CLEI electronic journal, Volume 27, Number 2, Paper 6, July 2024

Beyond Security: Understanding the Multiple Impacts of Security Smells for Microservices

Francisco Ponce
Universidad Técnica Federico Santa María & ITiSB - Universidad Andrés Bello
Valparaiso, Chile
francisco.ponceme@usm.cl

and

Jacopo Soldani
University of Pisa, Pisa, Italy
jacopo.soldani@unipi.it

and

Carla Taramasco
ITiSB - Universidad Andrés Bello, Viña del Mar, Chile
carla.taramasco@unab.cl

and

Antonio Brogi
University of Pisa, Pisa, Italy
antonio.brogi@unipi.it

and

Hernán Astudillo
ITiSB - Universidad Andrés Bello, Viña del Mar, Chile
hernan.astudillo@unab.cl

Abstract

Microservice-based applications enable building *cloud-native* applications, namely applications that can fully exploit the benefits of cloud computing. Along with its benefits, microservices come with new security challenges, including security smells, viz., symptoms of bad (though often unintentional) design decisions that might affect application security. This study aims to explore the impacts of microservice security smells –and of the refactorings known to mitigate their effects– beyond security. In particular, we systematically elicit possible impacts of smells and refactorings on applications' maintainability, performance efficiency, and adherence to microservices' key design principles. We then validate the elicited impacts through an online survey targeting experienced practitioners and researchers. Our main contributions include 35 validated impacts and a discussion of the survey results geared towards analyzing the (mis)alignment between practitioners and researchers. Finally, we also provide a holistic view of these impacts, through Softgoal Interdependency Graphs (SIGs).

Keywords: microservices, security smells, refactoring, maintainability, performance efficiency.

1 Introduction

Microservice-based applications (MSAs) enable building *cloud-native* applications [1], namely applications that can fully exploit the capabilities of cloud computing, exhibiting distributed, dynamic, and fault-resilient behavior [2]. MSAs are essentially service-oriented applications adhering to an extended set of design principles [3], which include the independent deployability and horizontal scalability of microservices, failure isolation, and decentralization. With their gains, however, microservices also bring some pains, and securing microservice-based applications is certainly one of those [4].

MSAs introduce new security challenges, including the so-called *security smells*, namely the possible symptoms of bad design decisions (though often unintentional) that can negatively impact the overall security of an application. The effects of security smells can be mitigated by refactoring the application or the services therein while not changing the functionalities offered to clients. We elicited the most recognized security smells in our previous work [5] together with the refactorings known to mitigate their effects.

Keeping a security smell or applying a refactoring to mitigate its effects are design decisions that deserve careful analysis. Such design decisions indeed impact multiple different aspects of an application, including the adherence to microservices' key design principles [6] and other quality attributes besides security, such as maintainability and performance efficiency, which are crucial in microservice applications [4].

Consider the *centralized authorization* security smell, which occurs when a single component centrally authorizes all external requests sent to a microservice-based application [5]. When this security smell is present, the requests exchanged among the application's microservices are trusted without additional authorization controls, leaving them vulnerable to various attacks such as confused-deputy attacks, which can compromise the application's authenticity [7]. To mitigate the effects of *centralized authorization*, application architects may *use decentralized authorization*, which involves refactoring the application to implement fine-grained authorization controls at the microservices level [5]. This would not only improve the authenticity of the application but also have a ripple effect on other quality attributes and strengthen the adherence to key microservices design principles.

As knowing such impacts helps to make informed decisions about keeping a security smell or applying a refactoring [8], the research question we try to answer in this paper is the following:

> *Are the security smells and refactorings in [5] impacting on maintainability, performance efficiency, and adherence to microservices' key design principles?*

In this perspective, we start from the selected literature from [5] and we apply thematic analysis [9,10] to systematically elicit 42 possible impacts of security smells and refactorings on applications' maintainability, performance efficiency, and adherence to microservices' key design principles. We then validate the elicited impacts, by analyzing the results of an online survey targeting practitioners and researchers working with microservice applications. Our analysis also shows that the interviewed practitioners and researchers are mostly aligned in their agreement with elicited impacts. A few misalignments emerged on possible impacts on testability and performance efficiency, with practitioners being more cautious in agreeing with them compared to researchers.

Consequently, we address our main research question by unveiling 35 validated impacts of microservices' security smells and refactorings on maintainability, performance efficiency, and key design principles. For each impact, we provide a statement describing the impact type (positive or negative), the property it affects, and the rationale behind it, together with the agreement of the interviewed practitioners and researchers.

The rest of this paper is organized as follows. Section 2 provides the necessary background on microservice security smells and softgoal interdependency graphs. Sections 3 and 4 describe the design of our survey and discuss the collected responses, respectively. Section 5 discusses how to use the confirmed impacts, as well as the possible threats to the validity of our study. Sections 6 and 7 discuss related work and draw some concluding remarks, respectively.

This article extends our previous work [11] by providing (i) more details on the population participating in the survey, (ii) a holistic visualization of the confirmed impacts in the form of SIGs, and (iii) a discussion of how such a visualization enables the trade-off analysis we envisioned in [8].

2 Background

This section provides the necessary background on microservice security smells and refactorings and softgoal interdependency graphs.

2.1 Smells and Refactorings for Microservice Security

We hereafter recall the smells and refactorings for microservice security proposed in our previous work [5], focusing on those whose impact is analyzed in this paper.

2.1.1 Insufficient Access Control

This smell occurs on the microservices of an application that are not enforcing access control. This can possibly violate the confidentiality of the data and business functions of the microservices where access control is lacking, as attackers can trick a service and get data that they should not be able to get.

The possible effects of this smell can be mitigated by exploiting OAuth 2.0, which would enable microservices to control accesses. OAuth 2.0 indeed provides a token-based access control system that lets a resource owner grant a client access to a particular resource on their behalf. OAuth 2.0 is hence a natural candidate to enforce access control in a microservice-based application at each level, thereby including controlling the accesses to each microservice.

2.1.2 Publicly Accessible Microservices

A microservice of an application is publicly accessible when it can be directly accessed by external clients. This increases the application's attack surface and reduces its overall maintainability and usability. Also, if each publicly accessible microservice performs authentication, the full set of a users credentials is required each time, increasing the likelihood of confidentiality violations (e.g., with the exposure of long-term credentials).

The suggested refactoring is making microservices accessible *only* through a newly added API Gateway, which would act as an entry point for the application. This would enable centralizing authentication, overall reducing the attack surface of the application and simplifying the authentication itself. In addition, by using this approach development teams can also secure all microservices behind a firewall, allowing the API gateway to handle external requests and then communicate with the microservices behind the firewall.

2.1.3 Unnecessary Privileges to Microservices

This smell occurs when microservices are granted unnecessary access levels, permissions, or functionalities that they do not need to deliver their business functions. As a result, resources are unnecessarily exposed, increasing the attack surface, and the risk of confidentiality and integrity violations.

The suggested refactoring is to follow the least privilege principle, namely ensuring that microservices have access *only* to the least set of functionalities and data needed to suitably perform their business function. This would help to contain attacks' effects, e.g., if an attacker gets control of a microservice by exploiting its software vulnerabilities.

2.1.4 Non-Secured Service-to-Service Communications

This smell occurs whenever microservices in an application interact without establishing a secure communication channel, even if they are running in the same internal network. This could result in confidentiality, integrity, and authenticity issues, e.g., intruders could intercept the communication between two microservices and change the data in transit to their advantage.

Microservices should rather follow a zero-trust approach, by relying on Mutual TLS to secure service-to-service communications Mutual TLS indeed creates a secure communication channel that features data encryption and mutual authentication, hence preventing, e.g., man-in-the-middle attacks.

2.1.5 Unauthenticated Traffic

This smell occurs when a microservice receives unauthenticated requests from external clients or from other microservices of its same application, which may result in violating the application's authenticity. The microservices in an application should rather always authenticate and authorize incoming requests, to ensure that data arriving from external clients and exchanged among the application's microservices is trusted and has not been modified. If the traffic is not authenticated microservices are exposed to security attacks that may result in, e.g., tampering with data, denial of service, or elevation of privileges.

The suggested refactorings are using Mutual TLS, as well as OpenID Connect. The latter exploits ID tokens containing cryptographically signed user claims, which can be checked for integrity, and which can be used to perform access control at the microservice level.

2.1.6 *Multiple User Authentication*

This smell occurs when a microservice application provides multiple endpoints for user authentication, which can be exploited by an intruder to authenticate as an end-user. This increases the application's attack surface, and could also result in maintainability and usability issues.

The suggested refactoring is using a single sign-on, namely using a single entry point to handle user authentication and to enforce security for all the user requests entering a microservice application. This approach facilitates log storage and auditing tasks, by providing a centralized entry point that performs user authentication. The single sign-on can be realized by adding an API gateway and using OpenID connect, which are refactoring known to resolve other security smells, as recapped earlier in this section.

2.1.7 *Centralized authorization*

This smell occurs when a microservice application only handles authorization in one component, typically at the gateway of the application, while it does not enact any fine-grained authorization control at the microservices level. Such centralized authorization may result in authenticity violations, since microservices would trust the gateway based on its mere identity.

The suggested refactoring is using decentralized authorization, by transmitting an access token with each request, e.g., a JSON Web Token. The token provides a mechanism to safely pass user claims or data, together with a digital signature that guarantees its authenticity. Authorization can then be enforced also at the microservices level, as it gives each microservice more control to enforce its own access-control policies.

2.2 Softgoal Interdependency Graphs

Softgoal Interdependency Graphs facilitate the systematic modeling of the impact of design decisions (called *operationalizing softgoals*) on quality attributes (called *softgoals*). Specifically, they enable the representation of both positive and negative impacts of different design decisions on quality attributes, hence providing visual support for application architects to start a trade-off analysis process [12].

