through a rhombus, while DeployedOn relationships are represented by a dashed arrow.

5.1.2 Multiple Services in One Pod

The *Multiple Services in One Pod* smell occurs if two or more microservices are deployed by the same Pod resource [19], so when a Pod runs two or more containers that both execute business logic functionalities. As described in the previous Section 5.1.1, for modeling the deployment of a container on a Pod the DeployedOn relation is included in the graph. Each Compute node has one incoming relation for each component deployed on it, and this information can be so used for counting the number of microservices deployed by that Compute.

For applying this strategy, the Analyzer module of μFreshener considers all Compute nodes defined in the graph, and for each of them, it counts the number of incoming DeployedOn relations. If it finds a node that has two or more incoming relations, the *Multiple Services in One Pod* smell is detected on that Compute node. The *Multiple Services in One Pod* smell in the μTOSCA model is outlined by Figure 5.2, where two Service nodes have a relation to the same Compute.

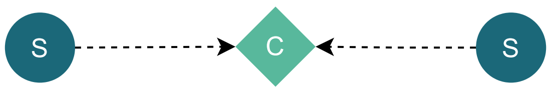


Figure 5.2: Representation of the *Multiple Services in One Pod* smell in the μTOSCA topology graph

The Listing 2 shows an example of a Pod resource, presented as YAML definition, which is affected by the *Multiple Services in One Pod* smell. In this case, the Pod defines two different containers, one running an Nginx instance and the other a MySQL database used by the web server. Analyzing the resource, one can notice the problems that this smell introduces into the system. The Pod is the smallest deployable unit in Kubernetes and so the two containers that it defines are highly interdependent. Therefore, they must be managed together, and it is not possible, for example, to stop or scale just one of them.

Knows Issues The strategy used to resolve these smells may result in the detection of some false positives. To extend the functionality provided by a Pod, in Kubernetes

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
  - name: database
    image: mysql:latest
    ports:
    - containerPort: 3306
```

Listing 2: Example of Pod resource affected by *Multiple Services in One Pod*

multiple design patterns [15] that define more than one container within the same Pod can be applied. An example of this is the *Adapter Pattern*, which consists of defining an additional container in the Pod that transforms the output of the primary container to match the rest of the application. These cases are not to be considered as smell, since the containers contained in the same Pod are not independent, but the applied detection strategy causes them to be recognized as such. Several strategies have been considered to avoid their detection, but none has proven effective, so the only way to avoid detection is to use the functionality of the Ignorer module.

5.2 Model Extension

A μTOSCA model defines the architecture of a microservice system using components and relationships. This model can be generated using different strategies, such as using μMiner, μTOM, or manual creation, but errors may be introduced in the generated representation regardless of the strategy used. Each error in the μTOSCA model results in an inaccuracy in representing one or more application components, which can lead to issues during the detection and refactoring steps. The more accurately the model represents the system components, the more precise the results obtained from the detection process will be.

To enhance the detection capability of μFreshener, we implemented the extension functionality of the μFreshener++ tool, which corrects and enriches the μTOSCA model by utilizing information from the Kubernetes deployment files of the system. The extension of a μTOSCA model offers two benefits for smell detection: it reduces the number of false-positive smell instances and enables the detection of some true-negative instances. For instance, if a Service is a member of the Edge group without defining the *hostNetwork* or *hostPort* properties, μFreshener detects the *No API Gateway* smell on that node. However, by running the extension μFreshener++ recognizes that the Service is not externally exposed and removes it from the Edge group, thereby preventing the false-positive detection. On the other hand, μFreshener++ enables the detection of true negatives by representing Compute nodes within the graph, which are necessary for detecting *Multiple Services in One Pod*.

As outlined in the previous chapter, the Extender module of μFreshener++ implements this feature. This module consists of different workers, each of which performs a single task that modifies the μTOSCA model, resulting in highly cohesive, easily changeable, and readable workers. The following sections provide a summary of all ten workers developed, analyzing their scope and the transformations they apply to the μTOSCA model. For each worker, a figure is presented in the respective section that shows the example of the μTOSCA model before and after its execution. It is important to note that the components in these diagrams are not colored, except for those nodes that were added or modified by the worker. Additionally, it is valuable to notice that, as presented in Section 4.2.2, the prefix "Kube" in the name of a resource (i.e., KubeService, KubePod, etc.) indicates the class of the KModel that represents that resource within the code.

5.2.1 Name Worker

To perform its tasks, the μFreshener++ tool matches nodes and Kubernetes resources based on their names. No standard format for node names is defined for the μTOSCA model, so nodes can take any name. This can lead to malfunctions in μFreshener++, which is why a standardized naming format for μTOSCA nodes is required.

With this perspective, the Name worker was introduced, which is responsible for standardizing the names present in the μTOSCA model by renaming them using a common format. For almost all resources, the standard format of the name is

$$resource\text{-}name\ .\ resource\text{-}namespace\ .\ resource\text{-}type$$

where "resource" is the Kubernetes resource represented by the node. For example, for representing a Kubernetes Service named *orders*, the name given to the node is *orders.default.svc*. This name convention is good for all nodes except services. A node of type Service represents a container, but if two containers are defined by the same resource, the nodes have equal names using this format. For this reason, the name of the container is also inserted in front of the name of the Service, following the format

$$container\text{-}name\ .\ resource\text{-}name\ .\ resource\text{-}namespace\ .\ resource\text{-}type$$

For example, a container named *users* deployed by *users* Pod has name *users.users.default.pod.*

Some examples of changed names are described in Table 5.1. Service in μTOSCA models can be usually named in four different ways, indicating or not namespace and type. In every case, the output name given to the node is *order-container.order.my-app.pod*, which is a unique name for the resource. Even for other nodes, the situation is similar, adding namespace and type where needed, to better map the Kubernetes resource.

It is important to outline that this worker produces a dictionary that contains the

Node name	Node type	Result name
order	Service	order-container.order.my-app.pod
order.my-app	Service	order-container.order.my-app.pod
order.my-app.pod	Service	order-container.order.my-app.pod
order	Compute	order.my-app.pod
cart-service	MessageRouter	cart-service.default.svc
cart-ingress.default	MessageRouter	cart-ingress.default.ing
cart-ingress.default.ing	MessageRouter	cart-ingress.default.ing

Table 5.1: Example of name conversion applied by this worker

mapping between old and new names, and this map is then used for restoring the original names of nodes before exporting the file.

Known issues This approach has some limitations. To standardize a name, the node name must contain a reference to the resource represented. For example, in the Kubernetes deployment is defined a Kubernetes Service named "shop". To standardize the name of the corresponding *N* node, the name of *N* can be *shop*, *shop.default* but not *shop-message-router*, otherwise it becomes impossible for the worker to understand which resource it is. In addition, consider a Pod that deploys two containers and only one of them is represented in the model through a Service S_1 node. If the name of S_1 not follows the convention presented before including also the container name, the worker raises an error because it cannot determine what container is represented by that Service. Finally, another similar problem emerges when the Kubernetes deployment files present a Service and an Ingress resource with the same name *N*, but only one of them is represented in the model by a MessageRouter node *MR*. Even in this case, the worker raises another error because it does not know what resource is represented by *MR*.

The error raised outlines, in all cases, the name of the resource that caused it, and also the indication of how the name can be changed for allowing the correct execution.

5.2.2 Compute Node Worker

The *Compute Node* worker adds the information regarding the deployment of components in the μTOSCA model, by including Compute nodes and DeployedOn relationships. This worker adds a Compute node C_i for each KubeWorkload K_W object defined in the KubeCluster and then, for each container S_i defined by K_W, adds a DeployedOn relation between the Service node S_i and C_i ($S_i \longrightarrow C_i$). Figure 5.3 shows the transformation that this worker applies to the μTOSCA model. To contextualize the figure, it is necessary to assume that C_1 Pod defines S_1 and S_2 while C_2 Pod deploys S_3. Before running the worker, only S_i services (and MessageRouter) nodes are present in the model, so deployment is not represented, but after its execution, two Compute nodes C_1 and C_2 are added to the graph. In particular, C_1 has incoming DeployedOn relations from S_1 and S_2, while from S_3 starts a DeployedOn relation directed to C_2. In both cases, the

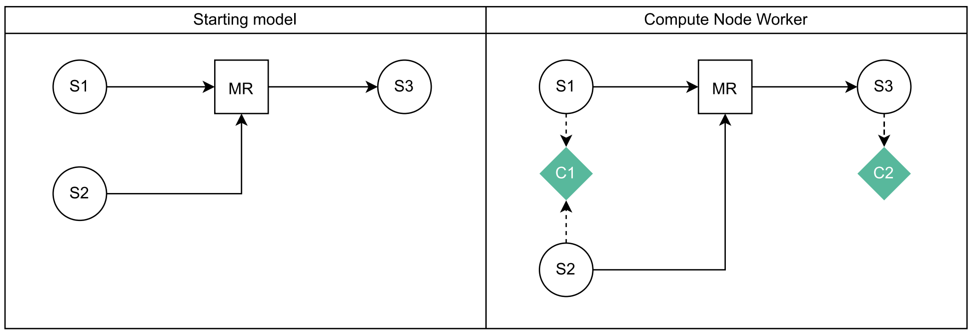


Figure 5.3: Changes applied to the model by the Compute Node worker

deployment of containers is correctly represented as assumed before.

5.2.3 Service Worker

The *Service* worker ensures that all Kubernetes Service resources are represented correctly in the μTOSCA model. For controlling their accurate representation, this worker performs three tasks, outlined in Figure 5.4, which consist of verifying that (1) all Kubernetes services are represented in the model, (2) all interactions involving them are modeled correctly, and (3) there are no Service nodes representing Kubernetes services.

The first control performed by the cluster consists in checking that every Kubernetes

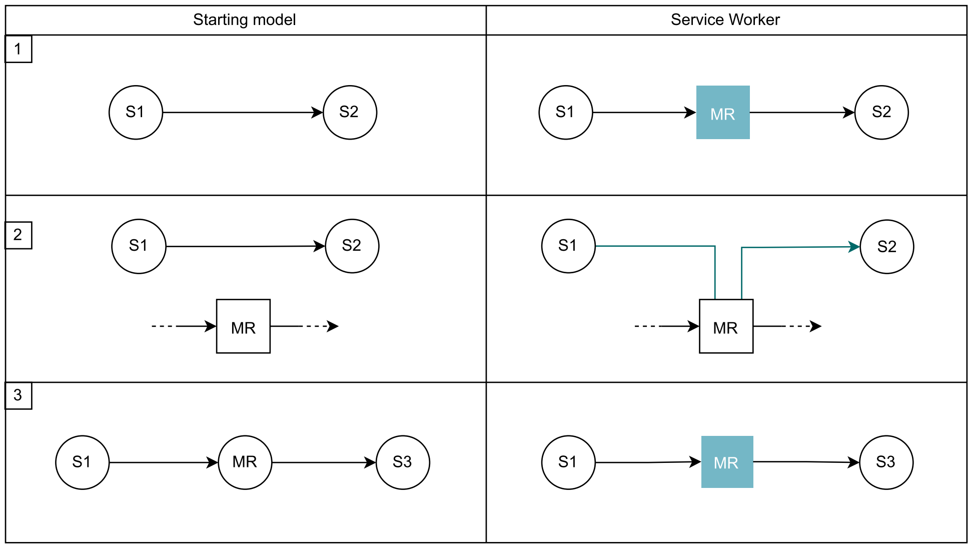


Figure 5.4: Changes applied to the model by the Service Worker worker

Service is represented in the graph. For each KubeService resource, K_S defined in the KubeCluster, the MessageRouter node representing it is searched in the μTOSCA model, and if it is not found, the worker adds a new MessageRouter M_R to the μTOSCA model. Each K_S resource exposes one or more Pods and, for each Container S_i defined by them, the worker adds an interaction $M_R \rightarrow S_i$. Finally, each interaction from a generic Service directed to S_i is redirected to M_R. Referring to example (1) in Figure 5.4, S_1 and S_2 communicate directly, but the worker recognizes in the Kubernetes deployment a Kubernetes Service has been defined. To represent the identified Service in the graph, the Extender adds the MessageRouter M_R node to it, adds the interaction $M_R \longrightarrow S_2$, and redirects all edge directed to S_2 to the M_R.

The second case considered by the worker is when a Kubernetes Service is represented correctly with a MessageRouter M_R node in the graph, but some interactions are missing. The worker controls in the μTOSCA graph if all interactions between each MessageRouter M_R node that models a KubeService and the Service S_i node that represents containers exposed by the Service are represented. In case of missing interaction, it is added the relationship $M_R \rightarrow S_i$, and each direct edge from a Service to S_i is redirected to M_R. Related to the example (2) in Figure 5.4, S_1 and S_2 communicate directly in the graph, but the Kubernetes Service represented by M_R exposes S_2. The worker detects this anomaly and redirects the communication for passing through M_R.

The last correction applied to the model regards Kubernetes services represented through Service nodes. The worker checks that each KubeService is represented with the Message Router type, matching the resource and the corresponding node in the graph. If the retrieved node is not modeled using a MessageRouter node, then it is converted to this type. Related to the example (3) in Figure 5.4, the worker detects that M_R is represented using the wrong type, and so converts it to MessageRouter.

5.2.4 Ingress Worker

The *Ingress* worker has the responsibility of verifying that each Ingress resource defined in the Kubernetes deployment is represented correctly in the μTOSCA model. This worker analyzes all KubeIngress K_I resources defined in the KubeCluster, for each of them control if it is represented in the μTOSCA model, and then:

- If in the μTOSCA model is not found any node representing K_I, a new I Message Router node is included in the μTOSCA model as a member of the Edge group. Then, the worker searches for each Kubernetes Service that is exposed by K_I, retrieves the corresponding MessageRouter M_R from the graph and adds an interaction $I \rightarrow M_R$. If the Kubernetes Service is not of NodePort or LoadBalancer type, M_R is also removed from the Edge group.

- If the MessageRouter node representing the Ingress K_I is found in the μTOSCA model, it is not necessary to add a new node in the graph, and the worker controls that all its interactions are represented correctly. It starts by searching for every Kubernetes Service exposed by K_I, then retrieves the corresponding MessageRouter nodes M_R from the graph and adds an interaction $I \longrightarrow M_R$. If the Kubernetes Service is not of NodePort or LoadBalancer type, M_R is also removed from the Edge group.

The operations described above are summarized by Figure 5.5, which outlines both cases this worker handles. In case (1), a MessageRouter M_R representing a Kubernetes ClusterIP Service is a member of the Edge. The worker detects the presence of a KubeIngress I that exposes the Service, so it adds the MessageRouter I to the model with an interaction directed to M_R, and also removes M_R from the Edge. In case (2), the Ingress is also represented in the model by a MessageRouter I but is missing the interaction directed to MR_2. The worker so detects this issue and adds the edge $I \longrightarrow M_R$ to the μTOSCA model.

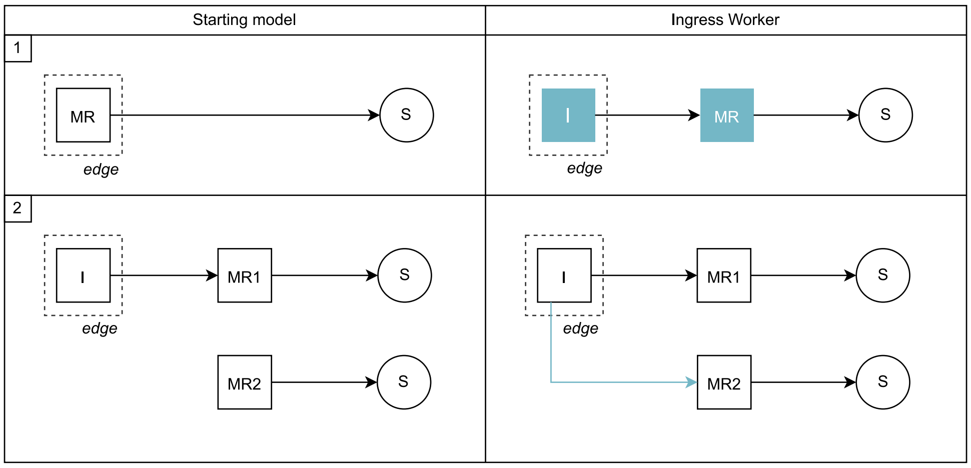


Figure 5.5: Changes applied to the model by the Ingress worker

5.2.5 Database Worker

The *Database* worker is designed to control that each database of the application is correctly represented using the Datastore type.This worker considers all Service nodes that do not have outgoing interactions because is considered that a datastore only holds data and receives messages without contacting anyone. For each Service considered, the worker retrieves the corresponding KubeContainer definition, and controls if at least one of these three situations occurs:

- The name of the container contains a substring which indicates that the container runs a database, i.e., the container is named "mysql-db" which contains the substring "mysql"
- The image executed by the container is an image of a database, i.e., the container runs "mongo:latest"
- The container exposes a standard database port, i.e., it defines the 3306 port which is the one used by MySQL databases

If at least one of these conditions is verified, then the analyzed container runs a database, and so the corresponding Service node is converted to Datastore type.

Figure 5.6 shows changes that this worker applies to the μTOSCA model. In this case, a communication between two services I and D is represented and, after running the

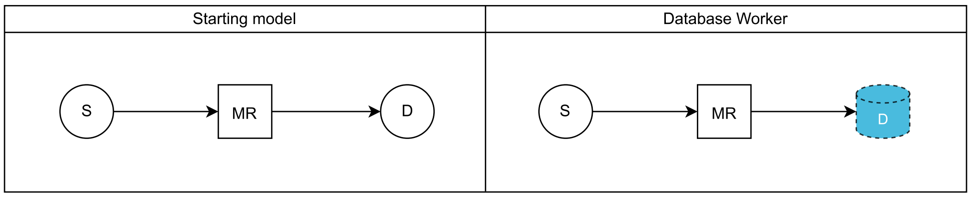


Figure 5.6: Changes applied to the model by the Database worker

worker, the Extender detects that D is a database because, for example, the container runs the official MySQL Docker image. In this case, one of the conditions mentioned before is verified and so D is converted to Datastore type.

5.2.6 Container Worker

The *Container* worker is responsible for checking whether or not Service nodes are correctly members of the Edge group.

For each container, the worker analyzes if the KubeWorkload resource that defines it has the property *hostNetwork* set or defines at least one *hostPort*. In case of a positive match, the container is exposed outside the cluster and so the corresponding Service node must be included in the Edge group. In case of a negative match, the corresponding Service node is removed from the Edge group. Figure 5.7 shows the described transformation applied to the μTOSCA model. In case (1) a Service is a member of the Edge, but the

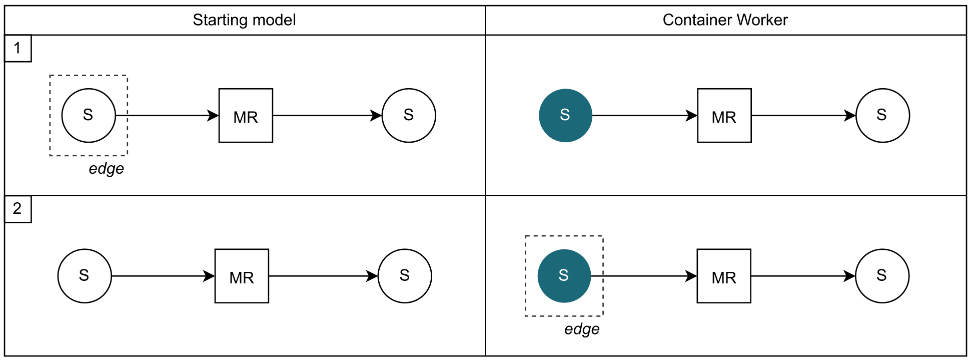


Figure 5.7: Changes applied to the model by the Container worker

worker detects that it does not satisfy the proprieties mentioned before and so removes it from the group. Case (2) shows the exact opposite, that is, a node added to the Edge because, for example, the represented container defines a *hostPort*.

5.2.7 Message Router Edge Worker

The *Message Router Edge* worker verifies that each MessageRouter is correctly included in the Edge group, and this is similar to what the *Container* worker does for Service nodes. This worker adds and removes MessageRouter nodes based on the *kind* property of the represented KubeService, discussed in Background Section 3.2.2. The worker parses each KubeService K_S defined in the KubeCluster and checks whether or not its type is consistent with the presence in the Edge group of the MessageRouter node representing it. Table 5.2 summarizes the case considered by this worker, based on the condition described.

K8s Service kind	Edge contains MR	Correction
ClusterIP	Yes	Remove MR from the Edge
ClusterIP	No	-
NodePort	Yes	-
NodePort	No	Add MR to the Edge
LoadBalancer	Yes	-
LoadBalancer	No	Add MR to the Edge

Table 5.2: Table that summarizes cases considered by thiw worker

The changes applied by this worker are outlined in Figure 5.8. In case (1), a MessageRouter M_R, for example of ClusterIP type, is removed from the group by the worker, while in case (2) the M_R is added to the group because, for example, of NodePort type.

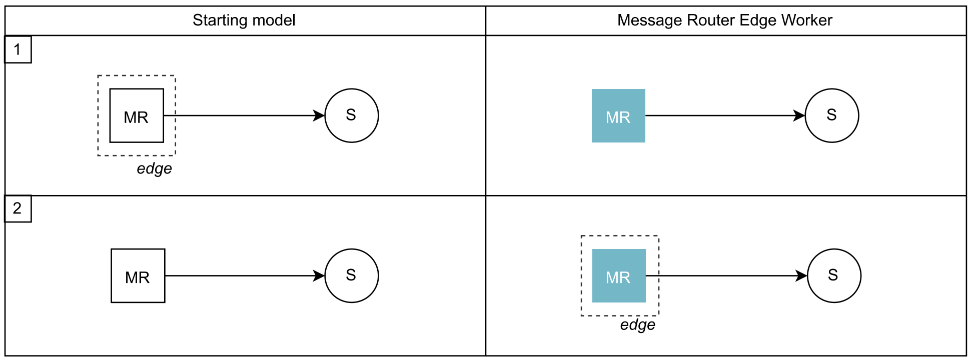


Figure 5.8: Changes applied to the model by the Message Router Edge worker

5.2.8 Istio Timeout Worker

The *Istio Timeout* worker is the one responsible for checking timeouts defined using Istio resources. As outlined in the previous Section 3.2.3, with Istio a timeout can be configured using two different resources, VirtualService or DestinationRule.

When a timeout is specified using the VirtualService resource, it is possible to specify *host*, *destination*, and *timeout* fields. Without detailing too much the meaning of the fields (which can be explored in the Istio Docs [13]), we can affirm that two cases can be distinguished, $host = destination$ and $host \neq destination$, where *destination* and *host* are hostnames of Kubernetes Service. This worker starts by retrieving from the KubeCluster each V_S IstioVirtualService resource that has the *timeout* property set. Then, it searches in the μTOSCA model for the MessageRouter specified by the *destination* field, and it enriches incoming relations for adding the *timeout* property. The property is added to all incoming interactions when *host* = *destination*, and only in the relationship between *host* and *destination* nodes in the other case.

Considering DestinationRule resources, it is possible to define a timeout for communication directed to a Kubernetes Service for every request directed to it. The DestinationRule resource defines two fields important for this analysis, *host* and *timeout*, with *host* property that specifies the Kubernetes Service on which this rule applies. The worker searches in the μTOSCA model for the MessageRouter corresponding to the specified *host*, and sets the *timeout* property as true for every incoming interaction.

This check is outlined by Figure 5.9 where the worker finds a VirtualService with $host = M_R$ and a defined timeout, and so modifies all incoming relations to that node for setting the *timeout* property, represented with "t".

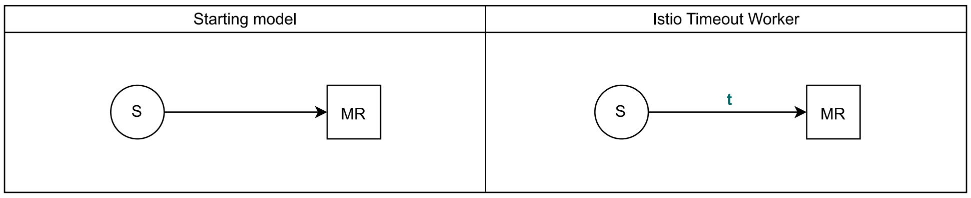


Figure 5.9: Changes applied to the model by the Istio Timeout worker

5.2.9 Istio Circuit Breaker Worker

The *Istio Circuit Breaker* worker controls the presence of defined circuit breakers in the communication. Using Istio it is possible to define a circuit breaker using the *DestinationRule* resource, by specifying, using the *trafficPolicy* field, policies that define circuit breaking for the communication, by for example defining the maximum number of connections, pending requests, etc [12]. These rules apply to each communication directed to the Kubernetes Service indicated by the *host* field. This worker analyzes each IstioDes-

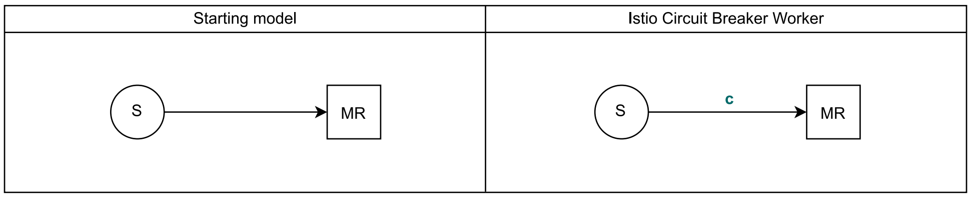


Figure 5.10: Changes applied to the model by the Istio Circuit Breaker worker

tinationRule defined in the KubeCluster, and then considers only the ones that specify the *trafficPolicy* field. For each DestinationRule, the worker retrieves from the μTOSCA model the MessageRouter node specified by the *host* property and then enriches all its incoming relations by adding the *circuit_breaking* property.

The analysis performed by this worker is outlined by Figure 5.10, where the worker finds a DestinationRule with $host = M_R$ and defined *trafficPolicy* field, and so modifies all incoming relations to that node for setting the *circuit_breaking* property, represented with "c" in the model.

5.2.10 Istio Gateway Worker

The *Istio Gateway* worker checks the presence of gateways defined using the dedicated Istio resource.

A Gateway resource defines two important fields for this analysis, a name N and one host H. A Gateway resource cannot expose directly a Kubernetes Service to the outside but it is necessary to deploy a proper VirtualService V_S resource. The V_S must (1) contain the gateway name N in its *gateways* field, (2) contain the host H in its "hosts" field, and (3) define a RouteRule that redirects traffic to the Kubernetes Service to expose. This situation is outlined in Figure 5.11.

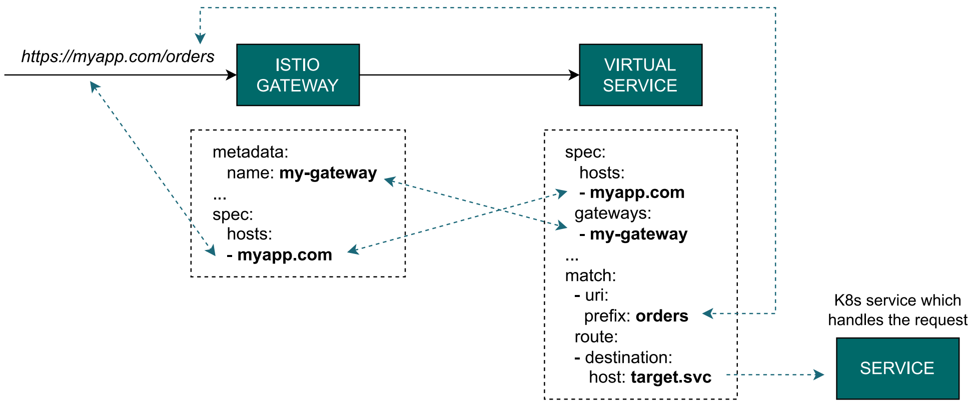


Figure 5.11: Representation of how an Istio Gateway exposes a Kubernetes Service

For each Istio Gateway defined in the KubeCluster, this worker finds the match with a VirtualService following the condition described before. In case of a positive result, it creates a new Message Router I_G node corresponding to the Gateway and includes it in the Edge group of the μTOSCA model. Then, it retrieves the Message Router M_R corresponding to the Kubernetes Service specified as the destination host of the RouteRule and adds an interaction $I_G \rightarrow M_R$ between the nodes. If M_R is at the Edge but is not NodePort or LoadBalancer, the worker also removes it from the group. Figure 5.12 outlines the changes in the model before and after applying the worker.

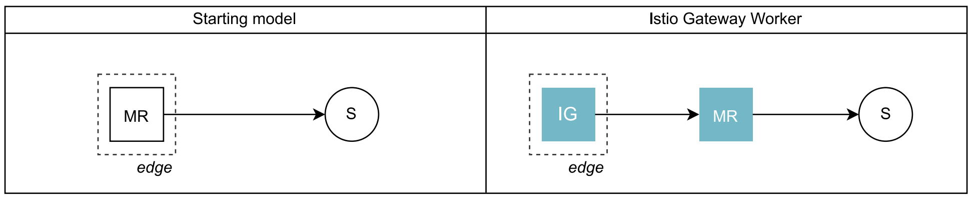


Figure 5.12: Changes applied to the model by the Istio Gateway worker

5.2.11 Worker summary

Section 5.2 presented all the workers developed, that the Extender module of μFreshener++ executes for performing the *extension* of the μTOSCA model. In total 10 different workers were presented, with 9 of them that perform permanent changes to the μTOSCA model, and the *Name* worker that instead applies temporary changes only for supporting the μFreshener++ execution.

As described, each worker can perform more than one *control*, i.e., the Container worker can (1) adds a Service that defines *hostNetwork* to the Edge (2) adds a Service that defined *hostPort* to the Edge and (3) remove a service from the Edge. All workers developed are summarized in Table 5.3, and for each are also outlined the *controls* that it can perform.

Worker	**Control**
Container	Add service to edge (hostNetwork) Add service to edge (hostPort) Remove service from edge
Compute	Add compute nodes
Database	Convert service to datastore
Ingress	Add missing ingress Add missing interactions
Istio Timeout	Add missing timeout
Istio Circuit Breaker	Add missing circuit breaker
Istio Gateway	Add missing gateway
MessageRouter Edge	Add message router to edge Remove message router from edge
Service	Add missing service Convert service to message router Change interactions

Table 5.3: Summary of workers presented in this Section

5.3 Summary

This chapter discussed the contributions made to the μTOSCA toolchain's smell detection capabilities. These contributions consist of two main modifications: (1) the detection of a new smell and (2) the extension of the μTOSCA model. To detect the new *Multiple Services in One Pod* smell, we added two new components to the μTOSCA definition, the Compute node and the DeployedOn relationship, which allow for modeling the deployment of Service nodes. After doing that, we modified the μFreshener tool to implement detection of the *Multiple Services in One Pod* smell. On the other hand, the extension activity involves enriching the μTOSCA model using information from Kubernetes deployment files, and this functionality was implemented in the μFreshener++ tool. While we introduced this feature previously, this chapter provides a more detailed analysis of each modification that μFreshener++ can make to the μTOSCA model to extend it.

We made these modifications to the toolchain with the aim of improving μFreshener's detection capabilities. The ability to detect new smell can enhance its capabilities by identifying problems that were previously unrecognized. On the other hand, the extension results in a μTOSCA model that is closer to the application deployment, potentially increasing the accuracy of odor detection.

6 Automated Refactoring Support

This chapter presents the refactoring feature of the μFreshener++ tool, which automatically applies some of the refactoring techniques outlined by Neri et al. [19] to remove architectural smells detected by μFreshener. To resolve a single smell, μFreshener++ creates, modifies, or deletes Kubernetes resources that define the system deployment.

Considered the *Multiple Services in One Pod* smell detection we introduced in this thesis (Section 5.1), the μTOSCA toolchain can detect five architectural smells for microservices, and μFreshener++ can apply five refactoring techniques to solve four of them, namely *Endpoint-based Service Interactions*, *Multiple Services in One Pod*, *No API Gateway*, and *Wobbly Service Interaction*. The only smell that cannot be refactored is *Shared Persistence*, as no automatic techniques can be implemented for its solving.

It is worth noting that some of the applied refactoring techniques require developers to make additional changes to the microservices. For example, after adding a new Kubernetes Service via refactoring, the developer must update the endpoints used in the microservices to take advantage of it. Thus, while some refactoring techniques can be fully automated, others may require additional some limited manual effort by the developer.

For each smell detected by μFreshener, this chapter presents all the refactoring techniques that can be applied for its removal [19]. Then, only the strategy implemented in μFreshener++ are discussed and analyzed in-depth.

6.1 Endpoint-based Service Interactions

The *Endpoint-based Service Interactions* smell occurs between two microservices X and Y when X directly invokes an instance of Y without passing through any service discovery mechanism [19]. This situation violates the *Horizontal Scalability* principle of microservices and leads to issues related to microservices scaling described in the previous Section 3.4.1.

Figure 6.1 outlines the representation of the smell in the μTOSCA model, and the refactoring that can be applied to remove it [19]. This smell appears in the graph as a Service X that has an interaction with a Service Y, and the edge linking them is not enriched with *service_discovery* property. Three different techniques can be used to

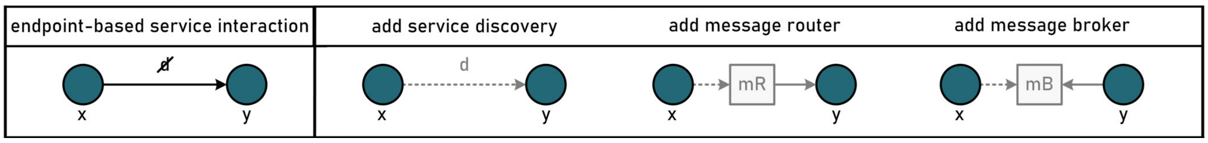


Figure 6.1: Representation of the smell and possible refactoring in μTOSCA models [27]

solve an *Endpoint-based Service Interactions* smell. The most common solution is *Add Service Discovery* refactoring, which adds a service discovery to the communication that can be implemented as a registry listing all running microservices' locations. When X needs to contact another microservice Y, it asks the registry for Y's current location and then uses the obtained result to send requests. Another solution that can be applied is *Add Message Router*, which consists of adding a message router between the two services to break the direct interaction. When X needs to contact Y, it sends its message to the message router, which then forwards it to an available instance of Y. The last available solution is *Add Message Broker*, which requires changing the communication pattern by implementing the message broker pattern to decouple the direct interaction. In this way, X sends the requests to the message broker Y, and, when an instance of Y is ready, it retrieves the message from the message broker and performs the required task. Any refactoring presented solves the problem related to horizontal scalability by breaking direct interaction and avoiding direct communication from one microservice instance to another.

Of the three refactoring techniques presented, μFreshener++ implements only *Add Message Router*, as it can be applied automatically by generating a new Kubernetes Service. Automatic application of refactoring *Add Service Discovery* was not feasible, because the introduction of this mechanism into the system requires modification of all microservices by the developer. *Add Message Broker*, on the other hand, requires a change in the communication paradigm between microservices, and again the developer needs to perform consistent changes to the microservices involved.

6.1.1 Add Message Router refactoring

Considering an interaction X⟶Y, this smell is detected by μFreshener on the Y node. μFreshener++ starts by retrieving the Pod that defines the container represented by Y and then uses the K8s_template μFreshener++ module to generate a Kubernetes Service that can expose this Pod within the cluster.

The *Add Message Router* technique is summarized in Figure 6.2, where are outlined both changes to the μTOSCA model and the topology graph. This refactoring modifies the Kubernetes deployment by introducing a new resource Service Z that exposes the Pod Y (represented by the Service node Y in the μTOSCA model). These changes applied to the Kubernetes deployment are reflected in the μTOSCA model by (1) the addition of a new node MessageRouter Z, (2) the elimination of the direct communication $X \longrightarrow Y$, and (3) the introduction of two new interactions, X⟶Z⟶Y, representing the communication between services after the introduction of Z. The generated Kubernetes Service resource is shown in Listing 3.

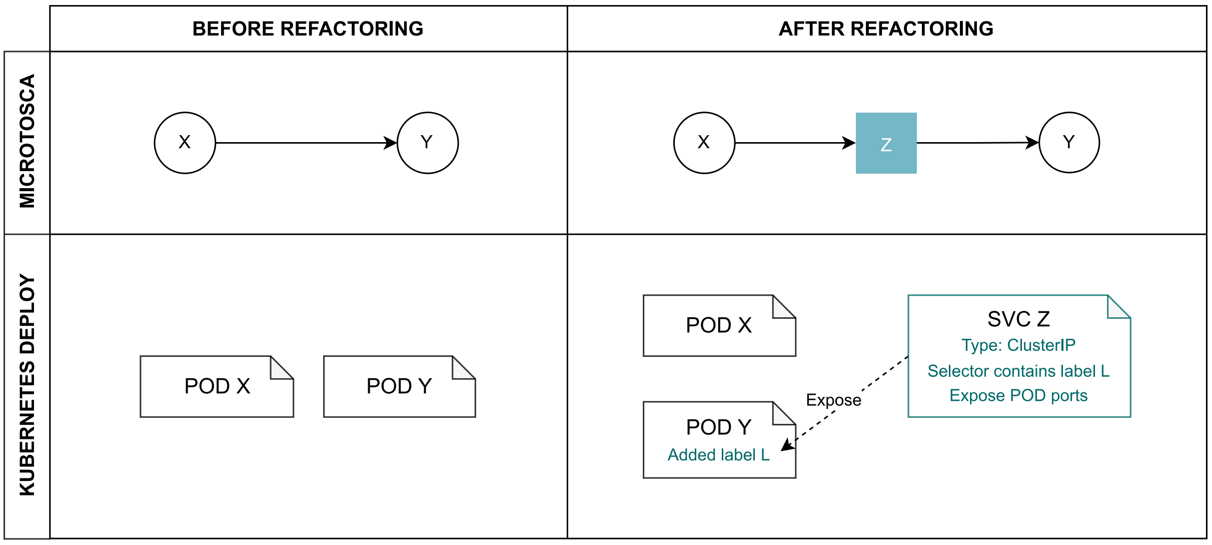


Figure 6.2: Refactoring on μTOSCA model and Kubernetes deployment files applied by Add Message Router

