

Supporting the resolution of microservice security smells

Francisco Ponce^{a,*}, Jacopo Soldani^{b,*}, Gabriele Squarciapino^b,
Hernán Astudillo^c, Antonio Brogi^b

^a Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile

^b Dipartimento di Informatica, Università di Pisa, Pisa, Italy

^c ITiSB, Universidad Andrés Bello, Viña del Mar, Chile

ARTICLE INFO

Keywords:

Microservices security
Security smells
Refactoring
TriSS
Teamwise
Decision models

ABSTRACT

Microservice-based applications offer scalability and flexibility but introduce security challenges, including security smells, i.e., possible symptoms of poor security decisions. The occurrence of these smells can be resolved via known refactorings, which, however, impact not only security but also other quality attributes, such as maintainability and performance. Deciding which refactoring to implement first is complex due to service interdependencies and varying business priorities. In this paper, to support practitioners in deciding under these complex and dynamic conditions, we introduce a decision support method designed to account for these multifaceted constraints. The proposed approach offers a holistic perspective that considers the urgency of each security smell instance alongside the effort needed to concretely implement the refactoring and cross-team dependencies, ultimately enabling context-aware decisions. We also present a prototype tool implementing the proposed method and assess the perceived usefulness of our method through a preliminary empirical study following a one-group pretest-posttest design. After being introduced to the method, participants perceived the decision process as easier and reported greater confidence in their selected refactorings.

1. Introduction

Microservices-based applications (MSAs) enable the development of cloud-native applications [1], offering key advantages such as fault resilience and scalability [2]. At their core, MSAs are service-oriented applications that adhere to an extended set of design principles [3], including shaping services around business concepts, embracing decentralization, and ensuring the independent deployability and horizontal scalability of individual services. These additional principles make MSAs not only service-oriented but also highly distributed and dynamic in nature [4].

Along with their benefits, MSAs entail new challenges, and securing MSAs is certainly among them [5,6]. The possible security issues for MSAs include so-called *microservice security smells* [7]. These indeed denote the possible symptoms of bad decisions while designing or developing an MSA, which may lead to violations of authenticity, integrity, or confidentiality [7]. Security smells can be resolved by refactoring an MSA [8], but deciding whether to tolerate a given security smell or resolve it through refactoring requires careful trade-off considerations, as both the smell and its refactoring can impact other quality attributes (QA) beyond security, such as maintainability and performance [9].

* Corresponding authors.

E-mail addresses: francisco.ponce@pucv.cl (F. Ponce), jacopo.soldani@unipi.it (J. Soldani), g.squarciapino@studenti.unipi.it (G. Squarciapino), hernan.astudillo@unab.cl (H. Astudillo), antonio.brogi@unipi.it (A. Brogi).

<https://doi.org/10.1016/j.jlamp.2026.101178>

Received 15 June 2026; Received in revised form 17 September 2026; Accepted 18 September 2026

Available online 21 September 2026

2352-2208/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

Furthermore, multiple *instances* of security smells may affect an MSA, as the same smell can manifest across different services. This means that different decisions may be made in resolving individual instances of the same smell. For example, resolving the *centralized authorization* security smell requires applying the *use decentralized authorization* refactoring, which negatively impacts a service's testability and resource utilization [9]. Consequently, an instance of this security smell might be resolved if such QAs are not important for the affected service. In contrast, we may end up deciding to keep the *centralized authorization* smell instance affecting a service of the same or another MSA, because resource utilization is crucial for that specific service.

Additionally, certain smell instances may be more "urgent" than others, particularly if they affect services supporting core functionalities and/or if they impact critical quality attributes. Therefore, architects must not only find trade-offs for resolving individual smell instances but also decide which refactoring to apply first [10]. This decision becomes even more complex with the increasing number of services composing an MSA, their diverse quality requirements, and the varying impacts of security smells and refactorings on these attributes. Consequently, determining which security smell instances should be resolved first is inherently challenging and costly [11]. Once refactorings are identified, teams must decide which ones to implement, considering factors such as the urgency of each smell instance, its impact on quality attributes, and the effort required for its implementation [12].

However, determining which refactoring to apply first involves practical considerations beyond urgency, quality, and effort. Given the distributed nature of MSAs, where service responsibility is distributed across different teams, refactoring a particular service can impact the decisions and development timelines of other teams managing interconnected services. In such cases, certain refactorings cannot be handled within a single sprint by one team alone but instead require setting up cross-team projects that demand additional time, budget, and coordination, significantly increasing the cost and complexity of deciding which refactorings to apply first.