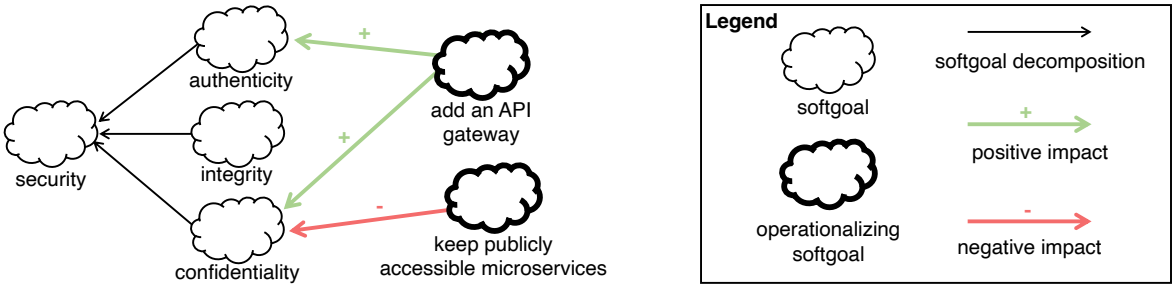


Figure 1: Example of a Softgoal Interdependency Graphs

Figure 1 presents a Softgoal Interdependency Graph exemplifying the impacts on *security* of two design decisions, namely, the operationalizing softgoals *keep publicly accessible microservices* or *add an API gateway*. Security is decomposed into three different security properties (viz., the softgoals *confidentiality*, *integrity*, and *authenticity*). The positive or negative impacts of each design decision are represented by green or red arrows, respectively. Impacts are also labeled with `+` or `-` to denote that a design decision helps to achieve or hurts a quality attribute. Other possible labels are `++` or `--` to denote that a design decision unequivocally ensures or compromises a quality attribute.

In previous work [8], we introduced a method for conducting trade-off analyses utilizing Softgoal Interdependency Graphs (SIGs) [12]. SIGs offer a comprehensive visual representation of the positive or negative impacts of each security smell and refactoring on individual software quality attributes and microservices' key design principles. In this paper, we will use the method introduced in [8] to illustrate the validated impacts of each security smell and refactoring.

3 Survey Design

We aim to complement the analysis of microservices' security smells and refactoring in our previous work [5], by assessing the possible impacts of both smells and refactorings on other properties than security. More precisely, we focus on two quality properties defined by the ISO 25010 standard [7], viz., *performance efficiency* and *maintainability*, which are crucial to microservices [4], as well as on the adherence of an application to microservices' key design principles [6]. We hereafter illustrate how we selected the primary

studies from which to extract such possible impacts, as well as the process we enacted to analyze and assess such impacts.

3.1 Literature selection

The 55 white/grey primary studies providing the state-of-the-art/state-of-practice on smells and refactorings for microservices security were already identified and classified in [5], by following the guidelines for conducting systematic literature reviews in [13], combined with those in [14] for systematically reviewing grey literature. We hence started from such 55 primary studies and the smells and refactorings discussed therein, as shown in Figure 2

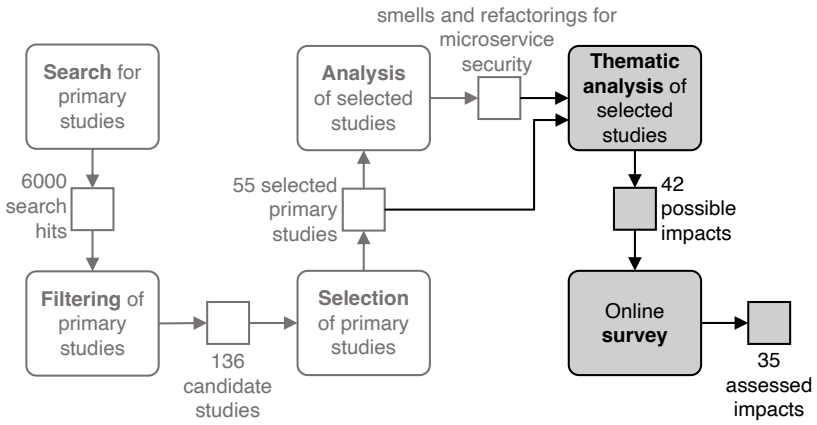


Figure 2: Research process. Grey boxes denote steps to elicit and assess possible impacts of smells and refactorings identified in our previous work [5] (whose steps are in white)

3.2 Thematic analysis

We analyzed the selected primary studies to elicit the possible impacts of the smells and refactorings from [5] on performance efficiency, maintainability, and adherence to microservices' key design principles.

The analysis was enacted by adopting thematic coding [9] and Krippendorf Kα-based inter-rater reliability assessment [10]. The first two authors annotated and labeled the selected primary studies to elicit possible impacts of smells and refactorings on the considered properties. The annotation and labeling were executed in parallel over two complementary partitions of the selected primary studies, to reduce potential observer biases. The coders were then switched to evaluate the inter-rater agreement on the two emerging lists of impacts. The inter-rater agreement was again measured by exploiting the Kα coefficient [10] to determine the agreement between the first two authors (who independently coded their partitions) on the emerging lists of bad and good practices for microservice security. The measured agreement was already higher than 80%, which is typically taken as a reference score for inter-rater agreement [10].

A final triangulation step was then performed to reduce potential biases further. The last three authors cross-checked the coding performed by the first two authors, with no prior information on the coding itself. This concluded our analysis process, which resulted in a total of 42 possible impacts, which are listed and discussed in Section 4.

3.3 Online survey

To further assess the identified impacts and understand whether industrial practitioners and academic researchers perceive them differently, we distributed an online survey to ask whether they agree with them. The survey was structured with initial profiling questions, used to distinguish industrial practitioners and academic researchers, and to ensure that respondents have experience in working with microservices. If this was the case, respondents were exposed to 42 statements, each presenting a different identified impact and its rationale, and they were asked to explicitly select their agreement with the statement, viz., *strongly disagree*, *disagree*, *neutral*, *agree*, or *strongly agree*.

The survey was published online from November 2022 to March 2023, collecting answers from a total of 21 respondents who explicitly declared that they were working with microservices. Out of such 21 respondents, 13 were industrial practitioners with different roles (1 software developer, 4 software engineers,

Table 1: Survey participants information

Role	3+ years of exp. with MS	1-3 years of exp. with MS
Software Architect	8	0
Software Engineer	0	4
Software Developer	0	1
Researcher	5	4

and 8 software architects), 8 of whom declared 3+ years of experience in working with microservices. The other 8 respondents were academic researchers, 5 of whom declared 3+ years of experience working with microservices[1]. Table 1 contains the details of the participants' information, indicating their current role and the distribution based on their years of experience with microservices.

4 Impacts of Security Smells

We hereafter present the candidate impacts of the security smells and refactorings for microservices, by considering one smell from our previous work [5] at a time. We also show whether/how interviewed experts agreed with such statements. The agreement will be displayed with bar plots, in which green is used to denote agreement, gray to denote neutrality, and yellow to denote disagreement. Darker shades of green/yellow denote stronger agreement/disagreement.

Also, we consider a candidate impact as confirmed if the absolute majority of respondents explicitly agree with it. At least 12 respondents must therefore agree or strongly agree with a candidate impact for the latter to be confirmed.

4.1 Insufficient Access Control (IAC)

IAC occurs when microservices lack proper access control, and its effects can be mitigated by exploiting OAuth 2.0. Indeed, IAC is known to negatively impact the confidentiality of a microservice application, whereas the use of OAuth 2.0 is positively impacting on it [5].

Five other possible impacts of IAC and the use of OAuth 2.0 emerged from the analyzed literature. These are listed in Table 2, with `IAC1-2` predicating over the IAC smell itself, while `IAC3-5` predicate over its possible refactoring. Each statement `IAC1-5` highlights the possible impact, the affected property, and the rationale behind the impact.

Table 2: Possible impacts of IAC

ID	Statement
`IAC1`	Insufficient access control may deteriorate the isolation of failures since an attacker can exploit a compromised microservice to provoke additional failures
`IAC2`	Insufficient access control may deteriorate the analysability of the application, making it more complex to diagnose the extension of damage (e.g., due to the confused deputy problem)
`IAC3`	OAuth 2.0 may increase the adherence to the decentralization principle of microservices, being it a distributed access delegation protocol
`IAC4`	OAuth 2.0 may increase the resource utilization of the application since, with this protocol, more information is transmitted on each request
`IAC5`	OAuth 2.0 may increase the reusability of the application since the organization can rely on standard libraries and platforms compatible with this industry-standard protocol

The agreement of respondents with the statements `IAC1-5` is plotted in Figure 3. From the figure, we can observe that the agreement with `IAC1`, `IAC2`, and `IAC3` is above 75%. Thus, most of the interviewed practitioners and researchers confirmed that IAC may deteriorate the isolation of failures and analysability of microservice applications, and that the use of OAuth 2.0 helps to achieve the microservices' decentralization principle.

The majority of interviewed practitioners and researchers also agree with `IAC5`, confirming that the use of OAuth 2.0 can increase the reusability of the application. The answers for this statement are shown in Figure 4(b), from which we can observe practitioners mostly agree with `IAC5`, with a tendency towards strong agreement. Researchers also overall agree, but the tendency is more towards light agreement/neutrality.

`IAC4` is instead not confirmed, as the overall agreement is lower than 50% (see Figure 3), meaning that respondents do not agree with OAuth 2.0 increasing the resource utilization of microservice applications.

[1] All collected responses are publicly available on Zenodo at `https://doi.org/10.5281/zenodo.7828029`

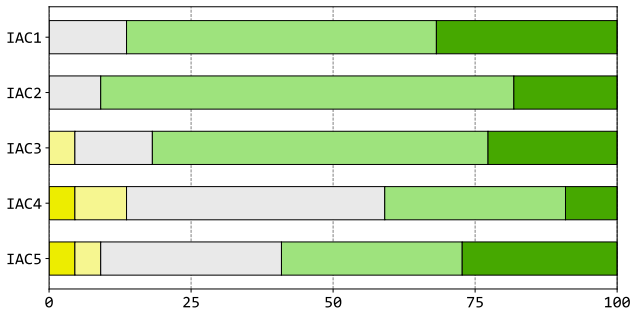
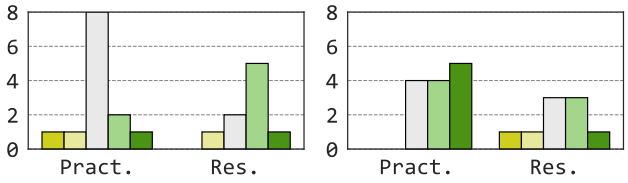


Figure 3: Agreement with the possible impacts of IAC

Figure 4(a) shows the actual answers given by practitioners and researchers. From the figure, we can observe that practitioners are mostly neutral about the statement, while researchers mostly agree with it.