```
apiVersion: v1
kind: Pod
metadata:
  name: Y
  labels:
    service: Y
    H.default-svc-mf: bf1dcef19b..
spec:
  containers:
  - name: Y
    image: ESSeRE/orders
    ports:
    - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: Y-clusterip-svc-mf
  namespace: default
spec:
  ports:
  - name: H.H.default-port-80-mf
    port: 80
  selector:
    H.default-svc-mf: bf1dcef19b..
```

Listing 3: Example of Service resource generated for exposing to the Pod to other microservices

This Kubernetes Service is of type ClusterIP, as the smell refers to internal interactions between microservices and so there is no need to make it accessible to external clients. The name of the resource (*Y-clusterip-svc-mf*) is generated starting from the name of the Pod Y, for having a direct reference to the exposed resource. Later we will refer to the Kubernetes Service using the name Z instead of its real name, to simplify the reading of this section. The namespace is the same as the Pod. To allow the Service to expose the Pod Y a new label is generated, *H.default-svc-mf:bf1dcef19b..*, which is included in the labels of the Pod and the selectors of the Service Z. The ports exposed by the Kubernetes Service are defined using all the ports of Y, so for every port on the Pod there is a corresponding opened port on Z. Each port is also assigned a name (if not already present), which refers to the resource (and port number) exposed by it.

The generated Service solves the *Endpoint-based Service Interactions* smell, but deploying it in the cluster is not enough to eliminate the smell detected. To finish the application of this refactoring, the developer must also change the host to which X sends requests, by modifying the address of Y to the hostname of the generated Kubernetes Service. The generated report signals the developer to change the hostname used and specifies the correct value to use by showing the Kubernetes Service hostname.

6.2 No API Gateway

The *No API Gateway* smell arises in the application when a microservice instance is directly accessible to the client from outside the system, i.e., when messages reach microservice instances directly instead of passing through a gateway [19]. This smell violates the *Horizontal Scalability* principle and causes problems when scaling a microservice, as described in the previous Section 3.4.2.

Figure 6.3 outlines the representation of the smell in the μTOSCA model and the *Add API Gateway* refactoring that can be applied to remove it [19]. This smell is detected when a Service or MessageBroker node is a member of the Edge group. *Add API Gateway*

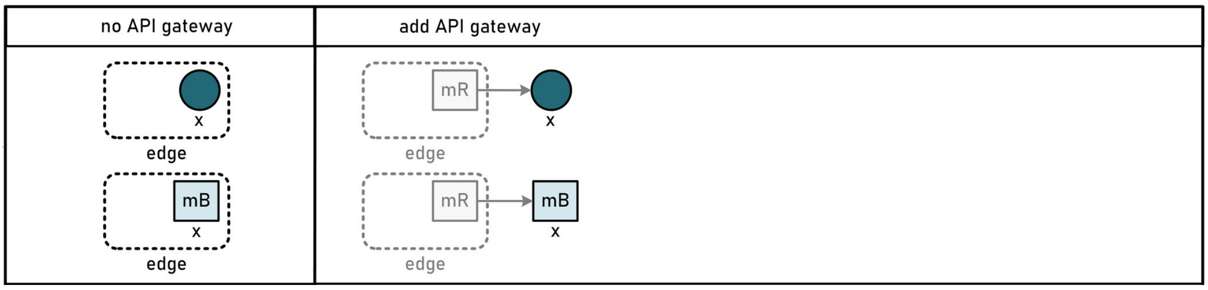


Figure 6.3: Representation of the smell and possible refactoring in μTOSCA models [27]

consists of introducing an API gateway that serves as an entry point into the system, and after applying this refactoring the microservices' functionalities are reachable by external clients only through the gateway. In this way, incoming messages are processed by the gateway and the load is distributed across all available replicas allowing *Horizontal Scalability*. This refactoring can be applied automatically by the μFreshener++ tool.

6.2.1 Add API Gateway refactoring

The *No API Gateway* smell is detected by μFreshener directly on the Service or MessageBroker node member of the Edge, which we will refer to as X following the naming of the scheme presented before.

μFreshener++ begins by retrieving the container directly exposed to the outside (represented by Service X) and the X Pod resource that defines it. These resources are then analyzed together to extract the ports accessible by external clients. During the ports extraction, two different cases can occur:

- The Pod defines the *hostNetwork=true* property, so it shares the same network as the host, and any port defined by the container must be exposed
- The Pod does not have the *hostNetwork* property set (or is false), so only ports exposed via the *hostPort* are considered.

After extracting the P list of ports to be exposed, μFreshener++ searches among all defined Kubernetes services to see if at least one of them (1) is of type NodePort or LoadBalancer, (2) is capable of exposing the P list of ports, and (3) defines at least one selector that matches a label defined by X. Depending on whether a Kubernetes Service is found or not, two cases can be distinguished.

If found, the Kubernetes Service is changed by adding P to its exposed ports list. Since the labels and selectors also match, this Service is ready for being deployed in the cluster. Otherwise, if no Kubernetes Service with the described characteristics is found, it is necessary to create a new Kubernetes Service Y that exposes the Pod X using the K8s_template module for its generation. After creating Y, the final step is to make the Pod unreachable from the outside by removing the *hostNetwork* property and any defined *hostPort*. The strategy described is summarized in Figure 6.4, which shows the transformations of both the μTOSCA topology graph and the Kubernetes deployment files.

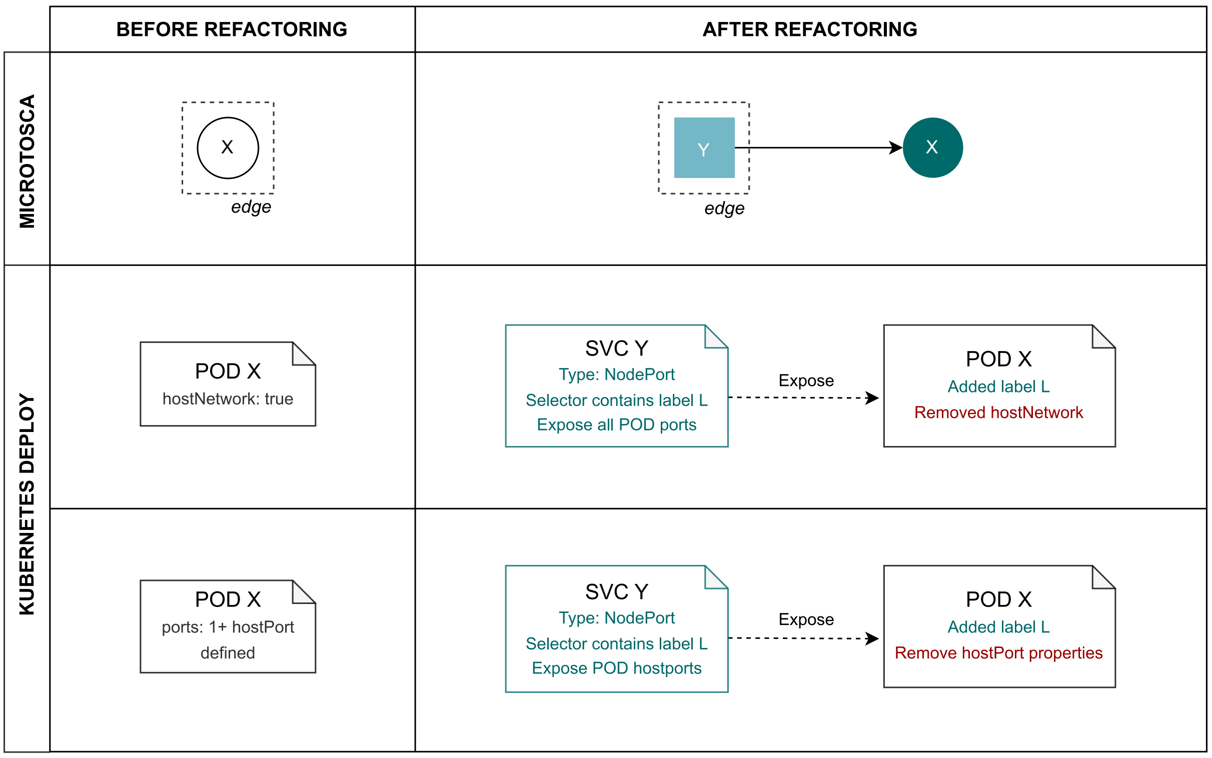


Figure 6.4: Refactoring on μTOSCA model and Kubernetes deployment files applied by Add API Gateway

Whether the Pod X exposes ports via *hostNetwork* or *hostPort*, a new resource Kubernetes Service of type NodePort is generated, which can expose the Pod X's ports to the outside if configured appropriately. The Pod X is also modified to remove the properties that expose it directly to outside the cluster. In the μTOSCA model, the newly generated Service is represented by MessageRouter Y member of the Edge group added to the graph. The Service X is also removed from the Edge group as it is no longer exposed, and the Y$\longrightarrow$X relationship is added to represent that the Service exposes the Pod. Listing 4 outlines an example of *Add API Gateway* refactoring, by showing the *orders* Pod resource (representing X) and the Kubernetes Service (Y) generated for exposing it.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: orders
  namespace: ns
  labels:
    orders.ns-svc-mf: hf5463fh34..
spec:
  containers:
  - name: C
    image: ESSeRE/orders
    ports:
    - containerPort: 80
      hostPort: 80
    - containerPort: 8080
      hostPort: 8086
    - containerPort: 3000