To address these challenges and to better support practitioners under these complex and dynamic conditions, we propose a decision support method specifically designed to address these multifaceted constraints. Our approach offers a holistic perspective that considers the urgency of each security smell instance along with the effort needed to concretely implement the refactoring as well as cross-team dependencies, ultimately enabling context-aware and sprint-aligned refactoring decisions. To this end, we provide the following main contributions:

- We introduce a Kanban-inspired visualization that organizes the refactorings needed to implement based on their associated urgency code and implementation effort. We also introduce a visualization inspired by ATAM utility trees [13], which provides insights on the relevance of each assigned service to the business, the importance of its key quality attributes, and the refactorings that a team needs to implement.
- We present a prototype tool that automates the decision support method, by providing a graphical environment where practitioners can navigate the architectural view, the Kanban-inspired view, and the ATAM-inspired view.
- We evaluate the perceived usefulness of the proposed decision support method through a preliminary empirical study following a one-group pretest-posttest design. The evaluation aims to address the following research questions: RQ_1 : *Does the proposed method facilitate the decision process when selecting which refactoring to implement?* RQ_2 : *Does the proposed method increase subjects' confidence in their selected refactorings?* After being introduced to the method, participants perceived the decision process as easier and reported greater confidence in their selected refactorings.

The rest of this article is organized as follows. Section 2 discusses related work. Section 3 provides the necessary background and introduces a motivating scenario. Section 4 describes the proposed decision support method. Section 5 presents the prototype tool. Section 6 discusses the evaluation assessment and its results. Finally, Sections 7 and 8 discuss threats to validity and draw some concluding remarks, respectively.

This paper extends our previous work [12] by instantiating the vision we outlined therein into a decision support method. More precisely, Section 4 presents our proposed decision support method in more detail. Additionally, the prototype tool presented in Section 5 and the method evaluation presented in Section 6 are brand new and first presented in this paper.

2. Related work

Refactoring MSAs to resolve their issues is crucial, as highlighted by Lenarduzzi et al. [14] and Besker et al. [10], who examine how software practitioners prioritize technical debt in modern development environments where continuous delivery pressures make balancing feature development and refactoring efforts critical. There is work to detect microservice security smell instances and reason about their resolution, e.g. Dell'Immagine et al. [15] address detection of security smells in Kubernetes-deployed applications, and Wizenty et al. [11] propose a model-driven approach to detect security smells in LEMMA models and recommend refactorings to resolve them. However, the challenge of prioritizing refactorings for detected security smell instances remains a relevant issue, as also noted by Cerny et al. [16]. While several approaches address refactoring prioritization, our proposal distinguishes itself by providing team-centric support for resolving security smell instances within team-owned microservices.

Existing approaches diverge from our method, primarily focusing on prioritizing refactorings for *code smells*, such as the ones described by Verma et al. [17] as opposed to the *MSA security smells* as identified by Ponce et al. [7]. For instance, Haendler et al. [18] rank code smells through execution trace analysis from runtime tests, while Gupta et al. [19] and Malhotra et al. [20] utilize static analysis tools to assess code smell severity in object-oriented applications. These methods, focusing on distinct smell types within object-oriented systems, lack the team-centric perspective inherent in our approach.

Other approaches prioritize code smell refactorings via source code analysis. Alshammari et al. [21] and Arcelli Fontana et al. [22] propose an intensity index to estimate code smell severity. Aung et al. [23], Lim et al. [24], and Pecorelli et al. [25] employ machine learning models for ranking. Closer to our work, Sae-Lim et al. [26,27] compute context-relevant indices from issue-tracking

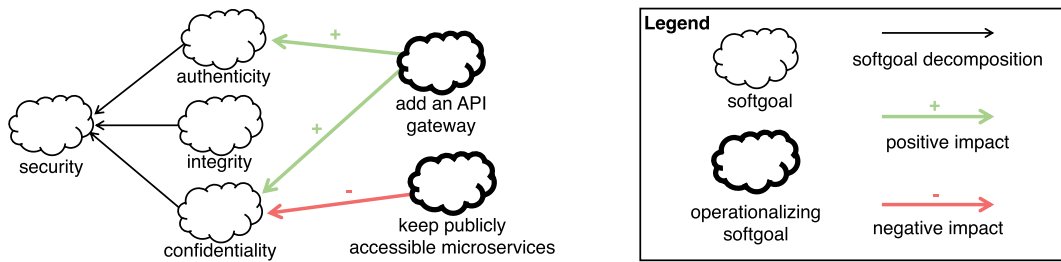


Fig. 1. Example of a softgoal interdependency graph.

systems, and Vidal et al. [28] combine component modification history, modifiability, and smell relevance to guide prioritization. These methods center on component-based application code smells, and do not provide the team-centric perspective that our decision support method offers.