(a) IAC4 (b) IAC5

Figure 4: Distribution of agreement with IAC-related statements.

Finally, fig. 5 provides a comprehensive visual representation of the positive and negative impacts of the Insufficient Access Control (IAC) security smell and its corresponding refactoring (Use OAuth 2.0).

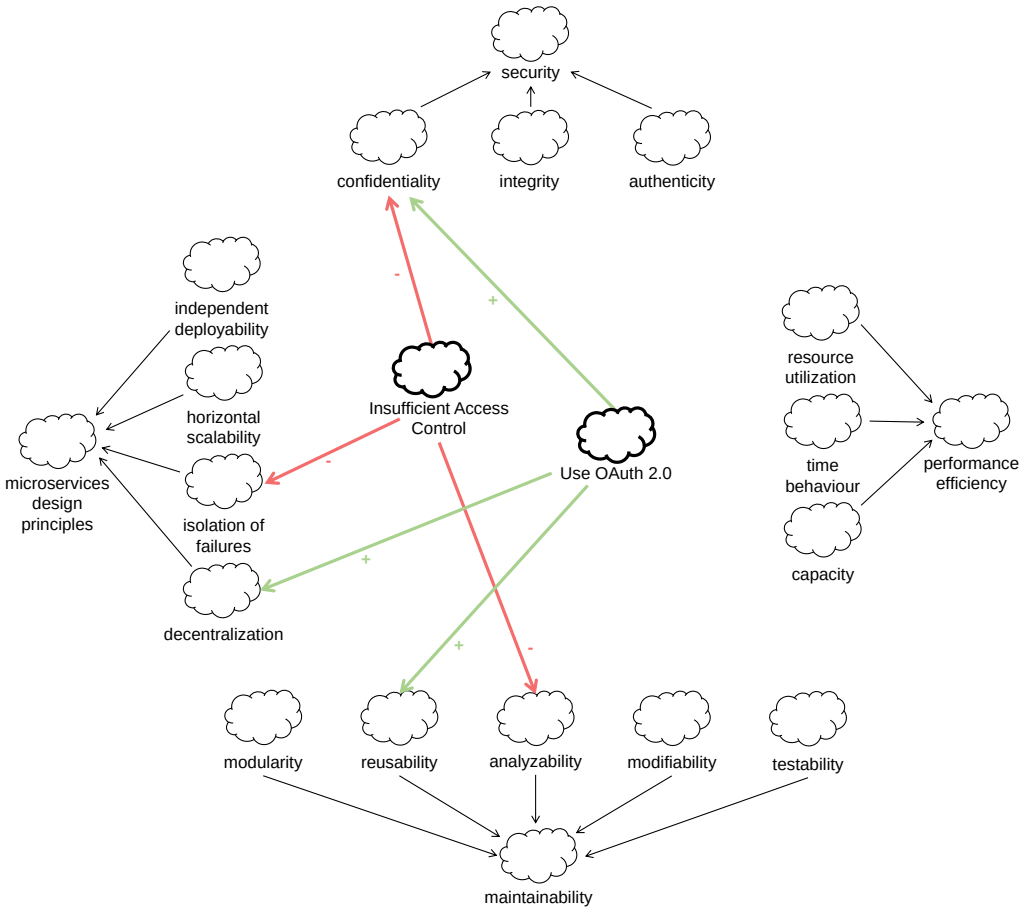


Figure 5: SIG that provides a holistic view of the validated impacts related to IAC

4.2 Publicly Accessible Microservices (PAM)

PAM occurs when microservices are directly accessible by external clients, and this can negatively impact on the confidentiality of a microservice application [5]. The effects of PAM can be mitigated by adding an API Gateway, to be used as an entry point for the application.

Nine other possible impacts of PAM and its refactoring emerged from the analyzed literature. These are listed in Table 3, which states the possible impacts, the property that its affected, and the rationale behind it. In the figure, PAM1, PAM3, and PAM8 predicate over the PAM smell itself, while the others predicate over its refactoring.

Table 3: Possible impacts of PAM

ID	Statement
PAM1	Having multiple publicly accessible microservices may deteriorate the testability of an application since there are multiple access points, each to be tested independently
PAM2	Adding an API Gateway may improve the testability since changes related to access to the application need to be tested only in the Gateway rather than on multiple entry points (being these services or other components)
PAM3	Having multiple publicly accessible microservices may deteriorate the isolation of failures of an application since they increase the attack surface which can be exploited to cause cascading failures
PAM4	Adding an API Gateway may increase the resource utilization of the application since additional resources are needed to actually run the API Gateway
PAM5	Adding an API Gateway may increase the time behavior of the application since it introduces additional communications/interactions between the Gateway itself and the other components
PAM6	Adding an API Gateway may improve the modifiability of an application since changes in how to access the application can be applied only to the Gateway, hence minimizing the impact on the other components
PAM7	Adding an API Gateway may improve the reusability of an application since the organization can rely on existing solutions for implementing a Gateway or reuse its own one for other applications
PAM8	Having multiple publicly accessible microservices may deteriorate the modifiability of an application, since a change in how to access the application may require applying such change to all publicly accessible microservices
PAM9	Adding an API Gateway may improve the analysability of an application since it simplifies evaluating the impact of changes in how to access the application (compared to the case of having multiple publicly accessible microservices)

The agreement of respondents with the statements PAM1-9 is plotted in Figure 6. From the figure, we can observe that all respondents agree with PAM7, confirming that adding an API Gateway improves the reusability of microservice applications. Adding an API Gateway also improves the modifiability and analysability of microservice applications, as witnessed by the large agreement with PAM6 and PAM9. The interviewed practitioners and researchers also largely agree with PAM3 and PAM8, hence confirming that the presence of PAMs may deteriorate the isolation of failures and the modifiability of a microservice application.

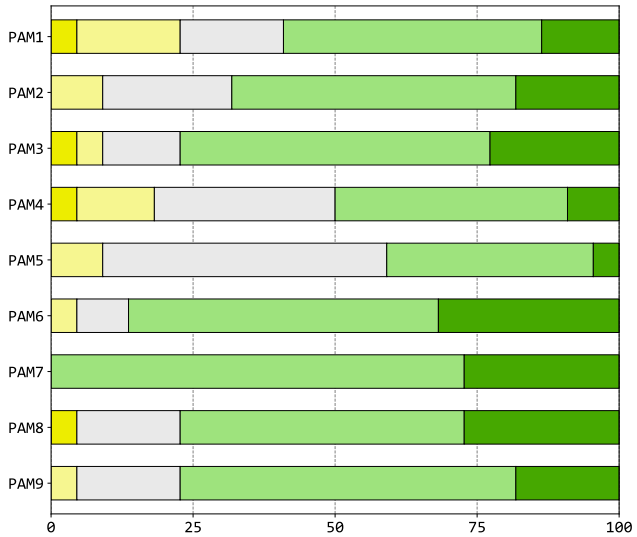


Figure 6: Agreement with the possible impacts of PAM

The majority of interviewed practitioners and researchers also agree with PAM1 and PAM2, confirming that having publicly accessible microservices may deteriorate the testability of an application, while adding an API Gateway may improve it. However, being the agreement with PAM1 and PAM2 lower than 75%, they deserve a closer look to check for possible differences in practitioners' and researchers' answers. These are shown in Figure 7(a-b), from which we can observe that most practitioners and researchers are aligned in mostly agreeing with both statements.

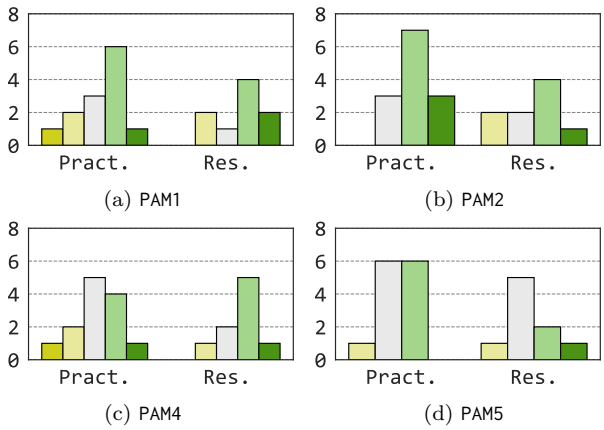


Figure 7: Distribution of agreement with PAM-related statements.

PAM4 is instead not confirmed, as the overall agreement is exactly 50% and we only consider impacts as confirmed only if the absolute majority of respondents explicitly agree. Figure 7(c) shows the distribution of answers given by interviewees. From the figure, we can observe that practitioners are more neutral than researchers, who mostly agree. So, there is not enough support from practitioners to confirm that adding an API Gateway would negatively impact on the resource utilization of an application.

PAM5 is not confirmed either, as the overall agreement is lower than 50%, mainly because of neutral opinions on the statement (Figure 6). Figure 7(d) shows the actual answers given by interviewees, by distinguishing between practitioners and researchers. From the figure, we can observe that practitioners are mostly divided between neutrality and agreement, while researchers are mostly neutral with PAM5. This means that, overall, respondents do not agree that adding an API Gateway may increase the time behavior of an application.

Finally, fig. 8 provides a comprehensive visual representation of the positive and negative impacts of the Publicly Accessible Microservices (PAM) microservice security smell and its corresponding refactoring.