```

```yaml
apiVersion: v1,
kind: Service,
metadata:
  name: orders-nodeport-svc-mf,
  namespace: ns
spec:
  type: NodePort,
  selector:
    orders.ns-svc-mf: hf5463fh34..
  ports:
  - name: C.orders.ns-port-80-mf
    port: 80
    nodePort: 30000
  - name: C.orders.ns-port-8080-mf
    port: 8080
    nodePort: 30001
```

Listing 4: Example of Kubernetes Service resource generated for exposing the Pod

The generated Kubernetes Service is of type NodePort for exposing microservice functionality directly on host ports, as was the case with *hostNetwork* and *hostPort* properties. The name (*orders-nodeport-svc-mf*) is generated from the name of the Pod it contains *orders* as a prefix. The namespace is the same as the *orders* Pod. The label *orders.ns-svc-mf: hf5463fh34..* is also generated, which is a random value generated assigned to both the Pod label and the Service selector for defining the Pod exposition. The Service also defines an exposed port (*nodePort*) for each *hostPort* defined by the Pod. If *orders* had the property *hostNetwork=true*, port 3000 would also be exposed.

It is important to note that the NodePort type only exposes ports in the range 30 000-32 767, but the Pod properties can expose every port number on the host. After applying Add API Gateway, the number of the port on which services are exposed to client changes, and the developer has to solve this problem, i.e., by implementing port forwarding. The report generated by μFreshener++ outlines each pair old-port:new-port.

6.3 Multiple Services in One Pod

The *Multiple Services in One Pod* smell occurs if a Pod executes more than two microservices [19], so in case a Pod deploys two or more container that both runs business logic. This smells violates the *Independent Deployability* principle, causing problems with microservices deployment.

Figure 6.5 illustrates the smell in the μTOSCA model and also presents the *Split Services* refactoring for solving it [19]. This smell is detected in the μTOSCA model when two or more Service nodes have a DeployedOn relation to a single Compute node, and so this represents two containers deployed by the same Kubernetes component.

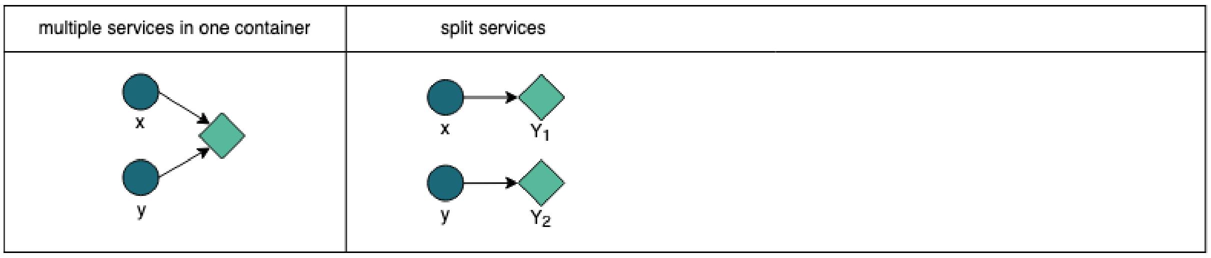


Figure 6.5: Representation of the smell and possible refactoring in μTOSCA models [27]

Refactoring *Split Services* can be applied to remove this smell and solves the problem by generating a resource (Pod) for each microservice (container) to be deployed. The μFreshener++ tool is able to apply this refactoring automatically, and the strategy adopted is described in the next section.

6.3.1 Split Services

The μFreshener tool detects the *Multiple Services in One Pod* smell directly on the Compute type node where one or more services are deployed. μFreshener++ begins its execution by extracting the Compute node from the smell and retrieving the Kubernetes Pod Y resource corresponding to that Compute node. For each X_i container defined by Y, a Y_i copy of Y is created that deploys only the X_i container. At the end of the procedure, the Kubernetes Pod Y is deleted, and in its place remains a set of pods $Y_{1..n}$ (with n containers), where each Y_i defines only the corresponding container X_i. In this way, the original Pod resource is split into many smaller resources, so that we have a set of containers that can be deployed independently.

The above process is illustrated in Figure 6.6, which shows the changes to both the μTOSCA model and the defined Kubernetes resources.

Analyzing the Kubernetes deployment changes, we can observe that the resource Pod Y defining two containers X_1 and X_2 has been split into two pods Y_1 and Y_2, each performing the deployment of the respective container X_i. This change is represented in the model by the creation of two Compute nodes Y_1 and Y_2, each with an incoming DeployedOn relationship originating from X_1 and X_2, respectively.

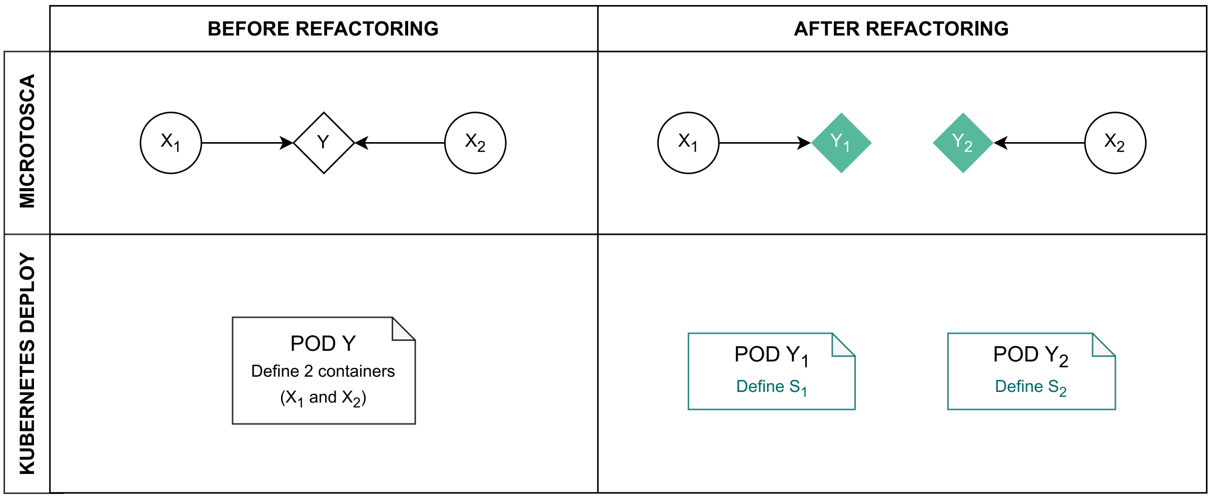


Figure 6.6: Refactoring on μTOSCA model and Kubernetes deployment files applied by Split Services

An example of the application of *Split Services* is shown in Listing 5. In this case, a Pod resource named *checkout* (Y) which deploys two containers *shipping* (X_1) and *payment* (X_2) is defined. When this refactoring is applied, *checkout* is split into two separate pods, *shipping-checkout* and *payment-checkout*, represented by the following two resources. Each created Pod is assigned a name to distinguish it from the others, which is created by concatenating the name of the container with that of the Pod itself.

It is important to point out that when refactoring is applied, the Y resource defining multiple containers is copied in its entirety (except for the containers) to create the new Y_i resource. This is done because the values of all Pod fields, such as labels, remain the same for all defined pods, and so a minimal impact on the system is achieved. For example, a Kubernetes Service that previously exposed a Pod and all the containers defined in it can continue to do so even if the Pod has been split since neither the labels nor the ports change. This can also be observed in the example where the *labels* are maintained for both created pods.

For this reason, *Split Services* is the only one of the presented refactorings that are applied automatically by μFreshener++ and therefore do not require any further intervention from developers to complete the refactoring.

```
apiVersion: v1
kind: Pod
metadata:
  name: checkout
  labels:
    application: store
spec:
  containers:
  - name: shipping
    image: ESSeRE\shipping
    ports:
    - containerPort: 80
  - name: payment
    image: ESSeRE\payment
    ports:
    - containerPort: 8080
```

```
apiVersion: v1
kind: Pod
metadata:
  name: shipping-checkout
  labels:
    application: store
spec:
  containers:
  - name: shipping
    image: ESSeRE\shipping
    ports:
    - containerPort: 80
```

```
apiVersion: v1
kind: Pod
metadata:
  name: payment-checkout
  labels:
    application: store
spec:
  containers:
  - name: payment
    image: ESSeRE\payment
    ports:
    - containerPort: 8080
```

Listing 5: Example of application of the Split Services refactoring on the orders-users Pod

6.4 Wobbly Service Interaction

An interaction between two microservices X and Y (with X that invokes Y) is "wobbly" when a failure of the invoked microservice Y results in triggering an error also in the microservice X [19]. This smell violates the *Isolation of Failures* principle.

The *Wobbly Service Interaction* smell is shown in Figure 6.7, along with the three refactorings that can be applied to resolve it [19]. It occurs in interactions $X \longrightarrow Y$ (with X Service and Y Service or MessageBroker) not enriched with *timeout* and *circuit_breaker* properties.

The first refactoring that can be applied is *Add circuit breaker*, which inserts a circuit breaker to the communication for stopping interactions directed to Y if it is too overloaded or fails. This way, the circuit breaker responds to X with an error message, and the failure is not propagated to other microservices. Another refactoring is *Add Message Broker*,

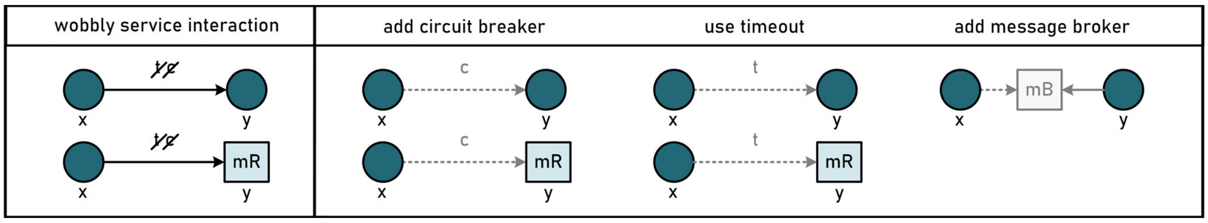


Figure 6.7: Representation of the smell and possible refactoring in μTOSCA models [27]

which adds a message broker that breaks the direct communication between X and Y. That way X sends a request to the broker and, when an instance of Y is ready to process a request, retrieves it from the message broker. In this way, errors remain confined to Y. The last solution presented is *Use Timeout*, which defines an interaction timeout to stop X waiting for a response when Y is in a failed state. When the timeout ends, X receives an error message, that can be managed, so that the error is not propagated.

Two out of three refactoring techniques described can be applied automatically by μFreshener++, *Use Timeout* and *Add Circuit Breaker* techniques. For the Add Message Broker refactoring, on the other hand, an automatic application algorithm could not be defined, due to the problems discussed before (Section 6.1).

However, the refactorings *Add Circuit Breaker* and *Use Timeout* have not yet been fully implemented. Using Istio resources, timeouts and circuit breakers can only be defined for communication directed to Kubernetes Service and not to individual pods. Using Istio is so possible to directly cover only the case of refactoring between interactions involving a Service and a MessageRouter. This, however, is not a problem, the other case is indirectly covered. When is present a direct interaction between two services, μFreshener++ applies the *Add Message Router* refactoring which generates a new Kubernetes Service to break the interaction. Once the Service is generated, one of the *Use Timeout* and *Add Circuit Breaker* options can be applied to resolve the *Wobbly Service Interaction* smell as well. The described procedure is outlined in Figure 6.8.

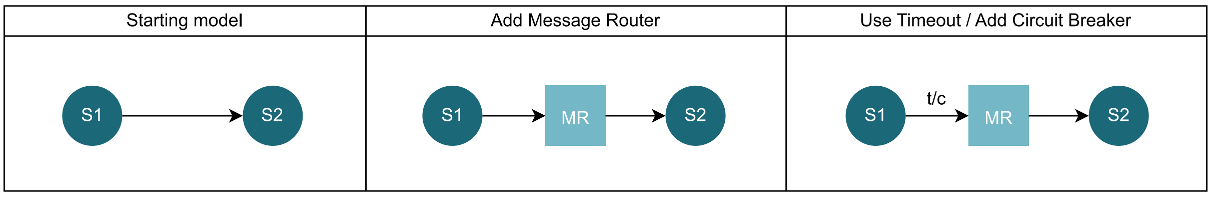


Figure 6.8: Example of solution of the smell between two Service nodes

6.4.1 Add Circuit Breaker

The *Wobbly Service Interaction* smell is detected on the Y node in a X$\longrightarrow$Y interaction, and μFreshener++ considers only cases when this node is a MessageRouter, as described before. The tool extracts the MessageRouter Y from the smell and retrieves the correspondent Kubernetes Service resource, on which the circuit breaker needs to be defined. To define a circuit breaker for a Kubernetes Service is used the Istio DestinationRule resource, following the configuration outlined in Istio docs.[9] The generation of the DestinationRule is performed by using the K8s_template module class, and after its generation is ready to be deployed for defining the circuit breaker to all communication directed to the Service. This strategy is summarized in Figure 6.9, where are illustrated the modifications made to the μTOSCA topology graph and Kubernetes deployment files. The DestinationRule defines circuit breaker rules for relations directed to Y, and

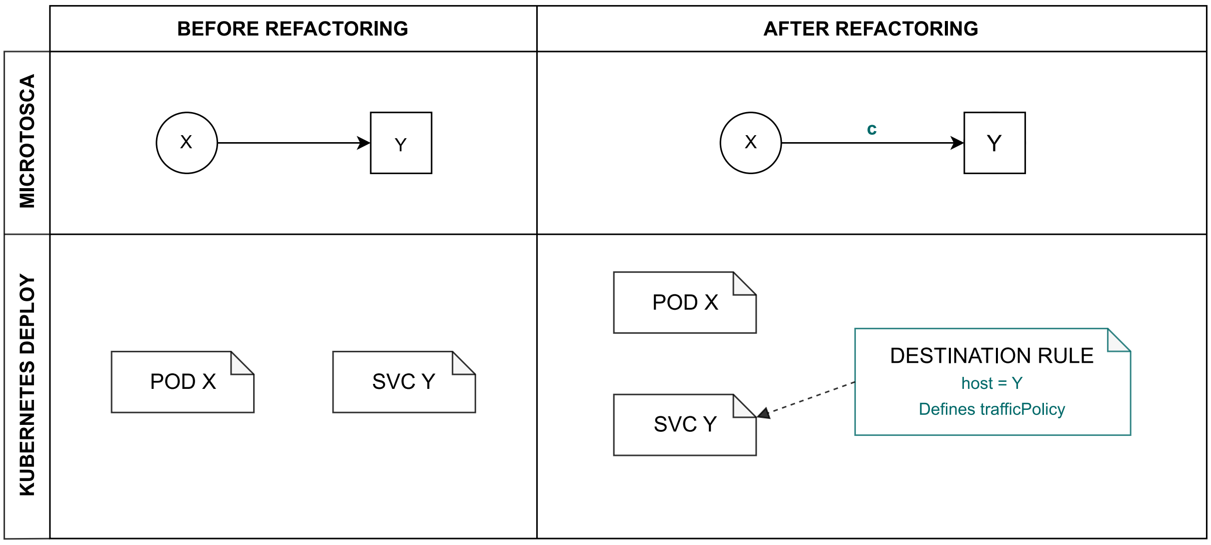


Figure 6.9: Refactoring on μTOSCA model and Kubernetes deployment files applied by Add Circuit Breaker

for representing this is the μTOSCA model all interactions $X_i \longrightarrow Y$ are enriched with the *circuit_breaker* property. An example of generated Kubernetes resource is described in Listing 6. An important property defined by the DestinationRule resource is *host*, which indicates the Kubernetes Service that this rule applies to. In this case, the value *Y.default* is the name.namespace of the Kubernetes Service on which the *Wobbly Service Interaction* smell was detected (or better, the smell is found on the Message Router that represents it). The circuit breaker is configured through the definition of the *trafficPolicy* property, which defines the *connectionPool* and *outlierDetection* properties, that respectively control the volume of connection to a Kubernetes Service and ejection of pods from the ones handled by the Service based on errors recorded. The configuration of the *trafficPolicy* is described in Istio docs.[10] It is also important to note that the values assigned to the various DestinationRule fields are not fixed in the code, but they can be configured directly via a dedicated configuration file. When this resource is deployed within the Kubernetes cluster, Istio applies the rules by configuring the circuit breaker for the specified Y service.