Several studies incorporate a team perspective, though with different objectives than our own. For example, Aldaej et al. [29] explored how software startups make and should make technical debt decisions across different stages, and produced a rule-based decision model capturing effective practices for managing technical debt over time, which offers guidance for startup teams to make better-informed technical debt decisions. Zhong et al. [30] explore the impacts, causes, and solutions of MSA architectural smells, which may harm system maintainability by creating poor microservice boundaries. They conducted an industrial case study on a telecommunication system, analyzing six common architectural smell types using quantitative and qualitative methods, and found that architectural smells do affect negatively modularity, modifiability, analyzability, and testability, leading to increased cross-team communication and fault-prone services.

Also closer to our work but with a different focus, Bacchiega et al. [31] report an industrial case study analyzing a microservice-based project to detect architectural smells, a key indicator of architectural debt. The study shows how identifying architectural smells can help evaluate architectural debt and guide refactoring decisions. Lastly, Agostini et al. [32] address the need for decentralized development and maintenance in MSA, where microservices are distributed among DevOps teams to minimize costly cross-team interactions. They identify three types of organizational smells that signal poor decentralization in team assignments, and propose corresponding refactorings to resolve them. While these works acknowledge the importance of team context, they focus on general technical debt, architectural smells, or organizational structure. Our proposal is distinct in its objective, since we aim to provide team-centric support for prioritizing the resolution of security smell instances.

Our prior work provides the foundation for the contribution of the present article. Ponce et al. [8] proposed a trade-off analysis approach that models the impact of MSA security smells on service quality attributes using Softgoal Interdependency Graphs. However, the approach considered security smells at the general level and did not account for the contextual characteristics of individual smell instances. Ponce et al. [33] addressed this limitation by introducing instance-level granularity and combining service business relevance with quality-attribute importance into an urgency code. However, the approach focuses on urgency and does not consider the feasibility of implementing the corresponding refactorings. Subsequently, Ponce et al. [12] outlined a vision for a teamwise decision-making process, but did not instantiate or evaluate it. The present article operationalizes this vision as a concrete decision support method. It extends the previous work by treating implementation effort and cross-team dependencies as first-class decision factors, realizes the method in an open-source prototype tool, and evaluates its usefulness through an empirical study.

3. Background and motivations

Microservice security smells are widespread and diverse, often affecting multiple services with varying levels of urgency. Practitioners must consider both the business relevance of the affected service and the impact on critical quality attributes. In this section, we first introduce Softgoal Interdependency Graphs (Section 3.1). Then we recap a triage method designed to systematically assign urgency codes (Section 3.2). Lastly, we motivate our decision support method using a scenario based on a third-party MSA (Section 3.3).

3.1. Softgoal interdependency graphs

Softgoal Interdependency Graphs support the systematic analysis of how design decisions, referred to as *operationalizing softgoals*, influence quality attributes, represented as *softgoals*. By explicitly modeling both positive and negative contributions of design decisions to quality attributes, SIGs provide application architects with a visual representation of these interdependencies, supporting the identification of conflicting concerns and the subsequent analysis of design trade-offs [34].

Fig. 1 presents a Softgoal Interdependency Graph (SIG) illustrating the impact of two design decisions on *security*: the operationalizing softgoals *keep publicly accessible microservices* and *add an API gateway*. The *security* softgoal is decomposed into three security properties, namely, the softgoals *confidentiality*, *integrity*, and *authenticity*. The impacts of each design decision are represented as positive or negative contributions using green and red arrows, respectively. These contributions are further annotated with + or -, indicating that a design decision contributes positively or negatively to the achievement of a given quality attribute. Stronger contri-