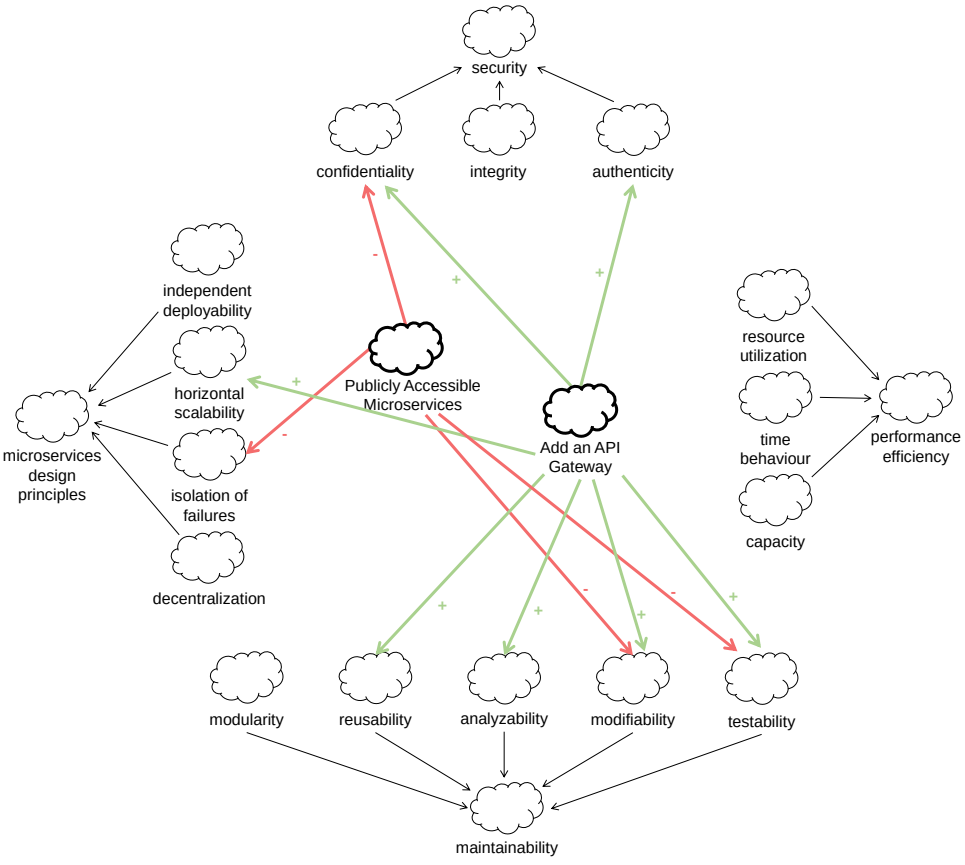


Figure 8: SIG that provides a holistic view of the validated impacts related to PAM

4.3 Unnecessary Privileges to Microservices (UPM)

UPM occurs when microservices are granted unnecessary privileges (e.g., access levels, permissions, or functionalities) that are not needed to deliver their business functions, and this may negatively impact on the confidentiality and the integrity of microservice applications. The impact can be reversed, if developers rather follow the least privilege principle [5].

Eight other possible impacts of UPM and its refactoring emerged from the analyzed literature. These are listed in Table 4, with UPM1, UPM3, and UPM5 predicating over the UPM smell itself, while the others predicate over following the least privilege principle.

Table 4: Possible impacts of UPM

ID	Statement
UPM1	Providing unnecessary privileges to microservices may deteriorate the resource utilization of an application since an attacker can exploit a compromised microservice to increase the resources consumed by the application.
UPM2	Following the Least Privilege Principle may improve the modularity of an application since it reduces the resources that a microservice can access, hence meaning that a change in a microservice could impact only on such resources.
UPM3	Providing unnecessary privileges to a microservice may deteriorate the analysability of an application since they complicate the impact of a change in such microservices on the rest of the application.
UPM4	Following the Least Privilege Principle may improve the isolation of failures in an application since it reduces the resources that a microservice can access, hence also reducing the services that can fail in cascade to a compromised microservice.
UPM5	Providing unnecessary privileges to microservices may deteriorate the isolation of failures in an application since an attacker can exploit the unnecessary privileges to cause failures in other microservices.
UPM6	Following the Least Privilege Principle may improve the analysability of an application since it reduces the resources that a microservice can access, hence limiting the analysis of the effects of a change in a microservice only to such resources.
UPM7	Following the Least Privilege Principle may improve the resource utilization of an application since access to a resource is provided to microservices only on an as-needed basis (hence reducing the waste of resources due to misuse).
UPM8	Following the Least Privilege Principle may deteriorate the testability of an application since it is more complex to test whether a microservice can access only the resources it actually needs.

The agreement of interviewed practitioners and researchers with the statements UPM1-8 is plotted in Figure 9. From the figure, we can observe that the agreement with UPM1 and UPM5 is above 75%, meaning that the UPM smell may deteriorate the resource utilization and isolation of failures of microservice applications. Respondents also overall agree with UPM4 and UPM7, which indicates that following the least privilege principle may improve the isolation of failures and resource utilization.

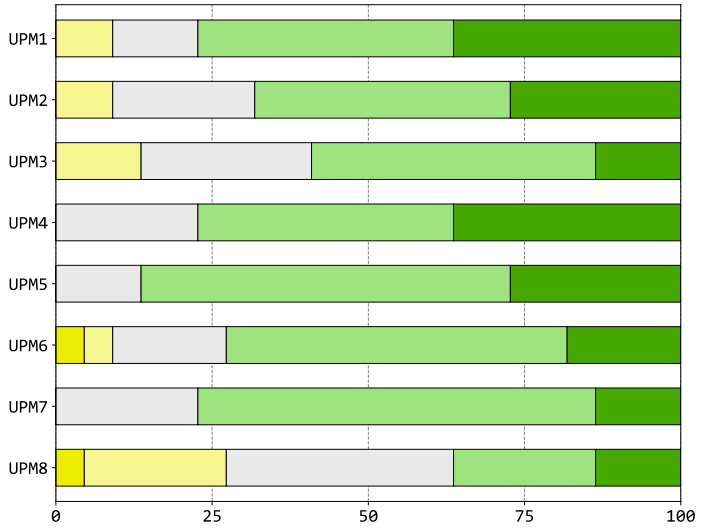


Figure 9: Agreement with the possible impacts of UPM

The majority of practitioners and researchers also agree with UPM2, UPM3, and UPM6, confirming that refactoring an application by following the least privilege principle improves the application's modularity and

analysability, whereas the UPM smell deteriorates its analysability. Figure 10(a-c) show that practitioners and researchers are aligned in mostly agreeing with these three statements.

UPM8 instead is not confirmed, as the overall agreement is lower than 50%, mainly because of the neutral opinions from both practitioners and researchers on the statement. From Figure 10(d) we can observe that practitioners are mostly divided between neutrality, disagreement, and agreement, while researchers are mostly neutral. This means that respondents do not overall agree that following the least privilege principle may deteriorate the testability of microservice applications.

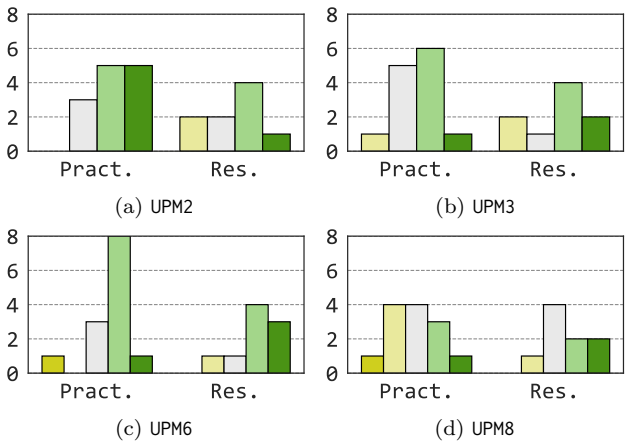


Figure 10: Distribution of agreement with UPM-related statements.

Finally, fig. 11 then provides a comprehensive visual representation of the positive and negative impacts of the Unnecessary Privileges to Microservices (UPM) security smell and its corresponding refactoring.

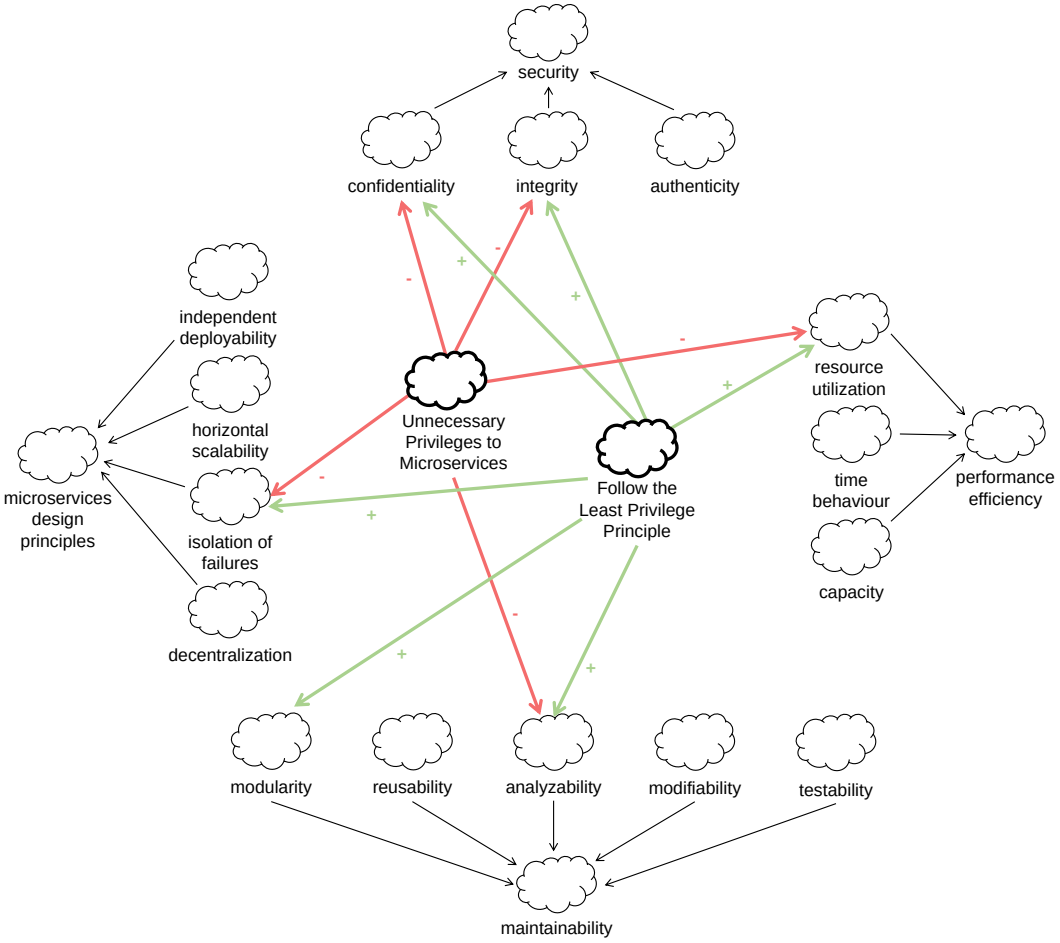


Figure 11: SIG that provides a holistic view of the validated impacts related to UPM

4.4 Non-Secured Service-to-Service Communications (NSC)

NSC occurs when some microservices in an application interact without establishing a secure communication channel, and its effects can be mitigated by using Mutual TLS. NSC is a security smell that negatively impacts on the confidentiality, integrity, and authenticity of microservice applications, whereas its refactoring has a positive impact on them [5].