[9]Istio: Circuit Breaking
[10]Istio: DestinationRule trafficPolicy

This technique is an incomplete refactoring because, although the circuit breaker mechanism can be defined for communication, to complete it the developer must modify the X microservice to handle any error that the circuit breaker returns to it.

```
apiVersion: networking.istio.io/v1alpha3,
kind: DestinationRule,
metadata: {
    name: Y.default-circuitbreaker-mf
    namespace: default
},
spec: {
    host: Y.default,
    trafficPolicy: {
        connectionPool: {
            tcp: {
                maxConnections : 5
            },
            http: {
                http1MaxPendingRequests: 1,
                maxRequestsPerConnection: 1
            }
        },
        outlierDetection: {
            consecutive5xxErrors: 5,
            interval: 1s,
            baseEjectionTime: 3m,
            maxEjectionPercent: 1
        }
    }
}
```

Listing 6: Example of DestinationRule resource generated for defining a circuit breaker

6.4.2 Use timeout refactoring

As for *Add Circuit Breaker* refactoring, for applying *Use Timeout* μFreshener++ extracts the MessageRouter Y from the smell and retrieves the corresponding resource Kubernetes Service Y for which the timeout needs to be set. To provide a timeout in a communication, the Istio VirtualService resource is used, which is configured as described in Istio documents.[11] This resource is generated by the K8s_template module, and once created and deployed, the timeout is set for all communications directed to the service. The procedure just described is outlined in Figure 6.10, where both the changes to the μTOSCA model and the Kubernetes deployment files are described.

A VirtualService resource is introduced in the Kubernetes deployment, which defines a timeout for all incoming messages directed to Y and specified that is applied on the

[11]Istio: Request Timeouts

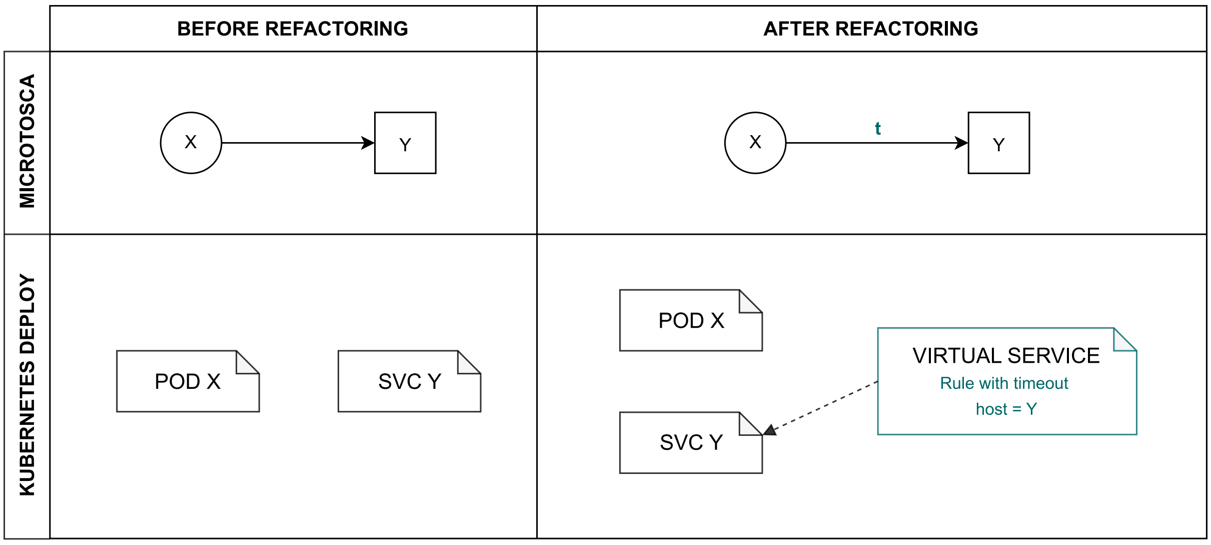


Figure 6.10: Refactoring on μTOSCA model and Kubernetes deployment files applied by Use Timeout

Y Kubernetes Service. For representing that in the μTOSCA model, all interactions $X_i \longrightarrow Y$ (X_i generic component of the system) are enriched with the *timeout* property. The VirtualService is generated using the functions of the K8s_template module, and an example of the generated resource can be observed in Listing 7.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: Y.default-timeout-mf
  namespace: default
hosts: [Y.default]
spec:
  http:
  - route:
    - destination:
        host: Y.default
    timeout: 2s
```

Listing 7: Example of VirtualService resource generated for setting a timeout

The resource defines the *Y.default* value for the *hosts* property, which matches the *destination.host* value, and configuring it this way defines the timeout for all incoming connections directed to the *Y.default* Kubernetes Service. Setting the timeout only requires specifying a value for the *timeout* property when defining a *rule*. When this resource is deployed within the Kubernetes cluster, Istio applies the rules by configuring the specified timeout for the indicated *Y* Service.

Even this technique is an incomplete refactoring because, although the timeout is defined for the communication, to complete the refactoring the developer must modify the *X* microservice to handle any error that it receives when the timeout expires.

6.5 Shared Persistence

The SP smell is present in a microservice system when more than one microservice accesses and manages the same datastore, violating the Decentralisation principle that supports decentralization in microservices systems [19].

The representation of this smell in μTOSCA models is outlined in Figure 6.11, along with the refactoring techniques that can be applied to eliminate it [19]. The *Shared Persistence* smell occurs when two or more microservices share the same datastore, and the smell is present in the μTOSCA model when a Datastore has more than one incoming interaction from a Service. To solve the *Shared Persistence* smell, several refactoring

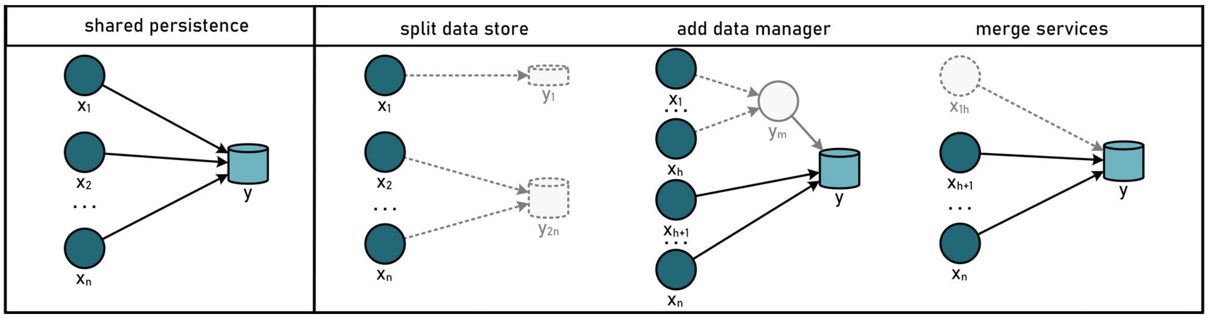


Figure 6.11: Representation of the smell and possible refactoring in μTOSCA models [27]

techniques are applicable, each aimed at ensuring that the database is used by only one microservice. The refactoring *Split Datastore* involves splitting the database into several smaller databases, each used by a different microservice. Each database must contain only the data used by the microservice using it. Another solution is *Add Data Manager*, which implies the inclusion of a data manager, a microservice responsible for accessing the database. Any microservice that needs to access the database must pass through the data manager, which is the only one that has access to the data store, thus avoiding multiple accesses. Finally, *Merge Services* consists of merging microservices using the same database, with the idea that microservices that need to access the same data perform similar tasks and therefore could be grouped.

For the refactoring techniques presented, it was not possible to develop algorithms capable of automatically applying them. The contribution of the tool would have been very limited in all three cases since their application is more a "manual" work performed by the developer that an operation that can be performed automatically by the μFreshener++ tool.

6.6 Summary

This chapter presented the refactoring tasks performed by μFreshener++, which can automatically or semi-automatically solve architectural smells in microservices [19]. For each detected smell, we discussed all possible refactoring techniques that can be applied, highlighting which ones can be implemented by μFreshener++ and which ones cannot. Then, we provided a detailed explanation of each refactoring technique implemented in μFreshener++, outlining the strategy adopted and showing examples of Kubernetes resources generated by μFreshener++. Of the five smells considered, only four could be solved, as no automatic refactoring solutions were found for the *Shared Persistence* smell.

Table 6.1 lists the architectural smells detected by the μFreshener tool, along with the corresponding refactoring strategies implemented in μFreshener++. The table also indicates whether each strategy requires additional manual effort by the developer to be considered fully implemented. A total of five refactoring techniques can be applied

Smell	**Refactoring implemented**	**Full automatic**
No API Gateway	Add API Gateway	N
Endpoint-based Service Interactions	Add Message Router	N
Shared Persistence	-	-
Wobbly Service Interaction	Add Circuit Breaker	N
Wobbly Service Interaction	Use Timeout	N
Multiple Services in One Pod	Split Services	Y

Table 6.1: Summary of refactoring techniques implemented

through μFreshener++. Only one of them is fully automatic (*Split Services*), while the other four require a (limited) effort from the developer to perform the refactoring.

This chapter has demonstrated the significant contribution of the μFreshener++ tool to the toolchain. By using the refactoring feature described, it is becomes possible to refactor the application directly, rather than just the model as with the μFreshener tool. Therefore, the introduction of this new feature aims to enhance the refactoring capabilities of the toolchain.

7 Results over Microservices Smell Detection and Refactoring

This chapter presents the validation we performed to evaluate the functionalities implemented in the μFreshener++ and μTOSCA tools. A first validation was performed during the development phase by introducing multiple unit tests in the μFreshener and μFreshener++ projects, which covered the developed functionalities. These tests were designed before implementing the tested features, applying the Test Driven Development strategy [1].

To perform the validation, we executed the μFreshener++ and μFreshener tools on two different open-source microservice-based projects (Section 7.1). The first analysis conducted regards the results obtained on the *Multiple Services in One Pod* smell detection, developed as an extension of the μFreshener tool (Section 7.2). Then, the results on the μFreshener++ extension task are analyzed (Section 7.3), by presenting the impact that its execution has on smell detection. Finally, we discuss the results obtained from the execution of the μFreshener++ refactoring functionality (Section 7.4), presenting in-depth some of the refactoring applied by the tool.

Model terminology For understanding the subsequent analysis, we introduce the required terminology related to the μTOSCA models (Figure 7.1) used during this analysis. The *Mined Model* is the μTOSCA model extracted from the mining tools of the μTOSCA

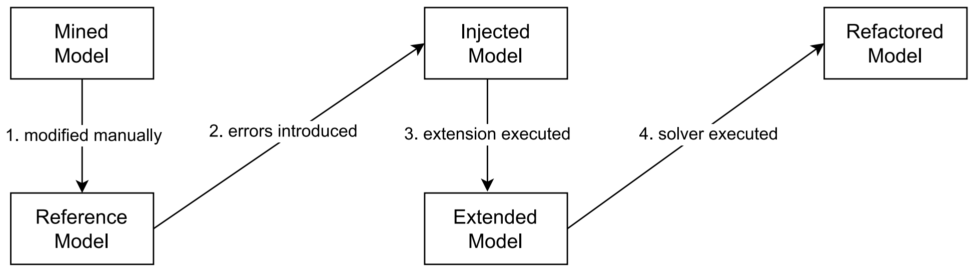


Figure 7.1: Terminology of μTOSCA models

toolchain (μMiner or μTOM). This model is then modified manually to produce the exact representation of the system components, the *Reference Model*. The *Reference Model* model is essential because it is used as a reference during the extender validation. During the validation of the μFreshener++ extension, some errors are introduced in this model, resulting in the *Injected Model*. The extension implemented by μFreshener++ is then executed on this model, which produces the *Extended Model*. Finally, the μFreshener++ refactoring is executed on the *Extended Model* to produce the *Refactored Model*, which is exported at the end of the execution and represents the system after the refactoring.

Table notation Several tables were presented in the analyses conducted and described in this chapter, and we will now present the notation used to refer to nodes in them:

- The name $example_{svc}$ denotes the Service node named *example*
- The name $example_{mr}$ denotes the MessageRouter node named *example*
- The name $example_{cmp}$ denotes the Compute node named *example*

7.1 Analyzed Projects

We validated μFreshener++ and μFreshener functionalities on two different open-source microservice-based applications, namely *MFDemo* and *Sock Shop*, both available on GitHub [18][24].

MFDemo is a project designed for validating the tools, which contains the μTOSCA definition of a demo system and its Kubernetes deployment. The application consists of 9 different microservices with 2 data stores, resulting in 32 nodes in the μTOSCA model and 25 Kubernetes resources defined. The other project considered is Sock Shop, a microservice-based demo application representing the user-facing part of e-commerce selling different types of socks. This system consists of 8 different microservices and 4 data stores, resulting in 26 nodes present in the graph and 29 Kubernetes resources defined. The μTOSCA definition of this system was produced using the μMiner tool.

7.1.1 MFDemo

MFDemo is the demo project developed for performing this validation task. Since the tools work by analyzing the μTOSCA model and Kubernetes resource, this project is not coded and contains only these artifacts. The *Reference Model* of MFDemo is outlined by Figure 7.2.

The system consists of nine microservices and two databases, whose names range from letter *A* to *N*. Each resource is provided by a different Compute node except for *F* and *G* services, which share the same Compute *FG* and so are provided by the same component. Communication between microservices is handled by message routers, with nine out of eleven microservices (including data stores) that have a message router exposing them. The only exceptions are *H* and *M*, with which *B* communicate directly, and *E*, which is directly accessible from external clients. In addition to *E*, there are four other access points to the system represented by as many message routers (*ingress-A*, *ingress-FG*, *gateway-BC*, *N*). The number of message routers and exposed services is very large concerning the application due to testing reasons.

The purpose of this project is to validate the functionalities of μFreshener++. In order to achieve this, specific architectural smells for microservices [19] were deliberately introduced during the design phase of the system. Table 7.1 provides an overview of all the smells introduced for this project.

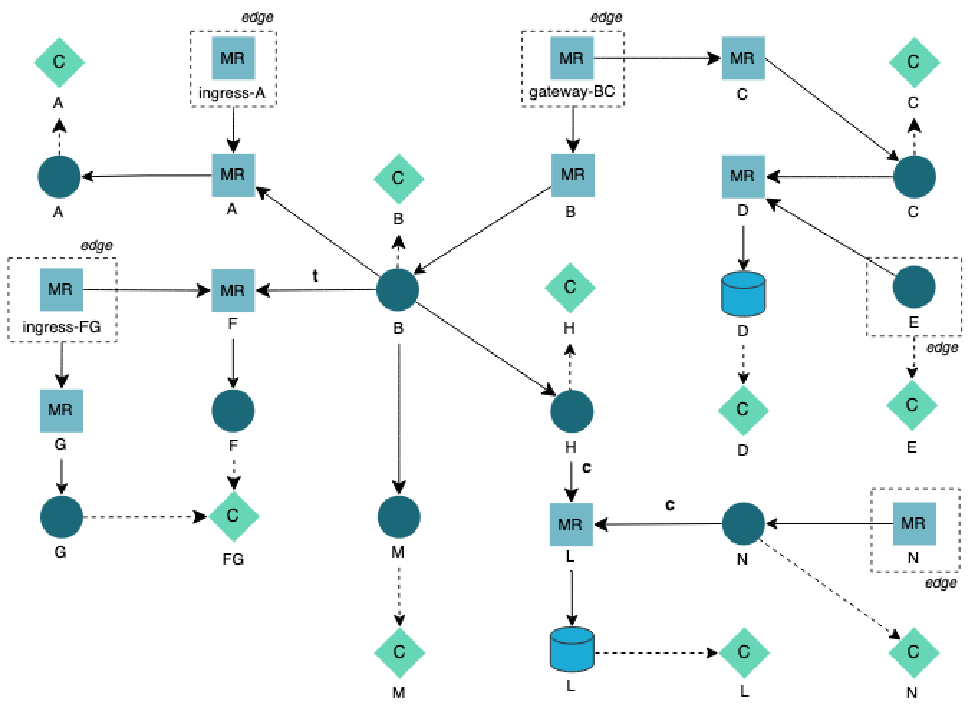


Figure 7.2: MFDemo Reference model

Smell	**Detected on**	**Caused by**
Endpoint-based Service Interactions	H_{svc}	$B_{svc} \longrightarrow H_{svc}$
Endpoint-based Service Interactions	M_{svc}	$B_{svc} \longrightarrow M_{svc}$
Multiple Services in One Pod	FG_{cmp}	$F_{svc} \longrightarrow FG_{cmp}$ $G_{svc} \longrightarrow FG_{svc}$
No API Gateway	E_{svc}	-
Wobbly Service Interaction	C_{svc}	$C_{svc} \longrightarrow D_{mr}$
Wobbly Service Interaction	E_{svc}	$E_{svc} \longrightarrow D_{mr}$
Wobbly Service Interaction	B_{svc}	$B_{svc} \longrightarrow A_{mr}$ $B_{svc} \longrightarrow H_{mr}$ $B_{svc} \longrightarrow M_{mr}$

Table 7.1: Smells present in MFDemo project

7.1.2 Sock Shop

Sock Shop [24] is a microservice application that simulates the user-related part of an e-commerce system for selling socks. To retrieve the μTOSCA model of Sock Shop, the μMiner tool was used, which generated the *Mined Model* outlined by Figure 7.3.

The system consists of eight microservices and four databases, two represented with the Datastore type and the other two (wrongly) using Service nodes. All components discussed above are exposed by a dedicated Message Router, and a Message Broker *rabbitmq* node is also present in the graph. It is important to note that two nodes are missing in the *Mined Model*, as the μMiner did not represent the Kubernetes Pod and Service called *session-db*.

The analysis of the μTOSCA model using μFreshener revealed the presence of only

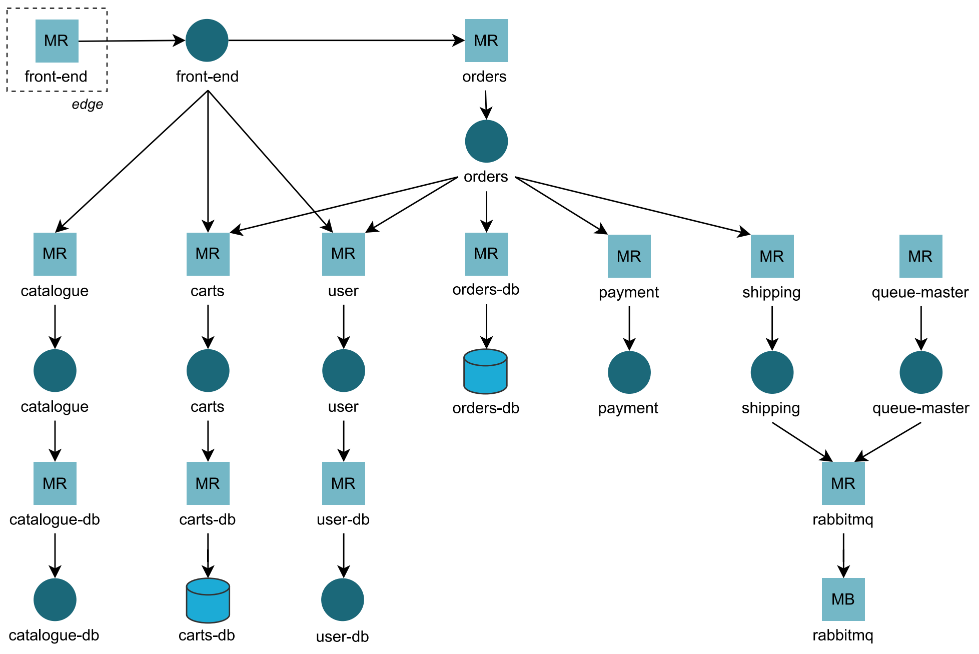


Figure 7.3: Sock Shop Mined model (generated by μMiner tool)

seven instances of the *Wobbly Service Interaction* smell in the system. To conduct a meaningful study on the refactoring feature of μFreshener++, it was necessary to deliberately introduce instances of architectural smells into the system. Below are the changes made to the system for injecting instances of each architectural smell:

1. Regarding *Wobbly Service Interaction*, it was not required to introduce more instances, because it is already well represented in the system with seven smells detected

2. Considering *Endpoint-based Service Interactions*, two instances were included in the application, eliminating a MessageRouter node from the μTOSCA model and the Kubernetes Service represented by it from the Kubernetes resources. The first Message Router removed is *payment*, representing a Kubernetes ClusterIP Service exposing the *payment* Pod. With its deletion, a direct interaction between *orders* and *payment* Service nodes were introduced, and the files representing the Kubernetes Service were canceled. The same procedure was carried out for *shipping* MessageRouter.

3. As for *No API Gateway*, only one instance can be introduced in the application, by removing the *front-end* Kubernetes Service from the deployment and its representing MessageRouter from the model. After removing them, a new entry point for the system was introduced by modifying the *front-end* Pod resource for defining an *hostPort*. The corresponding Service node is added to the Edge.

4. Concerning *Multiple Services in One Pod*, two instances were included in the system, by changing Kubernetes Pod resources for making them define more than one Container. The *carts-db* and *catalogue-db* pods were merged for generating the new

catalogue-carts-db Pod. The old pods were deleted. Kubernetes services exposing them were also merged, creating a new *catalogue-carts-db* Service represented by a MessageRouter node in the model. The other instance was included in the application by using a different strategy. The *payment* Pod was modified for making it define another container called *test*.

5. For the *Shared Persistence* smell, on the other hand, no automatic refactoring technique could be defined, so it was not necessary to introduce any instance of it into the application.

All the architectural smells present in the Sock Shop project are listed in Table 7.2. The first five rows correspond to the smells that were intentionally injected into the system, as described earlier. Injected smells in the table are marked with an asterisk next to the smell name.

Smell	**Node**	**Caused by**
EB*	payment$_{svc}$	orders$_{svc}$ $\longrightarrow$ payment$_{svc}$
EB*	shipping$_{svc}$	orders$_{svc}$ $\longrightarrow$ shipping $_{svc}$
MS*	orders$_{cmp}$	orders$_{svc}$ $\longrightarrow$ orders$_{cmp}$ test$_{svc}$ $\longrightarrow$ orders$_{cmp}$
MS*	catalogue-carts-db$_{cmp}$	catalogue-db$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{cmp}$ carts-db$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{cmp}$
NG*	front-end$_{svc}$	-
WS	carts$_{svc}$	catalouge-carts-db$_{svc}$ $\longrightarrow$ carts$_{mr}$
WS	catalogue$_{svc}$	catalogue$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{mr}$
WS	front-end$_{svc}$	front-end$_{svc}$ $\longrightarrow$ catalogue$_{mr}$ front-end$_{svc}$ $\longrightarrow$ carts$_{mr}$ front-end$_{svc}$ $\longrightarrow$ orders$_{mr}$ front-end$_{svc}$ $\longrightarrow$ user$_{mr}$
WS	orders$_{svc}$	orders$_{svc}$ $\longrightarrow$ carts$_{mr}$ orders$_{svc}$ $\longrightarrow$ orders-db$_{mr}$ orders$_{svc}$ $\longrightarrow$ payment$_{mr}$ orders$_{svc}$ $\longrightarrow$ user$_{mr}$ orders$_{svc}$ $\longrightarrow$ shipping$_{mr}$
WS	queue-master$_{svc}$	queue-master$_{svc}$ $\longrightarrow$ rabbitmq$_{mr}$
WS	shipping$_{svc}$	shipping$_{svc}$ $\longrightarrow$ rabbitmq$_{mr}$
WS	user$_{svc}$	user$_{svc}$ $\longrightarrow$ user$_{mr}$

Table 7.2: Smell present in Sock Shop project

Legend: **EB** Endpoint-based Service Interaction smell **MS** Multiple Services in One Pod smell **NG** No Api Gateway smell **SP** Shared Persistence smell **WS** Wobbly Interaction Service smell

After presenting errors in the *Mined Model* and changes made for introducing the smells, the *Reference Model* is introduced (Figure 7.4). This model includes all changes made for the introduction of smells, missing nodes related to session-db, and all Compute nodes.

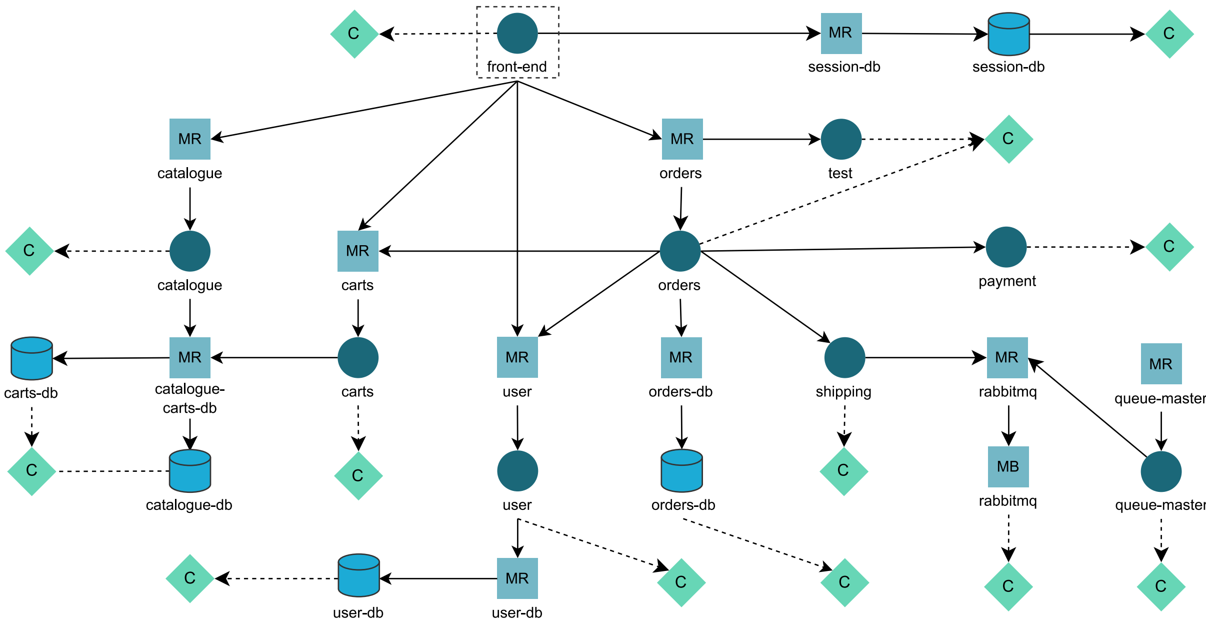


Figure 7.4: Sock Shop Reference model

7.2 Results on Multiple Services in One Pod smell detection

The first analysis performed regards the results obtained by running the new smell detector implemented in μFreshener for *Multiple Services in One Pod* smell identification. We executed the detection on the previously presented projects, MFDemo and Sock Shop, and on two demo projects implementing Kubernetes patterns, viz., Adapter-demo and Sidecar-demo. Adapter-demo and Sidecar-demo contain only a single Pod that implements the Adapter and Sidecar multi-container design pattern of Kubernetes [15], respectively.

We considered including other projects in this analysis to verify the new detector, but most of the demo applications on GitHub adopt the *one-container-for-pod* [16] model, so it is difficult to find applications where the *Multiple Services in One Pod* smell is present.

From the results obtained from the μFreshener smell detection executed on the projects, we extracted only the instances of *Multiple Services in One Pod*. This smell was recognized directly on the Compute node C_i that deploys multiple microservices. For each instance found, we manually checked the resource represented by C_i to verify that the identified smell was actually present in the system. The instances found are listed in Table 7.3, where we can see that in Sock Shop, two instances were found (injected earlier, Section 7.1.2) while there is only one for every other project.

As for MFDemo, the smell was detected on the Compute FG node (FG_{cmp}) and, by analyzing the related Kubernetes resource (Listing 8), we verified how indeed this Kubernetes Deployment defines two containers. Being a demo project these containers have no defined image but, considering that in the development of this application F and G were thought of as separate microservices, we could consider this smell instance as identified correctly on this resource.

Project	**Node**	**K8s resource**
MFDemo	FG_{cmp}	*FG* Deployment
Sock Shop	$orders_{cmp}$	*orders* Deployment
Sock Shop	$catalogue\text{-}carts\text{-}db_{cmp}$	*catalogue-carts-db* Deployment
Adapter-demo	*multi-adapter*$_{cmp}$	*multi-adapter* Pod
Sidecar-demo	*multi-sidecar*$_{cmp}$	*multi-sidecar* Pod

Table 7.3: Instances of *Multiple Services in One Pod* smell found in the two projects