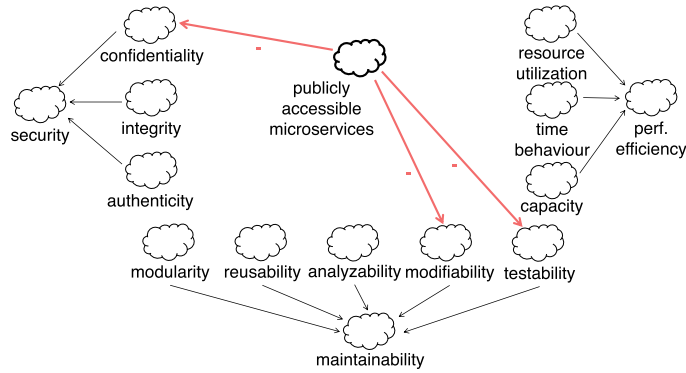


Fig. 2. SIG showing impact of publicly accessible microservices. Negative impacts are shown with red arrows labeled “-”. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Impact on QAs					
High	<i>m</i>	<i>M</i>	<i>h</i>	<i>H</i>	Service relevance
Medium	<i>L</i>	<i>m</i>	<i>M</i>	<i>h</i>	
Low	<i>l</i>	<i>L</i>	<i>m</i>	<i>M</i>	
None	$\emptyset$	<i>l</i>	<i>L</i>	<i>m</i>	
	None	Low	Medium	High	

Fig. 3. TriSS’ color-coded triage matrix: *H* (high); *h* (medium to high); *M* (medium); *m* (low to medium); *L* (low); *l* (none to low); $\emptyset$ (none).

butions can be represented using ++ or --, indicating that a design decision provides a significant contribution toward, or substantially compromises, the corresponding quality attribute.

3.2. Triage security smells (TriSS)

When multiple instances of different security smells appear in an MSA, practitioners must identify the security smell instances that require urgent attention. This involves recognizing that, within any MSA, some services are more “relevant” to the business than others, since e.g., they implement core functionalities, and that services may have varying quality-attribute requirements. For instance, services handling sensitive data may prioritize confidentiality and integrity over availability, whereas frontend services may prioritize availability.

To address this, we have proposed a triage method (TriSS) for security smell instances affecting an MSA [33]. The method combines: (1) the business relevance of the service affected by the security smell instance, and (2) the impact of the smell instance on the quality attributes required by the affected service. This combination produces an “urgency code” assigned to each security smell instance, allowing practitioners to compare the urgency of resolving different smell instances within the same MSA. The urgency code enables the classification of smell instances into “urgency classes,” helping teams prioritize their efforts, e.g., starting with those assigned the highest urgency code.

TriSS relies on practitioners to specify the relevance of services (*None*, *Low*, *Medium*, or *High*), as they possess the contextual knowledge needed to make these assessments. For example, a certain service may be assigned *High* relevance due to its role in managing central business data, whilst a monitoring service might have relevance *None* since it primarily gathers and processes monitoring data. In addition, TriSS requires practitioners to classify the importance of quality attributes (QAs) for each service using the same *None–High* scale. For example, if leaking customer information is considered important but less critical than altering operational data, then a certain service requirement on QAs might have *High* relevance to integrity and authenticity but *Medium* relevance for confidentiality.

The impact of security smells on quality attributes is drawn directly from our previous work [9], where we systematically elicited the impacts of security smells and refactorings beyond security. For instance, Fig. 2 illustrates the impacts of the *publicly accessible microservices* security smell [9] using a softgoal interdependency graph (SIG) [34], as introduced in [8]. In our previous work, we have shown that keeping (i.e., not resolving) instances of this security smell negatively impacts confidentiality, modifiability, and testability.

The SIGs for all other security smells are publicly available online at <https://ms-security.github.io>.

TriSS computes the urgency code for a security smell instance using the triage matrix shown in Fig. 3, which cross-references the affected service relevance (x-axis) with the highest priority of a negatively impacted QA (y-axis). The resulting urgency codes range from *H* (highly urgent) to $\emptyset$ (non-urgent), providing a clear, systematic way to prioritize which security smell instances to address first.