Table 5: Possible impacts of NSC

ID	Statement
NSC1	Non-secured service-to-service communications may deteriorate the analysability of an application, making it harder to diagnose the causes of failures (e.g., since data in transit can be modified if a service is compromised).
NSC2	Mutual TLS may improve the isolation of failures of microservices since this protects service-to-service communication from, e.g., man-in-the-middle, eavesdropping, and tampering attacks.
NSC3	Non-secured service-to-service communications may deteriorate the isolation of failures in an application since an attacker can exploit them to propagate failures across microservices.
NSC4	Mutual TLS may deteriorate the time behavior of the application since it encrypts service-to-service communications.

Other four possible impacts of NSC and its refactoring are listed in Table 5. In the figure, NSC1 and NSC3 predicate over NSC itself, while NSC2 and NSC4 pertain to its Mutual TLS-based refactoring. Figure 12 then shows the level of agreement among respondents concerning statements NSC1-4, with NSC3 confirmed by the vast majority of them, hence NSC may contribute to deteriorate isolation of failures in a microservice application.

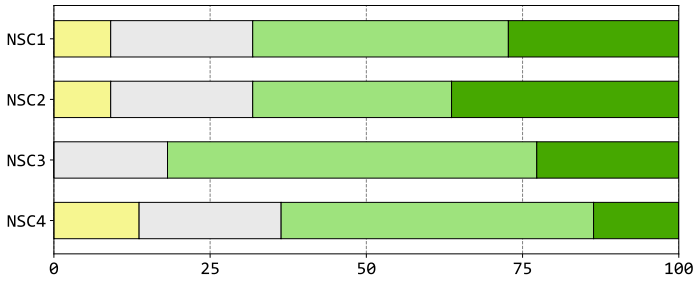
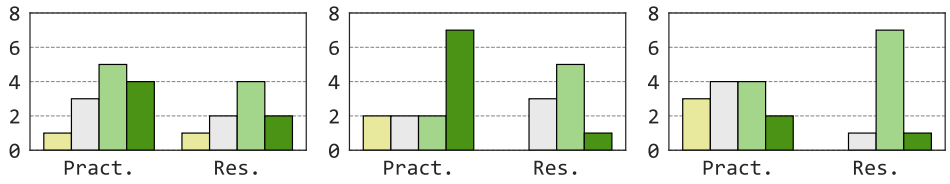


Figure 12: Agreement with the possible impacts of NSC

The majority of practitioners and researchers also agree with NSC1, which confirms the negative impact of the NSC smell on the analysability of microservice applications. The agreement also is reached for NSC2 and NSC4, confirming that the use of Mutual TLS improves an application's isolation of failures, at the price of negatively impacting on its time behavior. As observed in Figure 13, practitioners and researchers are mostly aligned in their agreement with these three statements, with practitioners mostly strongly agreeing with NSC2.



(a) NSC1 (b) NSC2 (c) NSC3

Figure 13: Distribution of agreement with NSC-related statements.

Finally, fig. 14 provides a comprehensive visual representation of the positive and negative impacts of the Non-Secured Service-to-Service Communications (NSC) security smell and its corresponding refactoring.

4.5 Unauthenticated Traffic (UNT)

UNT occurs when a microservice receives unauthenticated requests from external clients or other microservices of the application, which may negatively impact on the authenticity of microservice applications. The effects of UNT can be mitigated by using Mutual TLS and OpenID Connect [5].

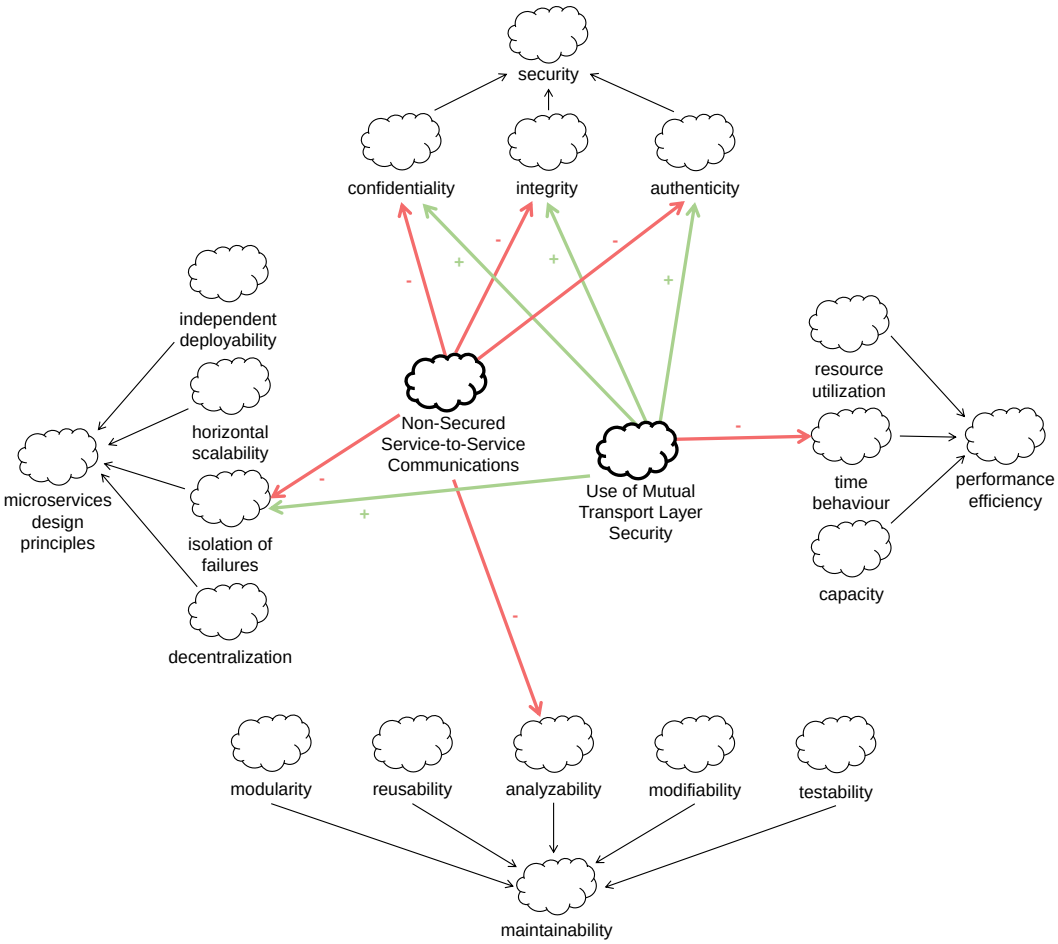


Figure 14: SIG that provides a holistic view of the validated impacts related to NSC

In addition to the already discussed impacts of using Mutual TLS (Section 4.4), three additional impacts emerged from the analyzed literature. These impacts are listed in Table 6, with UNT1 predicating over UNT itself, while UNT2 and UNT3 pertain to the use of OpenID Connect.

Table 6: Possible impacts of UNT

ID	Statement
UNT1	Having unauthenticated traffic may deteriorate the isolation of failures since an attacker can generate (unauthenticated) traffic to propagate failures across microservices.
UNT2	Using OpenID Connect may increase the adherence to the decentralization principle of microservices, being it, a distributed identity mechanism allowing to decentralize authentication.
UNT3	Using OpenID Connect may improve the reusability of the application since it can be realized by reusing existing software whose configuration is easy to reuse/replicate.

The agreement with the UNT statements is then shown in Figure 15. From the figure, we can observe that the agreement with UNT1 is above 75%, which indicates that having unauthenticated traffic in a microservice application can lead to the deterioration of isolation of failures.

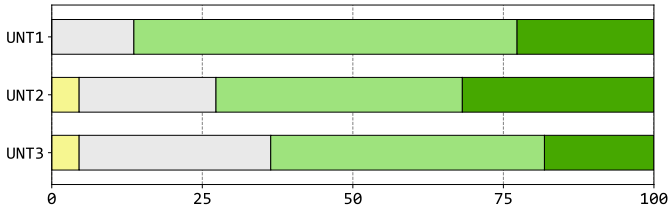
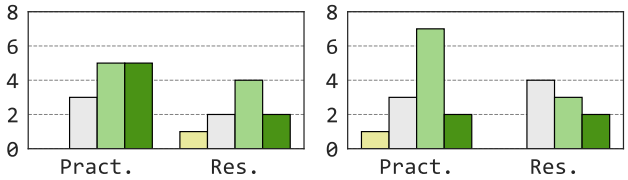


Figure 15: Agreement with the possible impacts of UNT

UNT2 and UNT3 are also confirmed by the majority of interviewees, meaning that OpenID Connect positively impacts on reusability and decentralization in microservice applications. By looking at Figure 16, we can observe that both practitioners and researchers are mostly aligned in their agreement with UNT2, whereas UNT3 is mostly agreed by practitioners, with researchers having a more neutral reaction to such statement.



(a) UNT2 (b) UNT3

Figure 16: Distribution of agreement with UNT-related statements.

Figure 17 then provides a comprehensive visual representation of the positive and negative impacts of the Unauthenticated Traffic (UNT) security smell and its corresponding refactoring.