```
 kind: Deployment
 metadata:
   name: FG
 ...
   template:
     spec:
       containers:
       - name: F
         ...
       - name: G
         ...
```

Listing 8: FG Kubernetes Deployment of MFDemo project

Considering Sock Shop, two instances were identified on the Compute nodes *orders* and *catalogue-carts-db* (Listing 9). Deployment *orders* defines two containers within it running two separate microservices, while *catalogue-carts-db*, the Deployment defines two containers that run two different database instances.

```
kind: Deployment
metadata:
  name: orders
...
  template:
    spec:
      containers:
      - name: test
        image: nginx
        ...
      - name: orders
        image: orders:0.4.7
        ...
```

```
kind: Deployment
metadata:
  name: catalogue-carts-db
...
  template:
    spec:
      containers:
      - name: catalogue-db
        image: catalogue-db:0.3.0
        ...
      - name: carts-db
        image: mongo
        ...
```

Listing 9: orders and catalogue-carts-db Kubernetes Deployment of Sock Shop project

In both cases, the resources run two microservices independent of each other, which should be separated to enable them to be independent. The *Multiple Services in One Pod* smell was so correctly detected by μFreshener in both cases.

Analyzing the smells identified on Adapter-demo and Sidecar-demo, we can see that the smells were identified on resources that implement Kubernetes design patterns. These two instances are considered as false positives, since the two pods *multi-adapter* and *multi-sidecar* does not run separate microservices. The resources on which the smell was found are shown in Listing 10, where both run the *nginx:alpine* container with an execution support container (*adapter* or *sidecar*).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-adapter
spec:
   containers:
  - image: nginx:alpine
    name: backend
    ...
  - image: nginx-prometheus-exporter
    name: adapter
    ...
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-sidecar
spec:
   containers:
  - image: nginx:alpine
    name: frontend
    ...
  - image: git-sync:v3.1.6
    name: sidecar
    ...
```

Listing 10: multi-adapter and multi-sidecar resources of Adapter-demo and Sidecar-demo projects

Summarizing the results obtained, μFreshener detected five instances of *Multiple Services in One Pod*, of which three were correctly identified as smells based on analysis of the deployment files. The remaining two instances were identified as false positives, as they were found to be an application of Kubernetes patterns [15]. However, this is a known problem with the developed detector (discussed earlier, Section 5.1), and these cases were included in the validation precisely to show the detection of these false positives.

7.3 Results on Extension

This analysis aims to verify whether the μFreshener++ tool is capable of performing the extension of the μTOSCA model. To achieve this, the μTOSCA *Reference Model* was modified to introduce some discrepancies between it and the Kubernetes deployment files. The modified model is referred to as the μTOSCA *Injected Model*. Next, we executed the μFreshener++ extension on this model to generate the μTOSCA *Extended Model*, which was then compared with the μTOSCA *Injected Model* and the μTOSCA *Reference Model*. The comparison involved two checks: (1) we verified that all discrepancies introduced in the *Injected Model* were corrected in the *Extended Model* and (2) we ensure that the *Extended Model* was equivalent to the *Reference Model*. After validating the performance of the extension of μFreshener++ on both projects (Section 7.3.2, 7.3.3), we conducted a small study on the impact that the extension had on the μFreshener smell detection (Section 7.3.4).

7.3.1 Methodology

This section outlines the methodology employed for conducting the analysis, in order to provide clarity regarding the steps taken during its execution. To describe the methodology as clearly as possible, it is presented as a numbered list, and this numbers are then used in subsequent sections as a reference to the stage described. The methodology is divided into the following stages:

1. The first step involves making changes to the *Reference Model* to create discrepancies between it and the Kubernetes deployment. Each change is recorded in the *Change Table*, which also specifies the worker that is intended to trigger the *change*. Each *change* introduces an *issue*, representing the discrepancy between the *Reference Model* and the Kubernetes deployment. All issues introduced are documented in the *Issue Table*.

2. The *changes* made to the *Reference Model* generate the *Injected Model*, which is then examined to visualize and analyze the described changes.

3. At this stage, the *Extended Model* produced after running μFreshener++ on the *Injected Model* is presented. This is the μTOSCA model modified by the μFreshener++ extension to remove all *issues* present. In the *Extended Model*, yellow nodes indicate the changes made by μFreshener++ to remove the *issues*. The dashed outline around the yellow nodes indicates that the node is modified by μFreshener++, while nodes with "normal" outlines are added by the tool to the μTOSCA model.

4. When μFreshener++ identifies an *issue*, it implements changes to the model to resolve it, and these changes are referred to as *solutions* in this analysis. Each *solution* applied by μFreshener++ is recorded in the *Solution Table*, which includes information about the resolved *issue* (if any) and the worker that was successfully tested. To achieve a complete correction of the model, all identified *issues* must be resolved by the tool.

5. The final check involves verifying the equivalence between the *Extended Model* and the *Reference Model* to confirm whether the tool has successfully restored the model to its original state. To conduct this verification, the Diffchecker tool[12] was utilized to analyze the differences between the YAML definitions of the *Reference Model* and the *Extended Model*.

The methodology described above was applied to analyze the impact of the μFreshener++ extension feature on two projects: MFDemo (Section 7.3.2) and Sock Shop (Section 7.3.3).

[12] `https://www.diffchecker.com/`

Controls Before presenting the application of this methodology on the projects, it is necessary to provide a summary of all the *controls* carried out by the μFreshener++ while applying the extension. All the workers developed for implementing the extension (Section 5.2) have been listed into Table 7.4, which outlines each worker's specific *control*. For instance, the container worker *adds a Service to the Edge* if it finds that it defines a *hostNetwork* (CT01) or a *hostPort* (CT02), and also *removes a Service from the Edge* if it does not define any of these properties (CT03).

Worker	**Ref**	**Control**
Container	CT01	Add Service to Edge
	CT02	Add Service to Edge
	CT03	Remove Service from Edge
Compute	CM01	Add compute nodes
Database	DB01	Convert Service to Datastore
Ingress	IN01	Add missing ingress
	IN02	Add missing interactions
Istio Timeout	IT01	Add missing timeout
Istio Circuit Breaker	IC01	Add missing circuit breaker
Istio Gateway	IG01	Add missing gateway
MessageRouter Edge	MR01	Add Message Router to Edge
	MR02	Remove Message Router from Edge
Service	SV01	Add missing Service
	SV02	Convert Service to Message Router
	SV03	Change interactions

Table 7.4: Control performed by the μFreshener++ workers

7.3.2 Results on MFDemo

The first project on which the extension results are discussed is MFDemo, whose corresponding *Reference Model* can be observed in Figure 7.2.

(1) By manually modifying the μTOSCA *Extended Model*, we applied a total of 11 *changes*, each triggering a different extender *control* (Table 7.4). The only *control* tested two times is DB01, for verifying that the Database worker could detect the container running a database from both port and image used. The *changes* applied to this model regard for example edits to Edge group members, or Datastore nodes converted to Service, and are all enumerated in Table 7.5. Each *change* applied to the model introduced an *issue* into it. For example, the change C01 removed node E from the Edge group, but this node is a container defined within a Pod that has the property *hostNetwork=true.* In this case, C01 introduced an *issue*, because the container represented by E is directly reachable by external clients but is not included in the Edge group. All *issues* introduced by the *changes* applied are organized in Table 7.6, where we can observe that the 11 *changes* made to the model introduced 13 *issues* (in the *Injected Model*).

ID	Changes	Control
C01	E_{svc} removed from Edge	CT01
C02	All compute nodes deleted	CM01
C03	D_{ds} converted to Service	DB01
C04	L_{ds} converted to Service	DB01
C05	ingress-A_{mr} deleted	IN01
C06	Interaction ingress-$FG_{mr} \longrightarrow G_{mr}$ deleted	IN02
C07	Timeout removed from interactions directed to F_{mr}	IT01
C08	Circuit breaker removed from interactions directed to L_{mr}	IC01
C09	gateway-BC_{mr} deleted	IG01
C10	N_{mr} removed from Edge	MR02
C11	L_{mr} deleted	SV01

Table 7.5: MFDemo changes applied to the model

ID	Issue introduced	Change
I01	E_{svc} has hostNetwork property set but is Edge member	C01
I02	All compute nodes missing	C02
I03	D_{svc} represents a database (Pod exposes MySQL port)	C03
I04	L_{svc} represents a database (Pod runs MySQL image)	C04
I05	Kubernetes ingress-A not represented	C05
I06	A_{mr} is in the Edge with type=ClusterIP	C05
I07	Missing interaction ingress-$FG_{mr} \longrightarrow G_{mr}$	C06
I08	A DestinationRule defines timeout for F_{svc}	C07
I09	A VirtualService defines circuit breaker for L_{svc}	C08
I10	B_{mr} and C_{svc} are in the Edge with type=ClusterIP	C09
I11	Istio Gateway gateway-BC not represented	C09
I12	N_{mr} is not in the Edge with type=NodePort	C10
I13	Kubernetes Service L not represented	C11

Table 7.6: MFDemo Issue table

(2) Considering all modifications listed, the *Reference Model* underwent many changes that modified it, and after applying all of them we obtained the *Injected Model* (Figure 7.5). Summarizing the differences between the *Injected Model* and *Reference Model*, we can state that: (A) 2 nodes are not present in the Edge, Service E and MessageRouter N, and 4 are wrongly included in the group, message routers A, B, C, and Service M (B) 10 Compute nodes are missing (C) the interaction $ingress\text{-}FG \longrightarrow G$ is missing (D) 3 MessageRouter are missing from the graph, $ingress - A$, $gateway - BC$, and L (E) 2 databases are represented using Datastore nodes instead of Service type, nodes D and L (F) 2 timeout and 2 circuit breaker properties are missing.