For example, consider a security smell instance that negatively impacts *confidentiality* and *analyzability*. If the importance of these quality attributes for the affected service has been estimated as *Low* and *Medium*, respectively, the overall impact on QAs for this

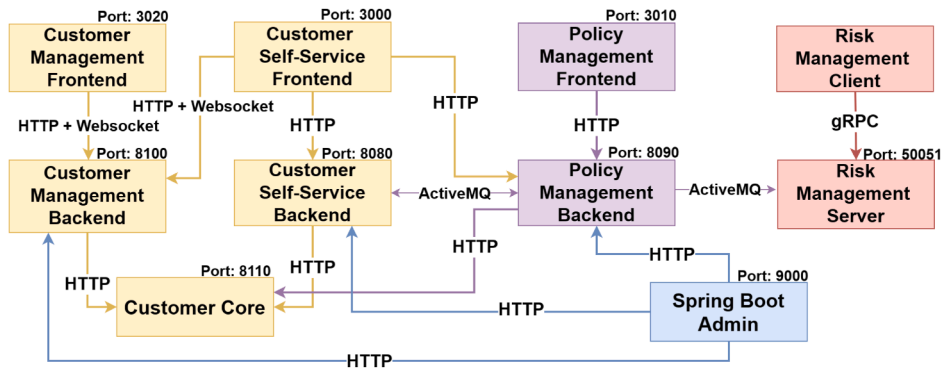


Fig. 4. Lakeside mutual MSA architecture.

Table 1
Security smell instances affecting the Lakeside Mutual MSA.

Security smell	Affected services
Publicly accessible microservices	Customer Core, Customer Management Backend, Customer Management Frontend, Customer Self-Service Backend, Customer Self-Service Frontend, Policy Management Backend, Policy Management Frontend, Risk Management Server, Spring Boot Admin
Insufficient access control	Customer Management Backend, Customer Self-Service Backend, Policy Management Backend, Spring Boot Admin
Unauthenticated traffic	Customer Management Backend, Customer Management Frontend, Customer Self-Service Backend, Customer Self-Service Frontend, Policy Management Backend, Policy Management Frontend, Spring Boot Admin
Hardcoded secrets	Customer Management Backend, Customer Self-Service Backend, Policy Management Backend, Risk Management Server
Non-secured service-to-service communications	Customer Self-Service Backend, Customer Self-Service Frontend, Customer Management Backend, Customer Management Frontend, Policy Management Backend, Policy Management Frontend, Customer Core, Spring Boot Admin

small instance is computed as *Medium* (i.e., the highest of the two). Then, if the affected service relevance is *Low*, TriSS assigns an urgency code of *m* to the considered security smell instance, indicating its low-to-medium urgency.

Since a security smell instance is resolved by applying one of its known refactorings [7], the urgency code assigned to the smell instance is inherited by the corresponding refactoring. Accordingly, throughout the remainder of this article, we use the term *refactoring urgency* as shorthand for the urgency of the security smell instance that the refactoring resolves.

3.3. Motivating scenario

To illustrate and motivate our proposal, we will use Lakeside Mutual, a third-party MSA simulating an insurance company that is publicly accessible on GitHub at <https://github.com/Microservice-API-Patterns/LakesideMutual>. Fig. 4 presents the architecture of Lakeside Mutual, highlighting the services and their distribution among four development teams: Yellow, Purple, Red, and Blue. Each team is responsible for a subset of services, e.g., the Purple Team manages the *Policy Management Frontend* and *Policy Management Backend* services.

Lakeside Mutual serves as a demonstrator MSA, frequently used in MSA-related research [11,35–37]. Its source code, available online, includes 32 instances of five known security smells [33], all summarized in Table 1. Despite its relatively simple design, Lakeside Mutual effectively showcases the complexity of prioritizing the security smell instances affecting an MSA.

Identifying 32 different security smell instances affecting the Lakeside Mutual MSA raises several important questions: How does each security smell instance impact the overall quality of the MSA? Is the *hardcoded secrets* instance affecting the *Customer Self-Service Backend* more or less severe than the one affecting *Risk Management Server*? How should instances of the *hardcoded secrets* smell be compared to those of other security smells? More generally, which security smell instances are more “urgent” to address?

Suppose that the Product Owner of the Lakeside Mutual insurance company specifies the relevance of each service and the importance of its quality attributes as in [Table 2](#). For instance, *Customer Core* is assigned medium business relevance and prioritizes security properties, since it handles customers' personal information. Indeed, with the information indicated in [Table 2](#), we can exploit the TriSS method (see [Section 3.2](#)) to assign each security smell instance an urgency code.

Consider, for instance, the *publicly accessible microservice* smell instance affecting the *Customer Self-Service Backend* service. Keeping such *publicly accessible microservice* smell instance negatively impacts the confidentiality, modifiability, and testability of *Customer Self-Service Backend*, as displayed by the SIG in Fig. 2. At the same time, *Customer Self-Service Backend* sets the importance of confidentiality,

Table 2

Service relevance and QAs importance specified by the Lakeside Mutual Product Owner.