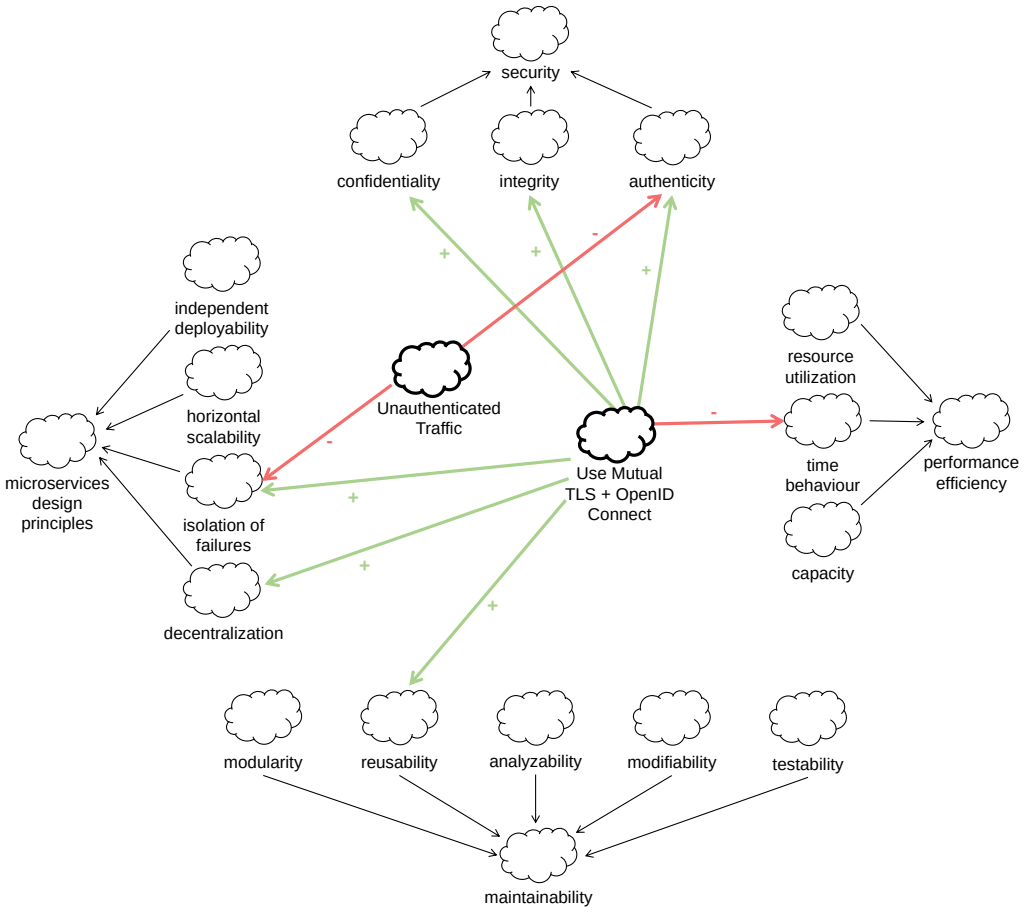


Figure 17: SIG that provides a holistic view of the validated impacts related to UNT

4.6 Multiple User Authentication (MUA)

MUA occurs when a microservice application provides multiple endpoints for user authentication, and its effects can be mitigated by using a single sign-on approach. MUA is a security smell that may negatively impact on the authenticity of microservice applications, whereas its refactoring has a positive impact on confidentiality and authenticity [5].

Three additional possible impacts of MUA have emerged from the analyzed literature, all predicating over the MUA smell itself (Table 7). The agreement among interviewees concerning MUA statements is shown in Figure 18, which confirms MUA1 and MUA2, namely that MUA may deteriorate the testability and modifiability of microservice applications. Such agreement mainly comes from researchers in the case of MUA1

Table 7: Possible impacts of MUA

ID	Statement
MUA1	Authenticating users in multiple different services may deteriorate the testability of an application since the multiple user authentication points should be tested independently.
MUA2	Authenticating users in multiple different services may deteriorate the modifiability of an application since a change related to user authentication should be applied to all user authentication points.
MUA3	Having multiple user authentication points may increase the adherence to the decentralization principle of microservices since access to the application is distributed across all entry points.

(Figure 19(a)), whereas practitioners and researchers are aligned in agreeing with MUA2 (Figure 19(b)). MUA3 instead is not confirmed, as the overall agreement falls below 50% (Figure 18). This is mainly because the vast majority of practitioners disagree or are neutral to MUA3, as opposed to researchers who mostly agree with such a statement.

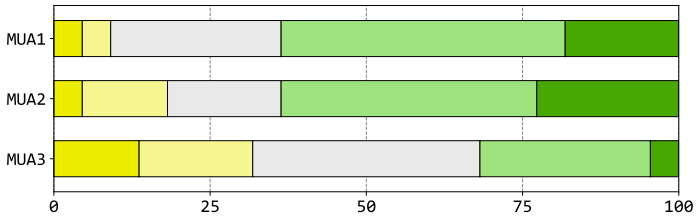
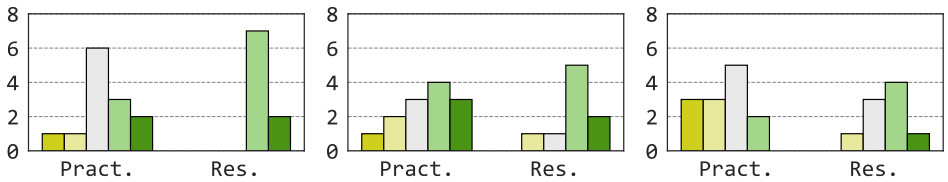


Figure 18: Agreement with the possible impacts of MUA



(a) MUA1 (b) MUA2 (c) MUA3

Figure 19: Distribution of agreement with MUA-related statements.

Finally, Figure 20 provides a comprehensive visual representation of the positive and negative impacts of the Multiple User Authentication (MUA) security smell and its corresponding refactoring.

4.7 Centralized Authorization (CNA)

CNA occurs when a microservice application only handles authorization in one component, and its effects can be mitigated by using a decentralized authorization approach (e.g., transmitting a token with each request). CNA negatively impacts the authenticity of microservice applications, whereas its refactoring has a positive impact on it [5].

Ten additional possible impacts of CNA and its refactoring have emerged from the analyzed literature. These impacts are listed in Table 8, with CNA1, CNA3, CNA7 and CNA9 focusing on the CNA smell, while the others on using a decentralized authorization approach. The agreement with CNA-related impacts is illustrated in Figure 21. From the figure, we can observe that the vast majority of interviewed practitioners and researchers agree with CNA2, hence confirming that – as expected – decentralizing authorization contributes to realizing the decentralization design principle of microservices.

Respondents also mostly agree with CNA1 and CNA9, confirming the negative impact of CNA on an application's time behavior and decentralization, whereas the agreement with CNA7 confirms that CNA can improve the testability of an application. From Figure 22, we can observe that the agreement with CNA1, CNA7, and CNA9 comes both from practitioners and from researchers, who are aligned in mostly agreeing with such statements.

As shown in Figure 21(a), the majority of respondents also agreed with CNA4, CNA5, CNA8, and CNA10, which are about the refactoring of CNA. This confirms that decentralizing authorization helps improving the modularity of a microservice application and its adherence to the independent deployability principle of microservices, at the price of deteriorating the application's testability and increasing its resource utilization.

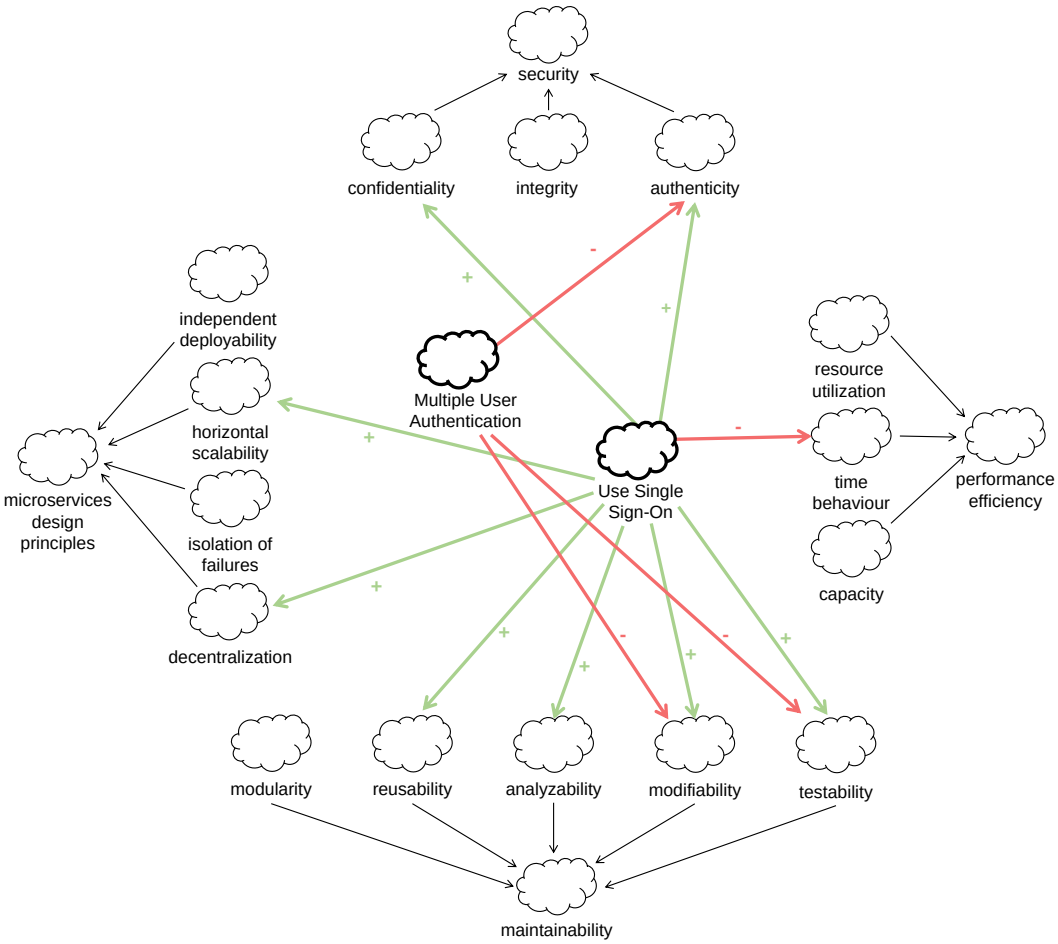


Figure 20: SIG that provides a holistic view of the validated impacts related to MUA

Table 8: Possible impacts of CNA

ID	Statement
CNA1	A centralized authorization may deteriorate the time behavior of an application, with the authorization server becoming a bottleneck when many microservices access it.
CNA2	Using decentralized authorization may increase the adherence to the decentralization principle of microservices, since access control is distributed across the application, with each microservice having its fine-grained access controls.
CNA3	A centralized authorization may deteriorate the analysability of an application, since it requires extensively considering all possible impacts (on all services in the application) of a change in the access control ruled by the authorization server.
CNA4	Using decentralized authorization may improve the modularity of an application, since each microservice is associated with its own fine-grained access controls, hence limiting the impact of changes only to such microservice.
CNA5	Using decentralized authorization may deteriorate the testability of an application, since more efforts are needed to independently test the access control on each microservice.
CNA6	Using decentralized authorization may improve the analysability of an application, since each microservice is associated with its own fine-grained access controls, hence requiring assessing the impact of changes only on such microservice.
CNA7	A centralized authorization may improve the testability of an application, since changes related to access control should be tested only in the central authorization server.
CNA8	Using decentralized authorization may increase the adherence to the independent deployability principle of microservices, since it avoids coupling microservices to an authorization server.
CNA9	A centralized authorization may reduce the adherence to the decentralization principle of microservices since authorization is centralized and fully managed by only one component.
CNA10	Using decentralized authorization may increase resource utilization, since more information is to be transmitted on each request (e.g., token containing roles and privileges).