(3) On the *Injected Model* was then executed the μFreshener++ extension for solving the *issues* presented before. This task produced as output the *Extended Model* model (Figure 7.6), where nodes modified or added by μFreshener++ are highlighted with yellow color. Summarizing the modifications that μFreshener++ applied to the model, 13 nodes had been added, of which 10 are Compute and three are MessageRouter. The nodes

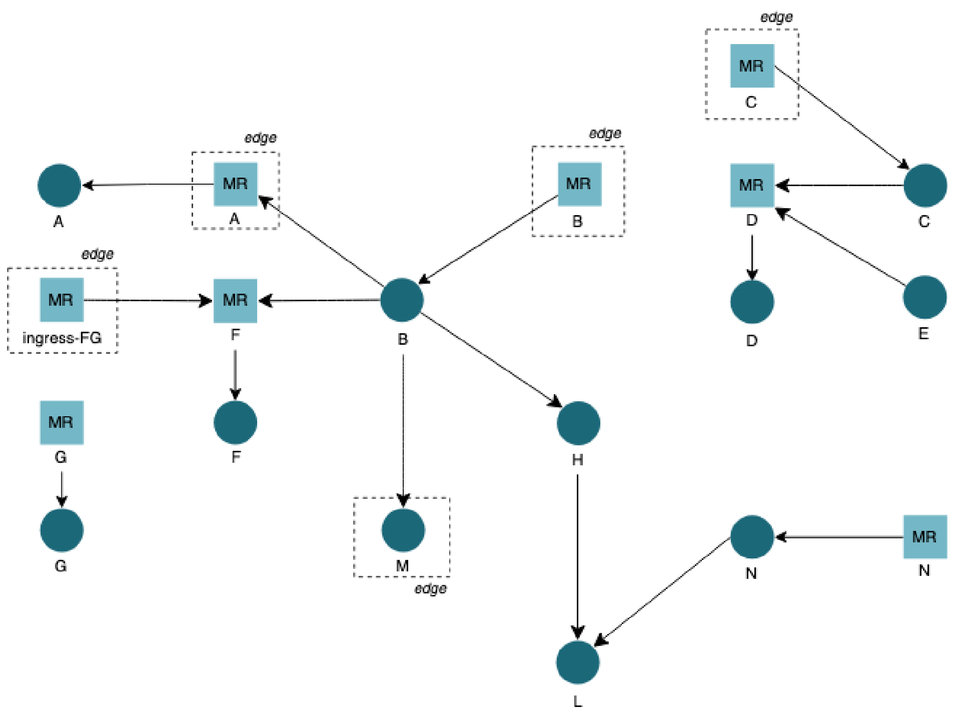


Figure 7.5: MFDemo *Injected Model*

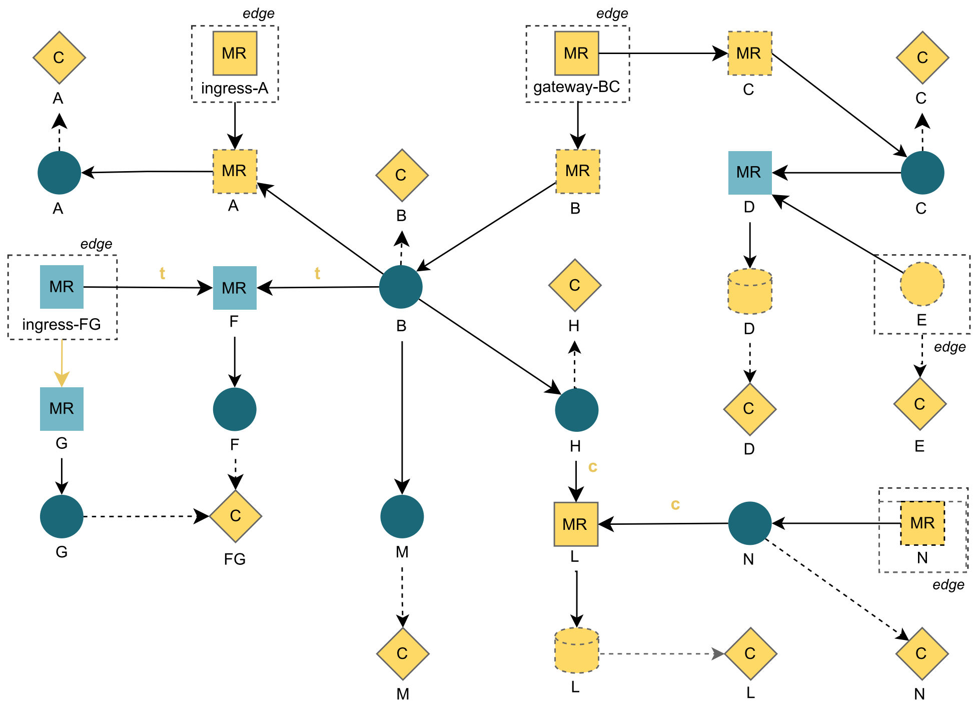


Figure 7.6: MFDemo *Extended Model*

modified are in total 6, with nodes conversion and changes to their presence in the Edge. In addition, were added one interaction and four proprieties, two timeouts, and two circuit_breakers.

(4) Every yellow node in the *Extended Model* represents a solution applied by μFreshener++ for solving one *issue* detected. All *solutions* applied are listed in Table 7.6, where for each is indicated which *issue* this *solution* solves (if found). In total, 13 *solutions* were

Solution	Issue	Control
E_{svc} added to the Edge group	I01	CT01
10 Computes node added	I02	CM01
D_{svc} converted to Datastore type	I03	DB01
L_{svc} converted to Datastore type	I04	DB01
ingress-A represented in the model	I05	IN01
A_{mr} removed from the Edge group	I06	IN01
Interaction ingress-$FG_{mr} \longrightarrow G_{mr}$ added	I07	IN03
Timeout property added to interactions directed to F_{mr}	I08	C07
Circuit_breaker property added to interactions directed to L_{mr}	I09	C08
Istio gateway-BC represented as a Message Router in the Edge	I10	IG01
B_{svc} and C_{svc} removed from the Edge group	I11	IG01
N_{mr} added to the Edge group	I12	MR01
Kubernetes Service L represented in the model	I13	SV01

Table 7.7: MFDemo Solution table

applied by the μFreshener++ tool, each that solved one issue presented before. The extension performed by μFreshener++ was then able to resolve all *issues* in the *Injected Model*, thus verifying the correctness of all *controls* tested by them (shown in Table 7.6).

(5) The last check performed was the analysis of the *Reference Model* (Figure 7.2) and *Extended Model* (Figure 7.6), for verifying that they represent the same components and interactions, i.e., they are identical. To verify this, we utilized the Diffchecker tool to identify any differences between the two YAML definitions of the μTOSCA models, and this analysis showed that the models (viz., *Reference Model* and *Extended Model*) were identical.

This section showed the validation activity performed on the MFDemo project. Thirteen issues were introduced in the *Reference Model* of the application, all of which were removed after running on μFreshener++ extension model (which produced the *Extended Model*). In addition, the *Reference Model* and the *Extended Model* were found to be identical after checking their YAML definitions with Diffchecker. Given these results, it is possible to confirm that μFreshener++ was able to solve all the problems introduced in MFDemo by returning the model to its original form, and thus all the *controls* considered in this analysis operated correctly.

7.3.3 Results on Sock Shop

The other project on which μFreshener++ extension was executed is Sock Shop, whose corresponding *Reference Model* can be observed in Figure 7.4.

(1) All seven *changes* applied to the *Reference Model* are enumerated by Table 7.8, each triggering *controls* both tested (CT03, CM01, SV01) and not tested (CT02, MR02, SV02, SV03) by the MFDemo project. The *changes* applied to the *Reference Model* regard nodes added and removed from the Edge group, components deleted from the graph, and addition of direct interactions among nodes, etc. All issues present in the *Injected Model* are outlined by Table 7.9. Each *change* applied introduced exactly one

ID	Changes	Control
C13	front-end$_{svc}$ removed from Edge	CT02
C14	test$_{svc}$ added to Edge	CT03
C15	All compute nodes deleted	CM01
C16	queue-master$_{mr}$ added to Edge	MR01
C17	rabbitmq$_{mr}$ deleted*	SV01
C18	user-db$_{mr}$ converted to Service	SV02
C19	Interaction orders$_{svc}$ $\longrightarrow$ users$_{svc}$ added	SV03

Table 7.8: Sock Shop Changes table

* This case is slightly different from what was tested with MFDemo because the node has two interactions in ingress instead of one.

issue, with a total of 7 issues (I15-I21) due to *changes* introduced in the model. Other 4 *issues* (I22-I25), however, were not injected in the model for performing this analysis, but were *issues* introduced from the Sock Shop architecture mining (the *Mined Model* is represented in Figure 7.3). In particular, the mining tool modeled two databases using Service nodes and had not included the nodes (relative to *session-db* microservice) into the graph. Since these issues were introduced in the context of real-life use of μTOSCA the toolchain, they were included in the *Injected Model* to verify the behavior of the μFreshener++ in these unexpected cases.

ID	Issue introduced	Change
I15	front-end$_{svc}$ has hostPort property set but is Edge member	C13
I16	Service test is Edge member with no hostNetwork/hostPort defined	C14
I17	All compute nodes missing	C15
I18	queue-master$_{mr}$ is in the Edge with type=ClusterIP	C16
I19	Kubernetes Service rabbitmq not represented	C17
I20	Kubernetes Service *user-db* represented using Service node	C18
I21	Interaction orders$_{svc}$ $\rightarrow$ users$_{svc}$ doesn't pass through the users$_{mr}$	C19
I22	Service catalogue-db represents a database (Pod exposes MySQL port)	*
I23	Service users-db represents a database (Pod exposes MongoDB port)	*
I24	Kubernetes Service session-db not represented	*
I25	Kubernetes Deployment session-db not represented	*

Table 7.9: Sock Shop Issue table

* Problems introduced by the μMiner tool

(2) Considering all modifications listed, the model underwent many changes that modified the represented graph. After applying all described changes to the model, the result obtained is the *Injected Model* (Figure 7.7). Summarizing the differences with the starting model, we can state that: (A) the *front-end* Service is included in the Edge group, and two nodes are missing from it, *test* Service and *queue-master* MessageRouter, (B) the *rabbitmq* MessageRouter is missing from the model, (C) the *user-db* MessageRouter

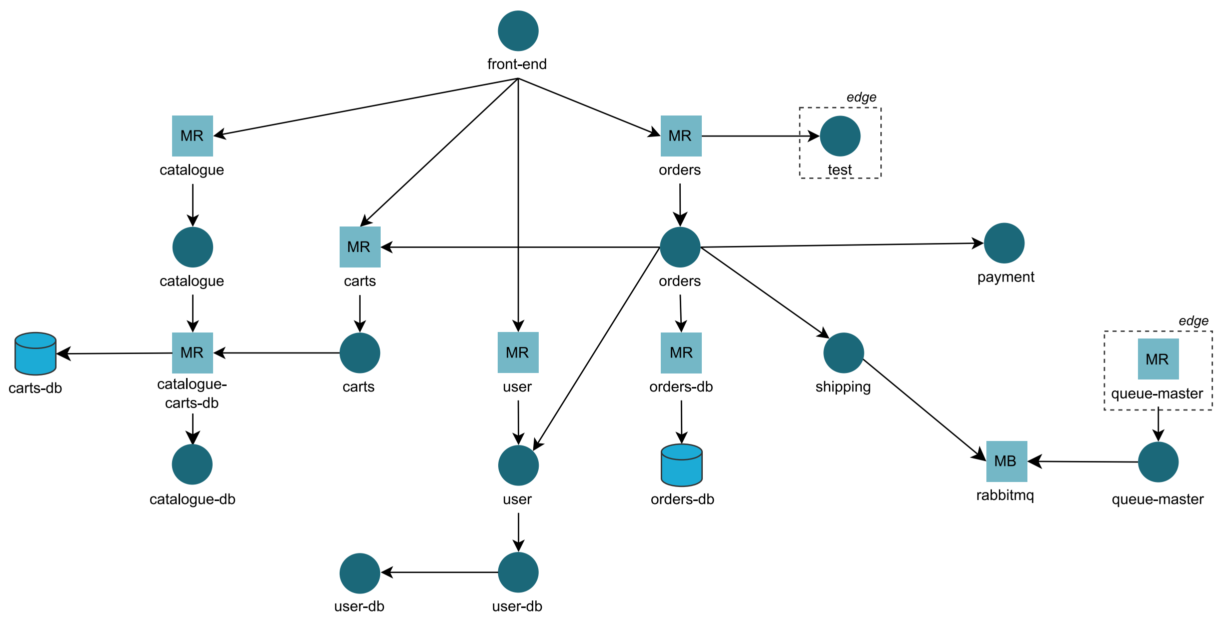


Figure 7.7: Sock Shop injected model

is modeled using a service instead of a MessageRouter, (D) the interaction *orders* → *users* changed its target, (E) all compute nodes are missing from the model, (F) *session-db* MessageRouter and Service node are missing, and (G) two databases *user-db* and *catalogue-db* are represented using Service node.

(3) On the *Injected Model* was then executed the μFreshener++ extension for solving the *issues* presented before. This task produced in the output the *Extended Model* model (Figure 7.8), where nodes modified or added by μFreshener++ are highlighted with yellow color. Summarizing the modifications that μFreshener++ applied to the model, 14 nodes had been added, of which 13 Compute and 1 MessageRouter. 2 nodes had been converted to Datastore, one to MessageRouter, and 2 changed their presence in the Edge group. Finally, new 2 interactions are added to the graph.

(4) Every yellow node represented in the *Extended Model* represents a solution applied by μFreshener++ for solving one (or more) *issue* detected. All *solutions* applied are listed in Table 7.10, where for each is indicated which *issue* this *solution* solves. In total, 9 *solutions* were applied by the μFreshener++ tool, each that solves one issue presented before. The total number of *issues* enumerated in Table 7.9 is 11, and so emerges that two *issues* had not been solved, namely I24 and I25. To recap, the Extender was able to solve 9 of the 11 *issues* introduced.

(5) The last check we performed is the analysis of the *Reference Model* (Figure 7.4) and *Extended Model* (Figure 7.8), for verifying that they represent the same components and interactions, i.e., they are identical. Even from this analysis, emerged that two nodes definition are missing, namely *session-db* MessageRouter and *session-db* Service.

This section showed the validation activity performed on the Sock Shop project. Eleven issues were introduced in the *Reference Model* of the application, and nine out of eleven were solved correctly by μFreshener++. Even the analysis with Diffchecker gave

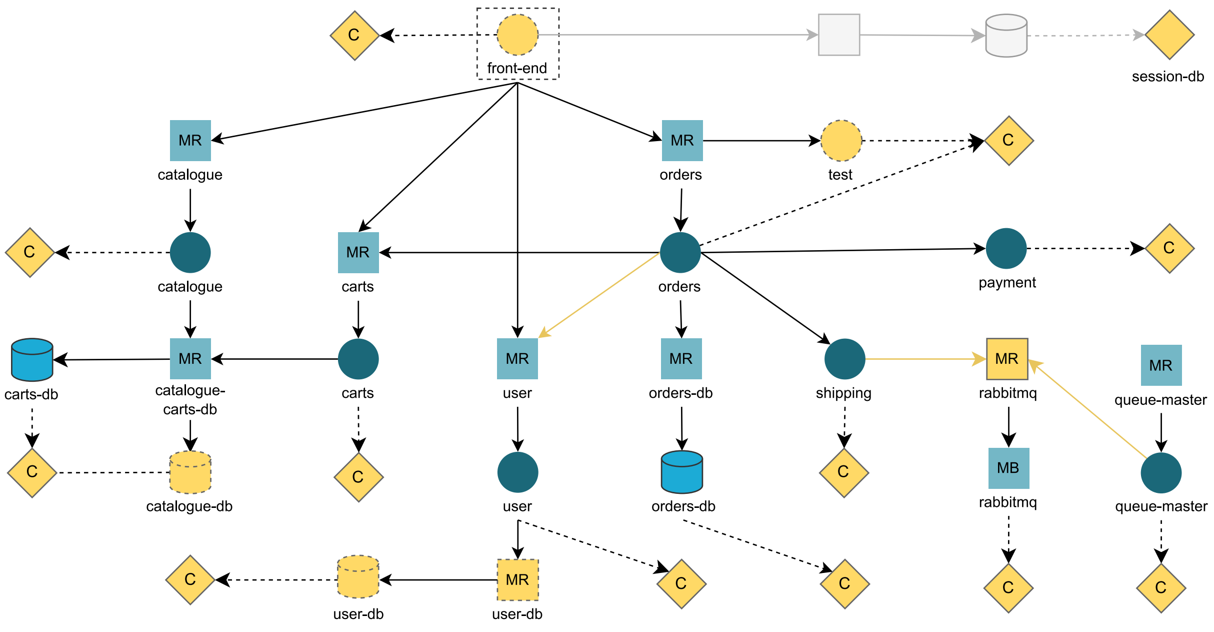


Figure 7.8: Sock Shop Extended model

Solution	**Issue**	**Control**
front-end$_{svc}$ added to the Edge group	I15	CT02
test$_{svc}$ removed from the Edge group	I16	CT03
13 Computes node added	I17	CM01
queue-master$_{mr}$ removed from the Edge group	I18	MR01
rabbitmq$_{mr}$ represented in the model	I19	SV01
user-db$_{svc}$ represented in the modelMessage Router	I20	SV02
Interaction orders$_{mr}$ $\rightarrow$ user$_{mr}$ added	I21	SV03
catalogue-db$_{svc}$ converted to Datastore type	I22	DB01
users-db$_{svc}$ converted to Datastore type	I23	DB01

Table 7.10: Sock Shop Solution table

the same results, with two nodes missing from the model related to *session-db* microservice. The two missing nodes signal that μFreshener++ was unable to represent the system correctly but, however, this is an expected result, as no worker was developed to represent missing Service nodes in the μTOSCA model. Related to that, it is important to underline two aspects: (1) even if the Service node was added to the μTOSCA model, not all the interactions involving it would have been present, and therefore it was a choice not to develop a dedicated worker for the addition of Service nodes (2) the non-addition of the MessageRouter node related to the Kubernetes Service session-db seems to be an error of the SV01 control. In reality, this node also would not have had any interactions, so it was a design choice of the worker not to represent it.

Even if these two nodes were not represented, the *issues* injected manually for testing extension *controls* were solved, demonstrating that *controls* tested by this project can solve the *issues* present in the *Injected Model*.

7.3.4 Impact of extension on smell detection

For analyzing the value of performing the μTOSCA model extension, it is possible to study its impact on the μFreshener smell detection. To conduct this analysis, we executed the μFreshener detection on the *Injected Model* and *Extended Model* of both MFDemo and Sock Shop projects, for studying the differences in instances found after and before running μFreshener++ extension. For each project is presented a Table that summarizes, for each smell instance, if it was found on the *Injected Model* and/or the *Refactored Model*.

Results obtained from MFDemo are outlined in Table 7.11, where we can observe a decrease of two instances found after running the extension (9 smell in *Injected Model* vs. 7 in the *Extended Model*). Considering *Endpoint-based Service Interactions*, the smell on L_{svc} is caused by two direct interactions $H_{svc} \longrightarrow L_{svc}$ and $N_{svc} \longrightarrow L_{svc}$, but Pod L in the Kubernetes deployment is exposed by a Kubernetes Service (not represented in the model). μFreshener++ so adds the MessageRouter for representing it, removing direct interactions and also the false positive detection. The execution of μFreshener++ added the Compute nodes to the μTOSCA model, and this allowed to perform the detection of the *Multiple Services in One Pod* smell on FG_{cmp}. So, μFreshener++ allowed the detection of a true negative instance. Taking into account *No API Gateway*, the smell found on M_{svc} is a false positive because it is a member of the Edge group, but the M Pod does not define any "exposing" property (*hostNetwork* or *hostPort*). μFreshener++ moved out the node from the Edge, removing the detection of this false positive. Considering node E_{svc} it is not a member of the Edge but the E Pod defines *hostNetwork=true.* In this case, μFreshener++ added it to the Edge allowing the detection of a true negative instance. The last smell considered is *Wobbly Service Interaction*, which both regard an interaction

Smell	**Node**	**Injected model**	**Refactored model**
Endpoint-based Service Interactions	H_{svc}	●	●
	M_{svc}	●	●
	L_{svc}	●	
Multiple Services in One Pod	FG_{cmp}		●
No API Gateway	M_{svc}	●	
	E_{svc}		●
Wobbly Service Interaction	B_{svc}	●	●
	C_{svc}	●	●
	E_{svc}	●	●
	H_{svc}	●	
	N_{svc}	●	

Table 7.11: Differences in smell found by analyzing Injected and Extended MFDemo models

directed to Service node L ($H_{SVC} \longrightarrow L_{SVC}$ and $N_{SVC} \longrightarrow L_{SVC}$). μFreshener++ added a MessageRouter in front of L, with all its incoming relation enriched with *circuit_ breaker* property, and so the *Wobbly Service Interaction* is not detected anymore for both smells. Even in this situation, two false positives had been removed from detection.