Service	Relevance	Importance of QAs
Customer Core	Medium	High: confidentiality, integrity. Medium: time behavior. Low: modifiability.
Customer Management Backend	Medium	High: integrity. Medium: authenticity, time behavior. Low: modifiability.
Customer Management Frontend	Low	High: integrity. Medium: authenticity, time behavior. Low: modifiability.
Customer Self-Service Backend	High	High: confidentiality, time behavior. Medium: integrity, authenticity. Low: modifiability, resource utilization.
Customer Self-Service Frontend	Medium	High: confidentiality, time behavior. Medium: integrity, authenticity. Low: modifiability, resource utilization.
Policy Management Backend	High	High: analyzability Medium: integrity, authenticity, confidentiality.
Policy Management Frontend	Medium	High: integrity. Medium: authenticity, confidentiality. Low: analyzability.
Risk Management Client	Low	High: time behavior, resource utilization. Medium: integrity. Low: reusability.
Risk Management Server	Low	High: time behavior, resource utilization. Medium: integrity. Low: reusability.
Spring Boot Admin	None	High: resource utilization. Medium: authenticity, time behavior. Low: reusability.

Table 3

TriSS resulting urgency codes for each security smell instance affecting the Lakeside Mutual MSA.

Urgency	Smell instances
<i>H</i>	(Publicly accessible microservices, Customer Self-Service Backend), (Unauthenticated traffic, Customer Self-Service Backend), (Unauthenticated traffic, Policy Management Backend), (Hardcoded secrets, Customer Self-Service Backend), (Hardcoded secrets, Policy Management Backend), (Non-secured service-to-service communications, Customer Self-Service Backend), (Non-secured service-to-service communications, Policy Management Backend), (Insufficient access control, Customer Self-Service Backend)
<i>h</i>	(Publicly accessible microservices, Customer Core), (Publicly accessible microservices, Customer Self-Service Frontend), (Publicly accessible microservices, Policy Management Backend), (Unauthenticated traffic, Customer Self-Service Frontend), (Hardcoded secrets, Customer Management Backend), (Non-secured service-to-service communications, Customer Management Backend), (Non-secured service-to-service communications, Customer Self-Service Frontend), (Non-secured service-to-service communications, Policy Management Frontend), (Non-secured service-to-service communications, Customer Core), (Insufficient access control, Policy Management Backend)
<i>M</i>	(Publicly accessible microservices, Policy Management Frontend), (Unauthenticated traffic, Policy Management Frontend), (Unauthenticated traffic, Customer Management Backend), (Non-secured service-to-service communications, Customer Management Frontend),
<i>m</i>	(Publicly accessible microservices, Customer Management Backend), (Unauthenticated traffic, Customer Management Frontend), (Hardcoded secrets, Risk Management Server)
<i>L</i>	(Publicly accessible microservices, Customer Management Frontend), (Unauthenticated traffic, Spring Boot Admin), (Non-secured service-to-service communications, Spring Boot Admin), (Insufficient access control, Customer Management Backend)
<i>l</i>	(Publicly accessible microservices, Risk Management Server),
$\emptyset$	(Publicly accessible microservices, Spring Boot Admin), (Insufficient access control, Spring Boot Admin)

modifiability, and testability to *High*, *Low*, and *None*, respectively (Table 2), which means that the impact on QAs of the considered security smell instance is *High*, since the confidentiality and modifiability have *High* importance and they are both negatively impacted by such smell instance. This, combined with the *High* relevance of *Customer Self-Service Backend*, results in the *publicly accessible microservice* smell instance affecting the *Customer Self-Service Backend* service getting assigned with the urgency code *H*, i.e., it is highly urgent.

By systematically applying the TriSS method to the 32 security smell instances, as we already illustrated above, we obtain the urgency codes displayed in Table 3. The table also partitions the security smell instances into urgency classes, which provide practitioners with information about which security smell instances are more urgent than others.