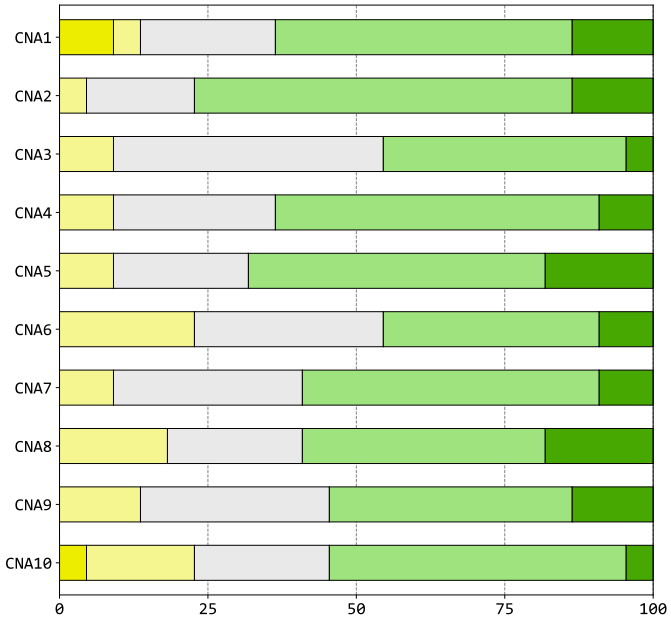


Figure 21: Agreement with the possible impacts of CNA

Again, the agreement with `CNA4`, `CNA5`, `CNA8`, and `CNA10` comes both from practitioners and from researchers, who are aligned in mostly agreeing with such statements.

The candidate impacts `CNA3` and `CNA6` are instead not confirmed, meaning that CNA and its refactoring are not considered to impact on the analysability of a microservice application. For `CNA3`, this is mainly because of the neutrality of practitioners, with researchers' answers being more distributed among disagreement, neutrality, and agreement (Figure 22(b)). In the case of `CNA6`, both practitioners' and researchers' answers are divided among disagreement, neutrality, and agreement, with a tendency towards agreement from practitioners and a tendency towards neutrality/disagreement for researchers (Figure 22(e)).

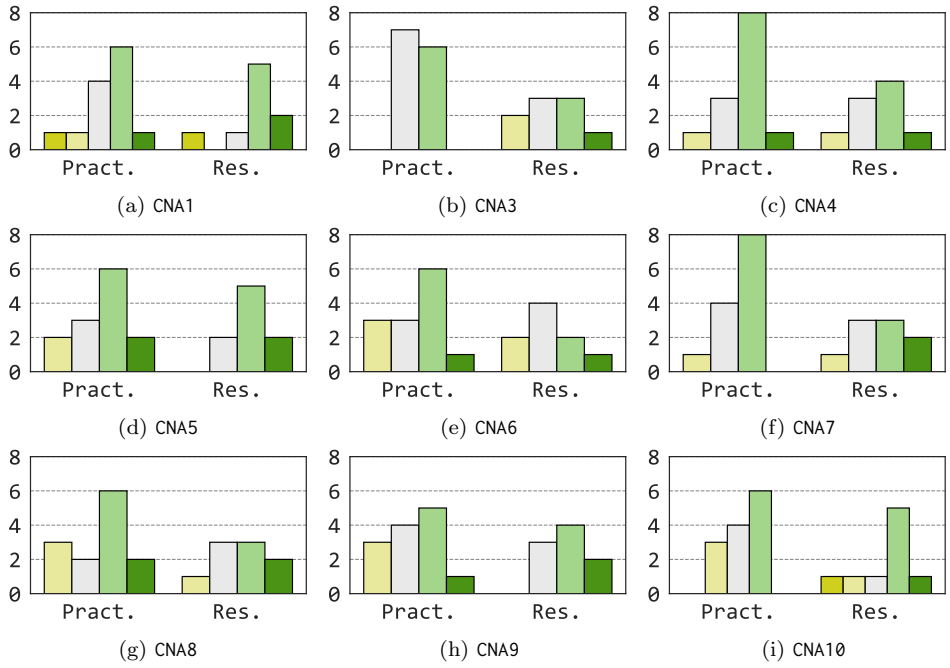


Figure 22: Distribution of agreement with CNA-related statements

Finally, fig. 23 provides a comprehensive visual representation of the positive and negative impacts of the Centralized Authorization (CNA) security smell and its corresponding refactoring.

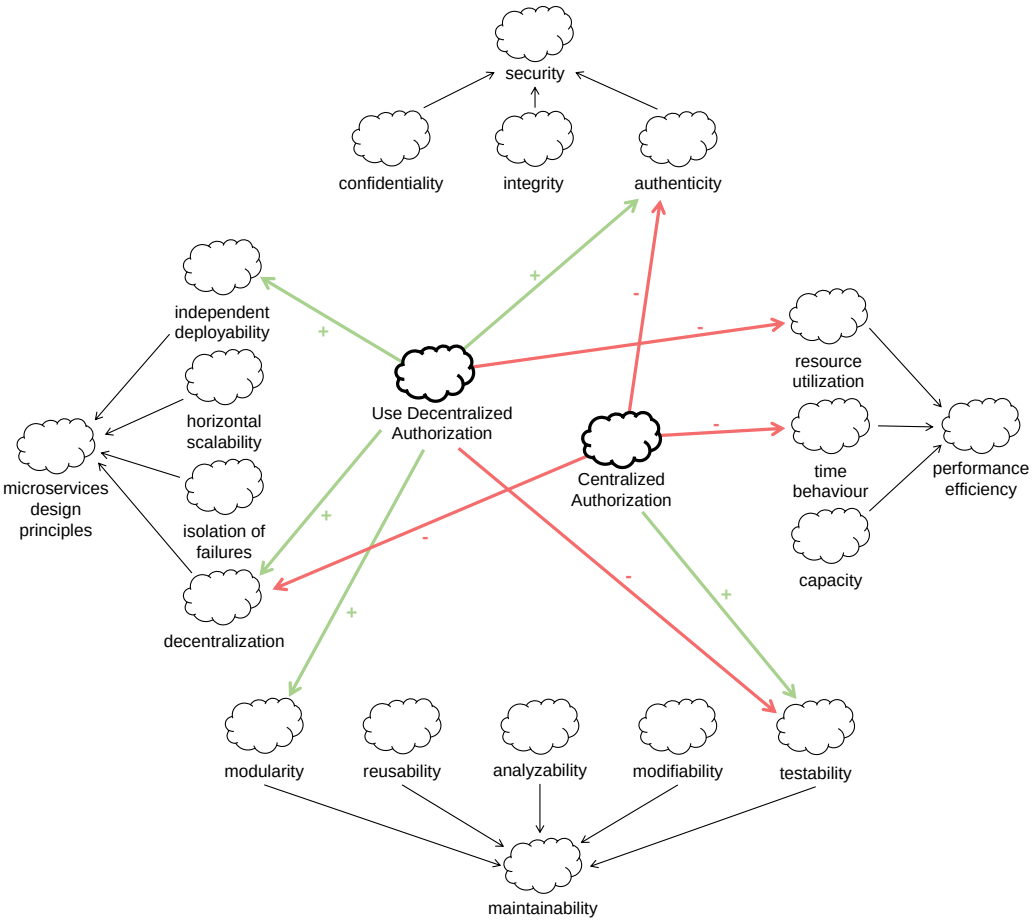


Figure 23: SIG that provides a holistic view of the validated impacts related to CNA

5 Discussion

We hereby provide further insights on how to possibly use the SIGs modeling the security smells' impacts confirmed with our study (Section 5.1). We also elicit and discuss the possible threats to the validity of our study (Section 5.2).

5.1 Using the SIGs with Confirmed Impacts

The SIGs modeling the confirmed impacts provide a holistic visualization of the confirmed positive/negative impacts of microservice security smells and its refactoring. Such a visualization enables the trade-off analysis that we envisioned in [8]. The SIGs can indeed help reason on whether it is worth applying a refactoring or whether it is more convenient to keep a smell, based on their respective impacts on quality attributes and microservices' key design principles.

For example, suppose that a microservice is affected by a *centralized authorization* smell, whose SIG is displayed in Figure 23. We can start by defining which of the softgoals in the SIG is more important for the affected microservice, and this actually depends on the context and desiderata of the MSA to which the microservice pertains. Suppose, for instance, that *resource utilization* is a priority, e.g., since the microservice is expected to run on a device with limited resources, or because increasing its resource utilization may get too costly. If this is the case, given that applying the *use decentralized authorization* refactoring negatively impacts on *resource utilization*, we may end up deciding that keeping the *centralized authorization* smell is more convenient. Instead, if we would have no restrictions on *resource utilization*, we may consider applying the refactoring, especially if *testability* is less important than the softgoals positively impacted by the *use decentralized authorization* refactoring (Figure 23). A similar reasoning can be carried out with any security smell affecting a microservice in an MSA, obviously adapted to the SIGs defining the smell's impacts.