Results obtained on Sock Shop are outlined in Table 7.12, where emerges that the number of smells detected is the same, but the nodes affected change. After the ex-

Smell	**Node**	**Injected model**	**Refactored model**
Endpoint-based Service Interactions	$payment_{svc}$	●	●
	$shipping_{svc}$	●	●
	$user\text{-}db_{svc}$ (svc)	●	
	$user\text{-}db_{svc}$ (pod)	●	
	$user_{svc}$	●	
Multiple Services in One Pod	$catalogue_{cmp}$		●
	$orders_{cmp}$		●
No API Gateway	$test_{svc}$	●	
	$front\text{-}end_{svc}$		●
Wobbly Service Interaction	$carts_{svc}$	●	●
	$catalogue_{svc}$	●	●
	$front\text{-}end_{svc}$	●	●
	$orders_{svc}$	●	●
	$queue\text{-}master_{svc}$		●
	$shipping_{svc}$		●
	$user\text{-}db_{svc}$ (svc)	●	
	$user_{svc}$	●	●

Table 7.12: Differences in smell found by analyzing Injected and Extended Sock Shop models

tension, three smells in *Endpoint-based Service Interactions* are no longer detected. The solution that led to the removal of two of them involved converting *user-db* from a Service to a MessageRouter, which eliminated two direct interactions between the Services $user \longrightarrow user\text{-}db$ and $user\text{-}db \longrightarrow user\text{-}db$. The other smell that was removed involved the direct interaction $orders \longrightarrow users$, which was corrected by the Extender by passing the interaction through the dedicated MessageRouter, thus removing the direct interaction. All three removed instances were so false positives. The extension performed by μFreshener++ added the Compute nodes to the μTOSCA model, and this allowed the detection of two new *Multiple Services in One Pod* smell on *catalogue* and *orders* Compute nodes, so enabling the identification of two instances not detected before. Even in this case, is detected one instance of *No API Gateway* in both models, but on different nodes. In the Injected the smell is recognized on the *test* node, which is wrongly at the Edge and removed from it by the Extender, and so this change removes the detection of one false positive. After the extension, the smell is detected on the *front-end* Service exposed on the outside, allowing so the detection of one true negative not considered before. Considering the last case, *Wobbly Service Interaction*, the extension removed the instance detected on the *user-db* Service node, related to $user\text{-}db \longrightarrow user\text{-}db$ interaction, simply because *user-db* is converted to Message Router type, and so there is no wobbly interaction between them. The addition of the *queue-master* MessageRouter, additionally, allowed the detection of this smell on two new smells. These two cases can be considered 2 false positives because the MessageRouter exposes one MessageBroker and so there is no wobbly interaction in this case. The introduction of these two false positives is however due to the detection and not related to the extension

The results discussed above are summarized in Table 7.13, where for each project is outlined the number discovered after running μFreshener++ extension regarding false positives removed (Removed FP) and added (Added FP), and true negatives detected (Added TN). Considering aggregated results of two projects, of the total number of

Project	**Removed FP**	**Added TN**	**Added FP**
MFDemo	4	2	0
Sock Shop	5	3	2
total	9	5	2

Table 7.13: Summary of false positives and true negatives after running the Extender

instances found before running the extension (21), 9 of these were no longer found after executing the μFreshener++ extension (43% of instances removed), and all of them proved to be false positives. Running the extension also enabled the detection of 7 new smell instances (37% of instances added). Of these 7 smells found, 5 turned out to be true positives (71%) while 2 others are false positives (29%) introduced by the extension. Analyzing the numbers, we can affirm that the μFreshener++ extension allowed us to (1) eliminate 9 false positives from detection and (2) detect 5 true negative instances. The only problem introduced concerns the two false-positive instances introduced, but as already explained it is caused by the detection strategy adopted by μFreshener.

7.3.5 Discussion

This section briefly discusses all the results obtained. Regarding the validation of the μFreshener++ extension, a total of 25 *issues* were introduced in the μTOSCA models. Considering MFDemo, all the *issues* were correctly resolved, producing as a result an Extended model perfectly loyal to the original. As far as the Sock Shop is concerned, out of a total of 11 *issues*, only 9 were correctly resolved by the Extender. In particular, it was not able to solve I24 and I25, related to the lack of a Service node in the μTOSCA model and of the relative MessageRouter that exposes it. This, however, was an expected result, and it was considered anyway because (1) it was introduced from a "real use case" (using μMiner) and (2) it can be a starting point for future developments. Despite these two unresolved *issues*, all *controls* performed by the worker were tested and confirmed to be functioning correctly. The results obtained for the *controls* are shown in Table 7.14, also indicating which execution tested that particular *control*, and how many times it was tested in total. Considering the impact of extension on μFreshener detection, executing it reduced the number of false positive instances by 43% (a total of 9), and

	CT01	CT02	CT03	CM01	DB01	IN01	IN02	IT01
MFDemo	1	-	-	1	2	1	1	1
Sock Shop	-	1	1	1	2	-	-	-
	IC01	IG01	MR01	MR02	SV01	SV02	SV03	
MFDemo	1	1	-	1	1	-	-	
Sock Shop	-	-	1	-	1	1	1	

Table 7.14: Summary of *controls* checked by different projects. The number indicates how may time the *control* is triggered

added the detection of 5 new smells that did not emerge from the Injected model. The extension also added the detection of 2 new false positives, due however more to the strategy applied by μFreshener than to the extension activity.

We can therefore assert that the extension succeeded in adding benefits to the analysis carried out, both concerning the representation of the application and the detection of smells.

7.4 Results on Refactoring

The analysis presented in this section aim to verify that μFreshener++ can perform the (semi) automatic refactoring of architectural smells for microservices, as described before. The strategy adopted consist in executing μFreshener on the *Extended Model* to get the list of smells present in the system, and then run on this project the μFreshener++ tool for performing refactoring. Each Kubernetes resource generated by the tool is then analyzed, for understanding if the changes applied can solve the smells detected. As before, this analysis was run on both MFDemo (Section 7.4.2) and Sock Shop (Section 7.4.3) projects.

7.4.1 Methodology

This section introduces the methodology applied for conducting this analysis, to facilitate the reading of the next sections. As for the extension, the methodology is presented using a numbered list, and the same numbers are then used in subsequent sections as a reference to the stage described. The methodology is divided into the following stages:

1. The analysis starts from the *Extended Model*, were is performed the smell detection using μFreshener tool to obtain the list of smell instances present in the system. This list is presented in the so called *Smell Table*.
2. The μFreshener++ tool is then executed on the *Extended Model*, for solving the smells detected in step (1). This execution produces the (final) *Refactored Model* for the system. In this model, each component added or modified by the model is colored yellow, and the outline style indicates if this node was added (normal outline) or modified (dashed outline) by μFreshener++.

3. The execution of μFreshener++ refactoring produces also the report, which is the log of the operations performed by the tool for refactoring the smells. This report is presented in a dedicated table named *Report Table*, where each row corresponds to a refactoring applied.

4. The last step is the analysis of the refactoring techniques applied. For each refactoring, the Kubernetes resource generated or modified by it are analyzed, in order to understand if the smell was corrected by μFreshener++. In the presentation of the analysis, not all refactorings are shown in detail, since the result of applying the same technique always produces a similar result. For this reason, only one case of each type of refactoring has been analyzed in detail, while all others are only briefly discussed.

The methodology presented was applied for carrying out the analysis regarding the μFreshener++ refactoring feature on the MFDemo (Section 7.4.2) and Sock Shop (Section 7.4.3) projects. Each section presenting the results is divided into the 4 points presented above, in order to have a reference to the steps applied.

7.4.2 Results on MFDemo

This section presents results obtained from running the μFreshener++ refactoring on the MFDemo project.

(1) The analysis started from the execution of μFreshener on the *Extended Model* (Figure 7.6), for generating the list of smells present in the system. All smells found in the MFDemo project are enumerated in Table 7.15. In total, 7 instances were de-

ID	**Smell**	**Node**	**Caused by**
EB01	EB	H_{svc}	$B_{svc} \longrightarrow H_{svc}$
EB02	EB	M_{svc}	$B_{svc} \longrightarrow M_{svc}$
MS01	MS	FG_{cmp}	$F_{svc} \longrightarrow FG_{cmp}$ $G_{svc} \longrightarrow FG_{svc}$
NG01	NG	E_{svc}	-
WS01	WS	C_{svc}	$C_{svc} \longrightarrow D_{mr}$
WS02	WS	E_{svc}	$E_{svc} \longrightarrow D_{mr}$
WS03	WS	B_{svc}	$B_{svc} \longrightarrow A_{mr}$ $B_{svc} \longrightarrow H_{mr}$ $B_{svc} \longrightarrow M_{mr}$

Table 7.15: MFDemo Smell table

Legend: **EB** Endpoint-based Service Interaction smell **MS** Multiple Services in One Pod smell **NG** No Api Gateway smell **SP** Shared Persistence smell **WS** Wobbly Interaction Service smell

tected, divided into 2 instances of *Endpoint-based Service Interactions*, 1 instance for both *Multiple Services in One Pod* and *No API Gateway*, and 3 *Wobbly Service Interaction*.

In addition to the node on which the smell was found, the interactions that led to its detection are also indicated, to facilitate the understanding of the refactorings applied.

(2) The execution of the μFreshener++ for refactoring the system generated the *Refactored Model* (Figure 7.9), which is helpful in visualizing the changes made to the system by μFreshener++.

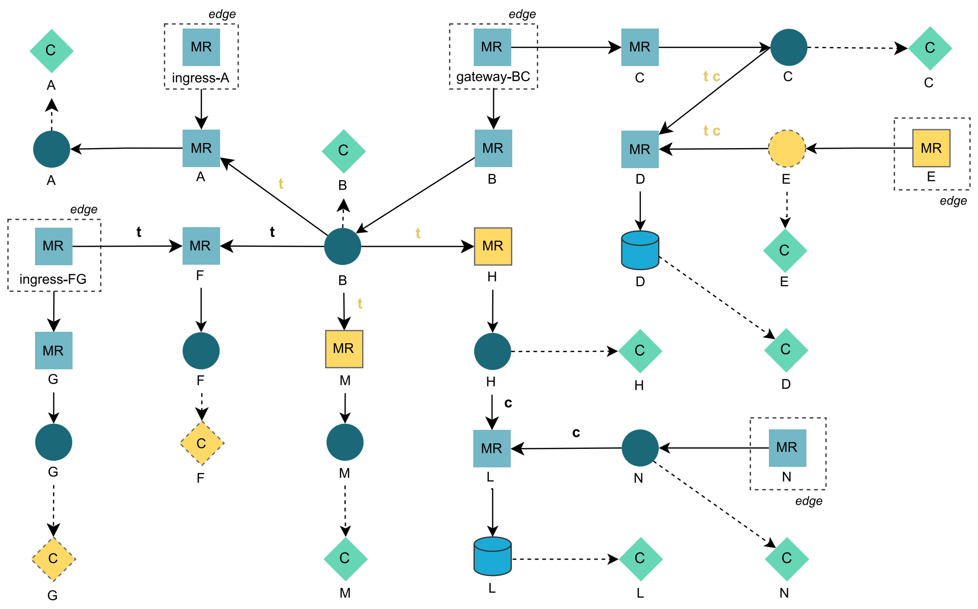


Figure 7.9: MFDemo Refactored model

By analyzing the graph, it is possible to visualize the changes introduced to the system by μFreshener++. For the resolution *Endpoint-based Service Interactions* (EB01 and EB02), μFreshener++ introduced two new message routers H and M exposing the related microservices. The resolution of *Multiple Services in One Pod* (MS01), on the other hand, required splitting the Compute FG into two different nodes (F and G), which deploy only one microservice. The resolution of *No API Gateway* (NG01), on the other hand, required both the introduction of a new entry-point for the system (MessageRouter E) and the modification of the microservice E to make it no longer reachable from the outside. For the resolution of *Wobbly Service Interaction* finally (WS01-WS03), several timeouts and circuit breakers were introduced in communications.

(3) The execution of μFreshener++ refactoring produced the report (Table 7.16) which illustrates all the refactoring techniques that were applied for solving the smells. Refactoring were not always completely applied da μFreshener++ because some requires additional developer intervention. These cases are indicated in the *Smell Table* through an asterisk. In total, μFreshener++ applied 7 different refactorings to the system, one for each smell identified, with only RF03 that is completeply applied.

(4) Finally, to verify the correct application of refactoring techniques, it was necessary to analyze the changes applied by μFreshener++ to the Kubernetes deployment. For each refactoring technique applied by the tool, an analysis in-depth of the changes applied is presented in the following list.

ID	Applied Refactoring	Kubernetes resources introduced
RF01*	Add Message Router (EB01)	H ClusterIP Service exposing Pod H
RF02*	Add Message Router (EB02)	M ClusterIP Service exposing Pod M
RF03	Split Services (MS01)	F-FG and G-FG Deployments
RF04*	Add API Gateway (NG01)	NodePort Service exposing E
RF05*	Use Timeout (WS01)	VirtualService defining timeout for D_{svc}
RF06*	Add Circuit Breaker (WS02)	DestinationRule defining circuit breaker for D_{svc}
RF07*	Use Timeout (WS03)	VirtualService defining timeout for A_{svc}, H_{svc}, M_{svc}

Table 7.16: Refactoring applied for solving smells detected in MFDemo

- *Add Message Router* (RF01, RF02) introduced two new ClusterIP Kubernetes Services into the system (MessageRouter H and M in the μTOSCA model). Only the application of RF01 is presented here, but the same applies for RF02. Considering the microservices B and H ($B_{svc} \longrightarrow H_{svc}$), they were able to communicate directly before the application of refactoring, with B having (ideally) the hard-coded location (IP) of H in its code-base. This way, in case H was scaled, the new instances would never be used by B. This refactoring introduced a new Kubernetes Service H (Listing 11) into the system, which allowed the direct communication between microservices to be broken, thus removing the problems associated with *Horizontal Scalability.*

```
apiVersion: v1
kind: Pod
metadata:
  name: H
  labels:
    service: H
    H.default-svc-mf: 4f1379ac..
spec:
  containers:
  - name: H
    image: generic-img
    ports:
    - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: H-clusterip-svc-mf
  namespace: default
spec:
  ports:
  - name: H.H.default-port-80-mf
    port: 80
  selector:
    H.default-svc-mf: 4f1379ac..
```

Listing 11: Kubernetes Service generated by Add Message Router refactoring (RF01) for exposing Pod H

The listing shows the example of the Kubernetes Service generated for exposing the Pod H. The Service shares a label with the Pod and defines port 80 which points to the 80 exposed by the container, and so it exposes the Pod H. After deploying this Service, to complete this refactoring the developer must change in the container B all requests that send messages directly to H, by changing the H's IP address to the hostname of the generated Kubernetes Service (i.e., *H-clusterip-svc-mf.default.svc.cluster.local:80*).

- The application of *Split Services* (RF03) caused the Pod *FG* to be split into two smaller pods *F* and *G*, with each defining one of the two containers *F* and *G*. This resulted in the *F* and *G* containers being defined by two different resources, making them effectively deployable independently of each other. Before the refactoring was applied, it was not possible to increase the running instances of only one or the other microservice, because in Kubernetes is scaled the whole Pod (which previously defined both containers inside it). After applying RF03, instead, it is possible to scale the microservice *G* without changing the number of instances of *F*, since two different pods define them. Listing 12 shows the application of refactoring, which separated the *FG* Pod into two smaller *F* and *G* pods.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: F-FG
  labels:
    service: FG
spec:
  replicas: 1
  selector:
    matchLabels:
      service: FG
  template:
    metadata:
      labels:
        service: FG
    spec:
      containers:
      - name: F
        image: generic-image
        ports:
        - containerPort: 80
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: G-FG
  labels:
    service: FG
spec:
  replicas: 1
  selector:
    matchLabels:
      service: FG
  template:
    metadata:
      labels:
        service: FG
    spec:
      containers:
      - name: G
        image: generic-image
        ports:
        - containerPort: 81
```

Listing 12: Kubernetes deployments generated by Split Services refactoring (RF03)

The old *FG* resource that defined both containers was deleted, and in its place the two pods represented were introduced. After deploying them, the refactoring is finished since, as explained in the refactoring description, they keep the same structure (i.e., the label has the same value) as the *FG* Pod and therefore from the point of view of the other components nothing changes.

- The application of *Add API Gateway* (RF04) introduced a new NodePort Kubernetes Service that exposes the Pod *E* outside the cluster. At the same time, *E* is made unreachable from the outside. With these modifications, the problem related to *Horizontal Scalability* is solved. Before refactoring, in fact, the external client directly contacted a running instance of the *E* microservice. If *E* was then scaled, the new instances were not reachable by the external clients, as they continued to use the contacted instance. By passing through a Kubernetes Service that exposes all instances, requests from clients are handled by the Service, which then routes the message to an instance of E. Listing 13 shows the application of refactoring, which modifies the definition of the E Pod and generates a new Kubernetes Service to expose it to the outside.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: E
  labels:
    service: E
spec:
  replicas: 1
  selector:
    matchLabels:
      service: E
  template:
    metadata:
      labels:
        service: E
        E.default-svc-mf: e32f5..
    spec:
      hostNetwork: false
      containers:
      - name: E
        image: generic-image
        ports:
        - containerPort: 80
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: E-nodeport-svc-mf
  namespace: default
spec:
  type: NodePort
  ports:
  - name: E.E.default-port-80-mf
    port: 80
    node_port: 30000
  selector:
    E.default-svc-mf: e32f5..
```

Listing 13: Kubernetes resources modified and created by Add API Gateway refactoring (RF04) (RF03)

By applying these changes, the Pod E is no longer directly exposed to the outside, as its property *hostNetwork* is modified by assigning it the value *false*. The Service *E-nodeport-svc-mf* shares a label with the Pod, defines port 80 which points to the 80 exposed by the container and exposes this port to the outside on the host port 30000. In this way, when the Service is deployed on the cluster, is possible to contact Pod E through host's port 30000, resolving so the smell. To complete the refactoring, the developer must perform port-forwarding to send requests directed to the old exposed port (80) to the new one (30000).

- The application of *Use Timeout* (RF05, RF07, RF08) introduced 3 new VirtualService resources into the system, which define the timeout for 3 different message routers. For this analysis, the smell identified on MessageRouter D (WS01) is considered, but the solution is similar for all 3 smells found. The VirtualService defined a new timeout in the communication, thus solving the problem related to error propagation when *D* fails. By having a defined timeout, if at some point of the execution *D* becomes unreachable, the microservices *C* and *E* do not remain in an infinite wait but receives an error message when the timeout expires, preventing error propagation. Listing 14 shows the generated VirtualService.

By performing the deployment of this resource, a timeout of 2 seconds is defined for all direct interactions to the *D.default* service. When *D* does not respond to a request for the specified amount of time, the microservice invoking *D* receives an error message from Istio. To complete this refactoring, the developer needs to make the change to all services that interact with the Kubernetes Service *D.default*, to handle any errors due to the expiration of the timeout.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: D-vs-timeout-mf
  namespace: default
spec:
  http:
  - route:
    - destination:
        host: D.default
    timeout: 2s
  hosts: [D.default]
```

Listing 14: VirtualService generated by Use Timeout refactoring (RF05)

- Refactoring *Add Circuit Breaker* (RF06) introduced a new DestinationRule into the deployment that defines a circuit breaker for all interactions directed to the *D* service. The error is resolved as with refactoring *Use Timeout*, except that in this case the error message is not returned to the caller when the timeout expires, but when *D* is too overloaded or unreachable. The Listing 15 shows the newly generated DestinationRule. By performing the deployment of this resource, a circuit breaker is defined for all direct interactions directed to the *D.default* service. When *D* is overloaded or unreachable, the microservice invoking *D* receives an error message. To complete this refactoring, the developer must perform the modification of all services that interact with the Kubernetes Service *D.default*, to handle any errors received from the circuit breaker.

The analysis of Kubernetes resources confirmed that μFreshener++ generated resources that can solve the smells detected. However, by analyzing both the *Refactored Model* (Figure 7.9) and the refactoring report (Table 7.16), we can observe that both a timeout and a circuit breaker were generated for the *D* service. This does not cause problems with the operation of the system, but it is definitely an unexpected behavior on the part of μFreshener++ that will need to be corrected in future versions of the project.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: D.default-circuitbreaker-mf
spec:
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 5
      http:
        http1MaxPendingRequests: 3
        maxRequestsPerConnection: 3
    outlierDetection:
      consecutive5xxErrors: 2
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 20
  host: D.default
```

Listing 15: DestinationRule generated by Add Circuit Breaker refactoring (RF06)

In total, μFreshener++ applied seven different refactorings to solve as many smells. This section has presented in detail five out of seven (RF01, RF03, RF04, RF05, RF06), as the other two not described (RF02, RF07) are exactly the same as shown but including different names and resources. In general, we can affirm that the tool performed the resolution of all 7 smells, but with the error described above related to the application of *Use Timeout* and *Add Circuit Breaker* techniques.

7.4.3 Results on Sock Shop

This section presents results obtained from running the μFreshener++ refactoring on Sock Shop project.

(1) The analysis started from the execution of μFreshener on the Extended model (Figure 7.8), for generating the list of smells present in the system. All smells found are enumerated in Table 7.17. The execution found 12 different smell instances, including 7 instances of *Wobbly Service Interaction*, 2 for *Endpoint-based Service Interactions* and *Multiple Services in One Pod*, and 1 instance of *No API Gateway*. Some of these smells were entered manually within the system (Section 7.1.2), and these numbers confirm exactly what was exhibited earlier (Table 7.2).

(2) The execution of μFreshener++ for refactoring the system generated the *Refactored Model* (Figure 7.10), which is helpful in visualizing the changes made to the system by μFreshener++. By analyzing the graph, it is possible to visualize the changes introduced to the system by μFreshener++. For the resolution *Endpoint-based Service Interactions* (EB03, EB04), μFreshener++ introduced two new message routers *payment* and *shipping*, exposing the related microservices. The resolution of *Multiple Services in One Pod* (MS02, MS03), on the other hand, required splitting the computes *orders* and *catalogue-carts-db*, for having four Compute that deploys four microservices. The resolution of *No*

ID	Smell	Node	Caused by
EB03	EB	payment$_{svc}$	orders$_{svc}$ $\longrightarrow$ payment$_{svc}$
EB04	EB	shipping$_{svc}$	orders$_{svc}$ $\longrightarrow$ shipping $_{svc}$
MS02	MS	orders$_{cmp}$	orders$_{svc}$ $\longrightarrow$ orders$_{cmp}$ test$_{svc}$ $\longrightarrow$ orders$_{cmp}$
MS03	MS	catalogue-carts-db$_{cmp}$	catalogue-db$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{cmp}$ carts-db$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{cmp}$
NG02	NG	front-end$_{svc}$	-
WS04	WS	carts$_{svc}$	catalouge-carts-db$_{svc}$ $\longrightarrow$ carts$_{mr}$
WS05	WS	catalogue$_{svc}$	catalogue$_{svc}$ $\longrightarrow$ catalogue-carts-db$_{mr}$
WS06	WS	front-end$_{svc}$	front-end$_{svc}$ $\longrightarrow$ catalogue$_{mr}$ front-end$_{svc}$ $\longrightarrow$ carts$_{mr}$ front-end$_{svc}$ $\longrightarrow$ orders$_{mr}$ front-end$_{svc}$ $\longrightarrow$ user$_{mr}$
WS07	WS	orders$_{svc}$	orders$_{svc}$ $\longrightarrow$ carts$_{mr}$ orders$_{svc}$ $\longrightarrow$ orders-db$_{mr}$ orders$_{svc}$ $\longrightarrow$ payment$_{mr}$ orders$_{svc}$ $\longrightarrow$ user$_{mr}$ orders$_{svc}$ $\longrightarrow$ shipping$_{mr}$
WS08	WS	queue-master$_{svc}$	queue-master$_{svc}$ $\longrightarrow$ rabbitmq$_{mr}$
WS09	WS	shipping$_{svc}$	shipping$_{svc}$ $\longrightarrow$ rabbitmq$_{mr}$
WS10	WS	user$_{svc}$	user$_{svc}$ $\longrightarrow$ user$_{mr}$

Table 7.17: Sock Shop Smell table

Legend: **EB** Endpoint-based Service Interaction smell **MS** Multiple Services in One Pod smell **NG** No Api Gateway smell **SP** Shared Persistence smell **WS** Wobbly Interaction Service smell

API Gateway (NG02), on the other hand, required both the introduction of a new entrypoint for the system (MessageRouter *front-end*) and the modification of the microservice *front-end* to make it no longer reachable from the outside. For the resolution of *Wobbly Service Interaction* finally (WS04-WS010), several timeouts and circuit breakers were introduced in communications.

(3) The execution of μFreshener++ refactoring produced the report, shown in Table 7.18, which illustrates all the refactoring techniques that were applied for solving the smells.

In total, μFreshener++ was able to apply 12 different refactorings to the system, one for each smell identified. Of these refactorings, only RF10 and RF11 can be considered complete, while the other require a little developer intervention on the system's microservices.

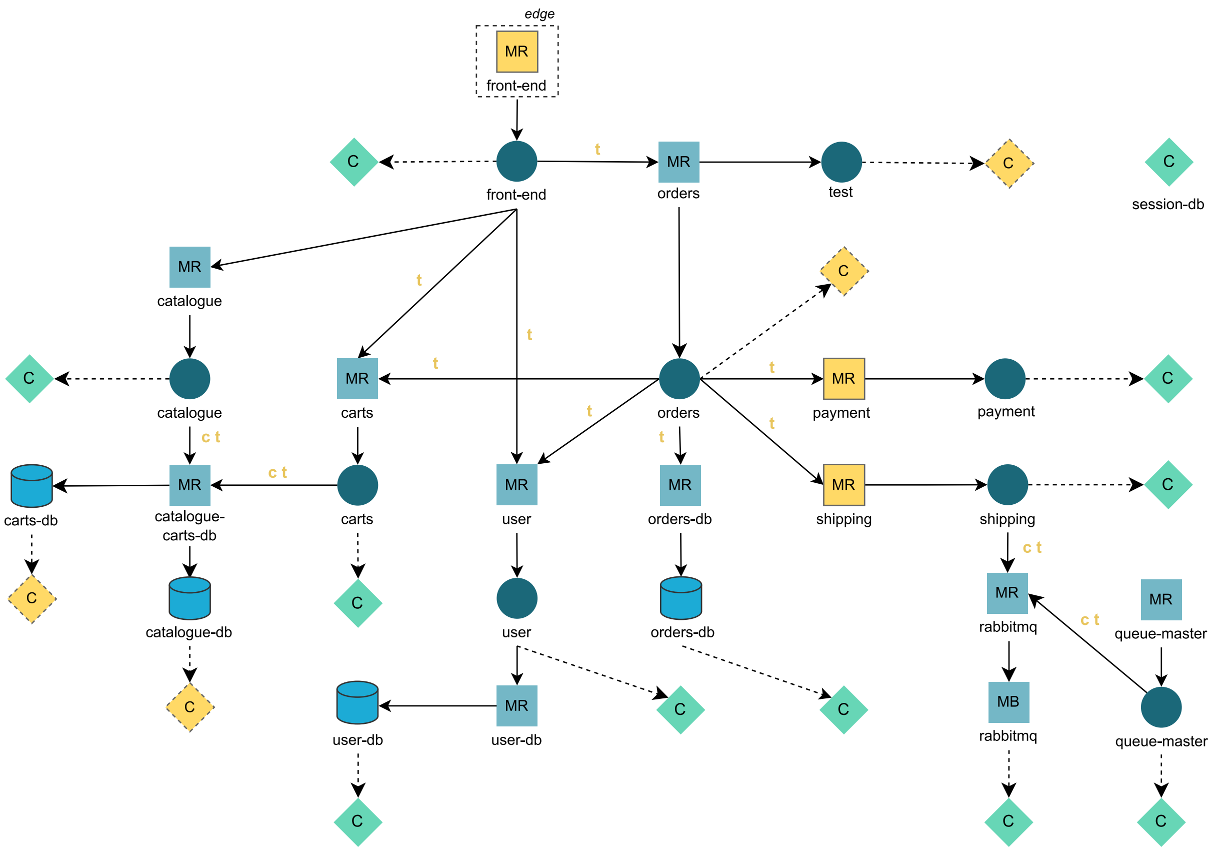


Figure 7.10: Sock Shop Refactored model

ID	Applied Refactoring	Kubernetes resources introduced
RF08*	Add Message Router (EB03)	*payment* Service exposing *payment* Pod
RF09*	Add Message Router (EB04)	*shipping* Service exposing *shipping* Pod
RF10	Split Services (MS02)	*test-orders* Deployment *orders-orders* Deployment
RF11	Split Services (MS03)	*catalogue-db-catalogue-carts-db* Deployment *carts-db-catalogue-carts-db* Deployment
RF12*	Add API Gateway (NG02)	*front-end* NodePort Service exposing *front-end* Pod
RF13*	Use Timeout (WS06)	VirtualService defining timeout for *catalogue-carts-db*
RF14*	Add Circuit Breaker (WS07)	DestinationRule defining CB for *catalogue-carts-db*
RF15*	Use Timeout (WS08)	VirtualService defining timeout for *catalogue* Service VirtualService defining timeout for *user* Service VirtualService defining timeout for *orders* Service
RF16*	Use Timeout (WS09)	VirtualService defining timeout for *orders* Service VirtualService defining timeout for *carts* Service VirtualService defining timeout for *payment* Service VirtualService defining timeout for *shipping* Service
RF17*	Use Timeout (WS10)	VirtualService defining timeout for *rabbitmq* Service
RF18*	Add Circuit Breaker (WS11)	VirtualService defining timeout for *rabbitmq* Service
RF19*	Use Timeout (WS12)	VirtualService defining timeout for *user-db* Service

Table 7.18: Refactoring applied for solving smells detected in Sock Shop

(4) Having already shown in detail the application of refactorings and how they solve the problem, only a general overview regarding the refactorings applied on this project is shown here. By analyzing the Refactored model, the generated Kubernetes resources, and the refactoring report, we can state that:

- The application of *Add Message Router* (RF08, RF09) added two new Kubernetes services of ClusterIP type into the system, represented by the message routers *payment* and *shipping* in the μTOSCA model. The introduction of these Kubernetes services solves the scalability problem as seen above, and thus the two smells are solved.

- The application of *Split Services* (RF10, RF11) causes the splits of pods *orders* and *catalogue-carts-db*. Specifically, the former is split into *orders-orders* and *test-orders*, and the latter into *catalogue-db-catalogue-carts-db* and *carts-db-catalogue-carts-db*. In both cases, the independence problem is solved, and therefore the smells can be considered solved.

- The application of *Add API Gateway* (RF12) causes the introduction of a new Kubernetes Service of type NodePort that exposes the Pod *front-end* outside the cluster, which at the same time is made unreachable from the outside. Again, the addition of this Service solves the related problem by solving the smell.

- The application of *Use Timeout* (RF13, RF15, RF16, RF17, RF19) introduced 5 new VirtualService resources into the system, which define timeout for 9 different Kubernetes services. This refactoring succeeded in solving the problem related to the isolation of failures.

- The application of *Add Circuit Breaker* (RF14, RF18) introduced two new DestinationRule resources into the deployment, that define the circuit breaker for all interactions to the Kubernetes services *catalogue-carts-db* and *rabbitmq*. Again, this refactoring is correctly applied and it solves the failure isolation problem.

The analysis of Kubernetes resources confirmed that μFreshener++ succeed in generating Kubernetes resources that can solve the smells. However, even in this case is present the problem of timeout and circuit breaker defined for the same property, on MessageRouter nodes *catalogue-carts-db* and *rabbitmq*.

In total, μFreshener++ applied 12 different refactorings to solve as many smells. Of the 12 techniques applied, however, only 10 can be considered as correctly applied, because the application of timeout and circuit breaker for the same Kubernetes Service is an unexpected behavior of μFreshener++.

7.4.4 Discussion

This section briefly discuss all the results obtained from running the μFreshener++ refactoring functionality.

Considering MFDemo, all smells detected by μFreshener on the μTOSCA model were solved, producing as output a new deployment which removes all the smells. To resolve the smells in the system, seven different refactorings were applied, but of these only six can be considered correctly applied. In one case, two instances of the *Wobbly Service Interaction* smell (here called WS_1 and WS_2) related to the same MessageRouter were found in the project. μFreshener++ applied *Use Timeout* for the resolution of WS_1, but it was defined for all incoming interactions to the MessageRouter, including the one related to the WS_2 smell. When μFreshener++ had to resolve WS_2 it already found a timeout defined in the interaction (so the smell was already resolved) and then applied *Add Switch* to resolve it. This caused both a timeout and a circuit breaker to be defined for the same Kubernetes Service. This behavior does not cause any problems to the system, and so the smell can be considered solved, but still it is an unexpected behavior that needs to be fixed in future releases of μFreshener++.

As for Sock Shop, all 12 instances found were resolved correctly by μFreshener++. However, even for Sock Shop the problem discussed above occurred, with two cases of timeouts and circuit breakers "overlapping" for the same interactions.

Project	**Detected smells**	**Solved smells**	**Successful refactoring**
MFDemo	7	7	6
Sock Shop	12	12	10
total	19	19	16

Table 7.19: Summary of smells solved and successful techniques applied

Analyzing the table above presenting final refactoring results, we can verify that all 19 smells were correctly resolved, thus solving 100% of the considered smells. Considering the refactorings, 3 techniques were not applied correctly, and therefore 84% of them can be considered successfully completed. The numbers for successful refactorings also consider the changes that the developer must apply to the microservices for their completion.

We can conclude by stating that μFreshener++ was able to solve all the identified smells by fixing all the smells and correctly applying 84% of the techniques.

8 Conclusions and Future Development

The main contribution of this thesis was the extension of the functionality of the μTOSCA toolchain. This extension involved the detection of a new microservice smell, improvements to existing smell detection capabilities, and the introduction of a feature for refactoring the identified smells.

As for the detection of a new of a new architectural smell for microservice, namely *Multiple Services in One Pod* [19], we extended the existing μFreshener tool by implementing into it a new refactoring strategy. To implement such strategy, we needed to extend the μTOSCA language to represent the deployment of microservices in the model.

As for the improvement of the μTOSCA toolchain smell detection, we introduced the extension functionality of μFreshener++ tool. The extension consists of correcting and enriching the μTOSCA model describing the application, by using information obtained from the Kubernetes deployment files of the analyzed system. Running the μFreshener++ extension improves the μFreshener smell detection results, by removing all inaccuracies from the μTOSCA model, making it as similar as possible to the real application. In this way, the toolchain smell detection is performed using a μTOSCA model that more accurately represents the system, thus removing some false positives and including some true negatives from detection.

As for the support of automated refactoring for four microservices smells, we introduced the refactoring functionality of μFreshener++. The refactoring consists of applying five refactoring techniques [19] to solve four architectural smells, namely *Endpoint-based Service Interactions*, *Multiple Services in One Pod*, *No API Gateway*, and *Wobbly Service Interaction*. Each smell is solved by μFreshener++ directly modifying the system's Kubernetes deployment by creating, modifying, or deleting new Kubernetes resources and, in most cases, an additional (limited) change to one or more microservices is required to complete the technique.

As for the validation, we executed μFreshener and μFreshener++ tools on two test projects. Regarding detection, the validation identified a total of 5 smells, of which 3 problems were actually present and 2 of them were false positives (purposely detected to demonstrate the known issue of the detection strategy). Considering the results obtained for the extension, μFreshener++ was able to solve all the problems introduced in the model, thus verifying that the developed functionality works properly. The only exception concerned two nodes that μFreshener++ did not include in the system, since it does not handle this case. So, 23 out of 25 issues were solved. This extension also had a positive effect on smell detection, reducing the number of false positives by 9 (from 19) and enabling the detection of 5 true negative and 2 false positive new cases. The refactoring results showed that μFreshener++ successfully remove all 19 smells identified in the two systems. However, of the 19 techniques used, only 16 can be considered successfully completed, while the other are considered failures.

This thesis presented the extension of the μTOSCA toolchain, but more can be done, both to refine what has been presented and to introduce new features.

During the description of this thesis, we presented several problems related to detection, extension, and refactoring. The detection of *Multiple Services in One Pod* performed by μFreshener may identify as smell cases where multi-container Kubernetes patterns are applied on pods. Therefore, the detection strategy could be modified to prevent these false positives cases from being identified, by recognizing recurrent pattern in the μTOSCA model that can signal the application of the Kubernetes pattern. To address the problem of a Service node not being added to the μTOSCA model, which emerged during validation, a possible solution is to develop a new extension worker that verifies the representation of each Pod in the model. During the refactoring process, it was found that μFreshener++ applied multiple techniques to the same node to address different smells, resulting in unnecessary Kubernetes resource defined. To prevent this, appropriate controls could be implemented during the application of refactoring techniques, for avoiding the application of a refactoring if the smell has been already (indirectly) solved by other techniques applied. All these improvements could be implemented in future versions of μFreshener and μFreshener++ tools.

In the context of this thesis, a validation that verifies the impact that refactorings have on the running system could be carried out. This could be done by analyzing the differences in the behavior of the system before and after the application of the refactoring, to show that the refactoring was indeed able to solve the problem in the system. This would show how the tool works in a "real world" environment, where the developer uses μFreshener++ for solving smells and then re-executes the refactored application.

The entire toolchain supports only Kubernetes deployments, but all tools are designed with maximum flexibility to incorporate additional implementation technologies. One idea for extending the capabilities could be to support other deployment technologies besides Kubernetes. In addition, tools such as Kompose [10], which transforms a Docker-compose deployment into a Kubernetes deployment, could also be exploited. In this way, support for Docker-compose could be provided by simply converting the deployment to Kubernetes and then using the features already developed.

The μTOSCA toolchain could be extended to include the detection and refactoring of other types of smells, such as Security Smells [22], which occur in microservice-based applications and indicate potential security vulnerabilities in the system. This extension could be accomplished by adding a new dedicated tool to the toolchain that implements detection and refactoring of these smells or by modifying the existing ones, namely μFreshener and μFreshener++.

References

[1] *Agile Alliance: Test Driven Development.* https://www.agilealliance.org/glossary/tdd/. last access 10-03-2023.

[2] Paolo Bacchiega, Ilaria Pigazzini, and Francesca Arcelli Fontana. "Microservices smell detection through dynamic analysis". In: *Proceedings of Euromicro Conference, SEAA-TD track* (Aug. 2022).

[3] Rodrigo Weissmann Borges. "Algorithm for Detecting Antipatterns in Microservices Projetcs". In: 2019.

[4] F. A. Fontana et al. "Arcan: A Tool for Architectural Smells Detection". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* 2017, pp. 282–285.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[6] Perera Indika Gamage Isuru Udara Piyadigama. "Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach". In: *2021 Moratuwa Engineering Research Conference (MERCon).* 2021, pp. 699–704. DOI: 10.1109/MERCon52712.2021.9525743.

[7] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: https://books.google.it/books?id=6oHuKQe3TjQC.

[8] Joshua Garcia et al. "Identifying Architectural Bad Smells". In: *2009 13th European Conference on Software Maintenance and Reengineering.* 2009, pp. 255–258. DOI: 10.1109/CSMR.2009.59.

[9] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN: 9780133065107. URL: https://books.google.it/books?id=qqB7nrrna%5C_sC.

[10] *https://kompose.io/.* https://github.com/Daviderendina/MFDemo. last access 23-03-2023.

[11] Tim Hübener et al. "Automatic Anti-Pattern Detection in Microservice Architectures Based on Distributed Tracing". In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 2022, pp. 75–76. DOI: 10.1145/3510457.3513066.

[12] *Istio Docs - circuit breaking.* https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/. last access 16-01-2023.

[13] *Istio Service Mesh.* https://istio.io/latest/about/service-mesh/. last access 01-03-2023.

[14] *Kubernetes Overview.* https://kubernetes.io/docs/concepts/overview/. last access 01-03-2023.

[15] *Kubernetes Patterns.* O'Reilly Media, Inc., 2019. ISBN: 9781492050230.

[16] *Kubernetes Pod docs.* https://kubernetes.io/docs/concepts. last access 18-01-2023.

[17] Abdullah Maruf et al. "Using Microservice Telemetry Data for System Dynamic Analysis". In: Aug. 2022, pp. 29–38. DOI: 10.1109/SOSE55356.2022.00010.

[18] *MFDemo project.* https://github.com/Daviderendina/MFDemo.

[19] Davide Neri et al. "Design principles, architectural smells and refactorings for microservices: a multivocal review". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Sept. 2019), pp. 3–15. DOI: 10.1007/s00450-019-00407-8. URL: https://doi.org/10.1007%5C%2Fs00450-019-00407-8.

[20] *Oasis TOSCA docs v2.0.* https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html. last access 10-01-2023.

[21] Ilaria Pigazzini et al. "Towards Microservice Smells Detection". In: May 2020. DOI: 10.1145/3387906.3388625.

[22] Francisco Ponce et al. "Smells and refactorings for microservices security: A multivocal literature review". In: *Journal of Systems and Software* 192 (2022), p. 111393.

[23] Dmytro Rud, Andreas Schmietendorf, and Reiner R. Dumke. "Product Metrics for Service-Oriented Infrastructures". In: 2006.

[24] *Sock Shop project.* https://microservices-demo.github.io/.

[25] Jacopo Soldani, Javad Khalili, and Antonio Brogi. "Offline Mining of Microservice-based Architectures". In: *International Conference on Cloud Computing and Services Science.* 2022.

[26] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232.

[27] Jacopo Soldani et al. "The TOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures". In: *Software: Practice and Experience* 51 (Apr. 2021). DOI: 10.1002/spe.2974.

[28] Davide Taibi and Valentina Lenarduzzi. "On the Definition of Microservice Bad Smells". In: *IEEE Software* vol 35 (May 2018). DOI: 10.1109/MS.2018.2141031.

[29] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Microservices Anti Patterns: A Taxonomy". In: June 2019. DOI: 10.1007/978-3-030-31646-4_5.

[30] *VmWare: What Are Containers?* https://www.vmware.com/topics/glossary/content/containers.html. last access 01-03-2023.

[31] Andrew Walker, Dipta Das, and Tom Černý. "Automated Microservice Code-Smell Detection". In: Jan. 2021, pp. 211–221. ISBN: 978-981-33-6384-7. DOI: 10.1007/978-981-33-6385-4_20.

[32] Muhammad Waseem et al. "On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study". In: *Evaluation and Assessment in Software Engineering*. EASE 2021. Trondheim, Norway: Association for Computing Machinery, 2021, pp. 201–210. ISBN: 9781450390538. DOI: 10.1145/3463274.3463337. URL: https://doi.org/10.1145/3463274.3463337.

[33] Muhammad Waseem et al. *Understanding the Issues, Their Causes and Solutions in Microservices Systems: An Empirical Study*. 2023. DOI: 10.48550/ARXIV.2302.01894. URL: https://arxiv.org/abs/2302.01894.