Additionally, the confirmed impacts presented in this article provide a knowledge base that can be used for further analyses. For example, we effectively used them to devise TriSS [15], a method to systematically triage the security smells affecting the microservices in an MSA. Borrowing from hospital emergency room triage practices, which assign an urgency code to incoming patients, TriSS enables assigning each security

smell with an urgency code, denoting how urgent is to intervene on such smell. The urgency code is obtained by combining the affected microservices' business relevance with the smells' confirmed impacts on security, performance efficiency, and maintainability.

5.2 Threats to Validity

Among the potential threats to validity classified by Wohlin et al. [16], four may apply to our survey. These are the threats to the *external*, *internal*, *construct*, and *conclusions* validity, which we discuss hereafter.

External Validity. The external validity concerns the applicability of a set of results in a more general context [16]. Since the primary studies considered by our thematic analysis were selected from a very large extent of online sources, the elicited impacts of security smells and refactorings may only be partly applicable to the state-of-the-art and state-of-practice on microservices. This may hence result in threatening the external validity of our study.

Actions to mitigate this potential threat were already taken in the literature selection (e.g., to avoid missing relevant literature), as we thoroughly described in our previous work [5]. To further reinforce the external validity of our findings, we organized multiple feedback sessions with all authors during our analysis of the selected studies, and we considered as validated *only* the impacts with which the majority of interviewed researchers/practitioners explicitly agreed. We indeed considered neutral opinions as non-agreement (rather than disagreement), as each neutral opinion was not providing enough evidence on the fact that an elicited impact was considered a true impact by a respondent. Also, because the analyzed literature was not covering some other impacts, we left blank space for respondents to propose impacts that were not emerging in our systematic analysis, getting very few suggestions — which were not enough to consider additional impacts to be added, even if we plan to further investigate on suggestion as part of our future work.

Construct and Internal Validity. The construct and internal validity of a study concern the generalizability of the constructs under study and the method employed to study and analyze data, respectively [16]. The construct and internal validity may hence be threatened by the potential types of bias involved when running a study, which we tried to mitigate by exploiting systematic analysis methods, such as thematic analysis and inter-rater reliability assessment (Section 3). Such systematic methods helped limiting potential biases, such as observer and interpretation biases, hence contributing to enhancing the validity of the analysis we performed on the data we retrieved.

Conclusions Validity. The conclusions validity is defined as the degree to which the conclusions of a study are reasonably based on the available data [16]. To mitigate potential threats to the conclusions' validity of our study, we exploited the above-described inter-rater reliability assessment to limit potential biases in our observations and interpretations. Additionally, we exploited the online survey to validate the elicited impacts, considering as valid impacts only those with which the majority of interviewed researchers/practitioners explicitly agreed.

6 Related Work

Various existing research works focus on microservice smells, e.g., [6], [17], [18], and [19]. For instance, the industrial survey reported in [17] explicitly defined 11 microservice-specific bad smells, which span from the design of the microservice applications to their actual development, also eliciting the best practices enabling to avoid incurring such smells. [6,18,19] instead mainly focus on architectural smells, geared towards checking whether the architecture of microservice applications adheres to the design principles that define microservices themselves. We here focus on security smells and their possible refactorings, which (to the best of our knowledge) have first been elicited in our previous work [5], which we, therefore, take as a starting point for our work.

Beyond the type of considered smells, it is worth relating our work with other research works exploiting surveys to shed light on microservices. In this perspective, the closest work to ours is [20], which also surveyed practitioners with questions on microservice security. The results of the survey in [20] highlight that microservices present unique security challenges. [20] also shows that access to security discussions (including design decisions, challenges, or solutions relating to security) is beneficial for making security decisions. Our work differs from [20] in its objectives, as we rather focus on the possible impacts of security smells and refactoring beyond security, namely on maintainability, performance efficiency, and adherence to microservices' key design principles.

Similar considerations apply to other studies exploiting surveys to shed light on microservices, such as, e.g., [21], [22], [2], and [23]. [21] explores the impacts of frameworks on the high availability of microservice applications, while [22] targets microservice-oriented maintainability assurance techniques. [2] and [23] instead focus on the commercial adoption of microservices and on migration from monoliths to microservices,

respectively. Our work hence differs from [21], [22], [2], and [23] in its objectives, as we focus on the impacts of security smells and refactorings on other quality attributes (namely, maintainability and performance efficiency) and on the adherence to microservices' key design principles.

7 Conclusions

We proposed an analysis of the impacts of security smells and refactorings on two other quality attributes that are crucial to microservices, viz., maintainability and performance efficiency, as well as on applications' adherence to microservices' key design principles. More precisely, we systematically elicited 42 possible impacts, which we validated through an online survey that confirmed 35 of such impacts. In analyzing the survey's results, we also commented on the answers given by interviewed practitioners and researchers, who were mostly aligned in agreeing with elicited impacts. A few misalignments emerged on possible impacts on testability and performance efficiency, with practitioners being more cautious in agreeing with them compared to researchers.

The elicited impacts can help decide whether to keep a security smell or apply a refactoring to mitigate its effects, based on their impacts beyond security itself. For this reason, and to feature the visualization needed to make informed decisions [8], we developed an open-source, visual tool that is publicly accessible online.[2] The tool visualizes the 35 validated impacts through the SIGs presented in Section 4. For future work, we plan to extend the tool into a full-fledged support for detecting security smells and reasoning on whether/how to refactor them, e.g., by integrating it with the security smell detection presented in [24], the trade-off analysis envisioned in [8], and the security smell triage method proposed in [15].

Additionally, while running the survey, the practitioners provided open feedback on other possible impacts of security smells. For instance, it was said that using "*OAuth 2.0 may potentially affect the time behavior of the application, as additional latency could be introduced due to token acquisition and validation processes*". All the possible impacts mentioned by practitioners are publicly available in the replication package published on Zenodo.[1] For future work, we plan to run a study (e.g., a survey with practitioners, similar to that presented in this article, or searching for examples/counterexamples) to confirm or rule out the impacts suggested by practitioners, as well as to double-check the exclusion of the impacts that were not confirmed by this study.

Acknowledgments This work was partially supported by ANID under grant `FONDECYT Regular 1201787` (Multimodal Machine Learning Approach for Selecting Pathological Activity Patterns in Elderlies); by project FREEDA (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU; and by project UNIPI PRA 2022 64 "OSMWARE", funded by the University of Pisa, Italy.

References

[1] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. [Online]. Available: https://doi.org/10.1109/MCC.2017.4250939

[2] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: an exploratory study," *Empirical Software Engineering*, vol. 26, no. 4, p. 63, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09910-y

[3] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, Jul 2017. [Online]. Available: https://doi.org/10.1007/s00450-016-0337-0

[4] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215 – 232, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2018.09.082

[5] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi, "Smells and refactorings for microservices security: A multivocal literature review," *Journal of Systems and Software*, vol. 192, p. 111393, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111393

[6] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: a multivocal review," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1, pp. 3–15, 2020. [Online]. Available: https://doi.org/10.1007/s00450-019-00407-8

[2] `https://ms-security.github.io`.

[7] ISO, "Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," ISO/IEC FDIS 25010, 2011, 1–34, 2011. [Online]. Available: https://iso25000.com/index.php/normas-iso-25000/iso-25010

[8] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi, "Should microservice security smells stay or be refactored? towards a trade-off analysis," in *Software Architecture: 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19–23, 2022, Proceedings*. Springer, 2022, pp. 131–139. [Online]. Available: https://doi.org/10.1007/978-3-031-16697-6_9

[9] T. Basit, "Manual or electronic? the role of coding in qualitative data analysis," *Educational Research*, vol. 45, no. 2, pp. 143–154, 2003. [Online]. Available: https://doi.org/10.1080/0013188032000133548

[10] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 2018. [Online]. Available: https://books.google.cl/books?id=nE1aDwAAQBAJ

[11] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, and A. Brogi, "To security and beyond: On the impacts of microservice security smells and refactorings," in *2023 XLIX Latin American Computer Conference (CLEI)*, 2023, pp. 1–10.

[12] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, "The NFR framework in action," in *Non-Functional Requirements in software engineering*. Springer, 2000, pp. 15–45. [Online]. Available: https://doi.org/10.1007/978-1-4615-5269-7_2

[13] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Technical Report EBSE-2007-01, 2007.

[14] V. Garousi, M. Felderer, and M. V. Mantyla, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.09.006

[15] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, and A. Brogi, "Triaging microservice security smells, with triss," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 698–706.

[16] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, ser. Computer Science. Springer Berlin Heidelberg, 2012. [Online]. Available: https://books.google.cl/books?id=QPVsM1_U8nkC

[17] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018. [Online]. Available: https://doi.org/10.1109/MS.2018.2141031

[18] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a collaborative repository for the documentation of service-based antipatterns and bad smells," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019. [Online]. Available: https://doi.org/10.1109/ICSA-C.2019.00025

[19] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2nd International Workshop on Refactoring*, ser. IWoR 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3242163.3242164

[20] A. Rezaei Nasab, M. Shahin, P. Liang, M. E. Basiri, S. A. Hoseyni Raviz, H. Khalajzadeh, M. Waseem, and A. Naseri, "Automated identification of security discussions in microservices systems: Industrial surveys and experiments," *Journal of Systems and Software*, vol. 181, p. 111046, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2021.111046

[21] G. Márquez, J. Soldani, F. Ponce, and H. Astudillo, "Frameworks and high-availability in microservices: An industrial survey." in *CIbSE*, 2020, pp. 57–70. [Online]. Available: https://cibse2020.ppgia.pucpr.br/images/artigos/2/S02_P2.pdf

[22] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service- and microservice-based systems," in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3194164.3194166

[23] L. Carvalho, A. Garcia, W. K. Assunção, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 2019, pp. 22–29. [Online]. Available: https://doi.org/10.1109/CESSER-IP.2019.00012

[24] P. Wizenty., F. Ponce., F. Rademacher., J. Soldani., H. Astudillo., A. Brogi., and S. Sachweh., "Towards resolving security smells in microservices, model-driven," in *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, INSTICC. SciTePress, 2023, pp. 15–26. [Online]. Available: https://doi.org/10.5220/0012049800003538