Security smell instances can be resolved through known refactorings [7]. However, determining which refactoring to apply first involves practical considerations beyond urgency. During this decision process, a broader set of constraints should be considered; factors such as the effort required to implement the associated refactoring, the need for coordination with other teams responsible for dependent or interacting services, and the limitations imposed by available time and resources must be taken into account when

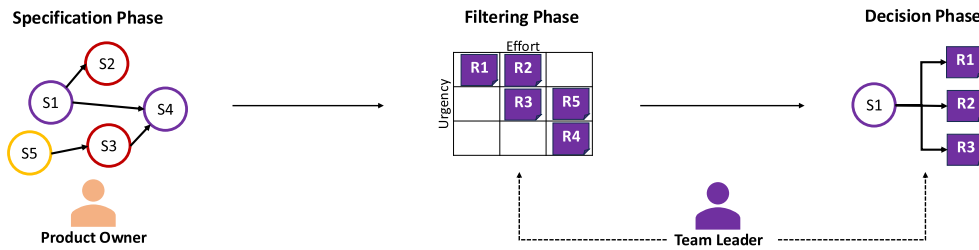


Fig. 5. Decision support method overview.

making the decision. These challenges are further amplified in modern Agile development contexts, where short time frames between sprints demand quick, well-informed decisions [38]. Practitioners must take rapid yet informed decisions, often needing to identify trade-offs among the urgency of addressing a security smell instance, sprint commitments, ongoing feature development, and strict delivery deadlines.

Therefore, while urgency remains a key factor, real-world decisions often require navigating within and across the urgency classes provided by TriSS. For example, two security smell instances might fall into the same high-urgency class, yet one may demand substantial implementation effort and cross-team coordination, while the other can be addressed independently by a single team within the constraints of the current sprint. Conversely, a smell instance classified with lower urgency might be selected if it represents a low-effort, high-impact opportunity that aligns well with sprint capacity and avoids disruption. In this context, the following section introduces a decision support method for addressing the complex and dynamic conditions involved in refactoring microservice security smells. The proposed approach provides a holistic perspective by considering the urgency of each smell instance, its refactoring implementation effort, and cross-team dependencies, thereby supporting context-aware and sprint-aligned refactoring decisions.

4. Decision support

As discussed in Section 3.3, determining which refactoring to apply first involves practical considerations beyond urgency. The decision must also account for factors such as the effort required to apply the refactoring, the coordination necessary with teams responsible for interacting services, and the limitations imposed by available time and resources. Furthermore, modern development contexts demand quick and well-informed decisions [38].

To address the challenges described above, we propose a method that supports practitioners in making informed decisions and improving the scheduling of refactorings aimed at mitigating the effects of security smell instances. Fig. 5 provides an overview of our approach, which enables practitioners to explicitly map each service in the MSA to its responsible team. Our decision support method offers a holistic view that integrates business relevance, the importance of key quality attributes, and the required refactorings. As illustrated in Fig. 5, our method is organized into three phases:

- In the *Specification Phase*, the Product Owner defines the services composing the MSA, specifies their interactions, and assigns the business relevance and key quality attributes. During this phase, the Product Owner can also register the security smell instances currently affecting the services composing the MSA. Combined with the previously entered information, this input enables the automatic calculation of urgency codes, for example, through the TriSS method [33]. The Specification Phase establishes the foundation for systematically reasoning about smell urgency, refactoring effort, and team responsibilities in subsequent phases.
- The *Filtering Phase* provides a structured visualization that organizes refactorings by relative urgency and the implementation effort estimated by the responsible team. As illustrated in Fig. 5, this Kanban-inspired view positions each refactoring within a matrix that captures the trade-off between these two dimensions. If business priorities shift or team effort estimates are updated, the placement of refactorings within the matrix automatically adapts, ensuring a flexible and systematic way of reasoning that remains robust to contextual changes. This phase enables teams to narrow down the set of potential refactorings by emphasizing those that best align with the current resource and time constraints. By explicitly mapping urgency against effort, the Filtering Phase helps teams to focus on the most pressing and achievable refactorings, while also documenting those that may be postponed.
- In the *Decision Phase*, we provide teams with an ATAM-inspired view, building on the concept of ATAM utility trees [13], which offers a structured approach to capturing and prioritizing quality attributes. Our adaptation extends this idea to present teams with a holistic view, which offers deeper insights into the business relevance of each service, the importance of its key quality attributes, and the set of refactorings that meet the constraints defined during the Filtering Phase, along with their associated effort. The ultimate goal of this phase is to guide teams toward the “most convenient” decision, one that balances urgency, effort, and business priorities, while maintaining transparency in the trade-offs considered. To better describe each of these two phases and views, some examples based on specific scenarios are presented below.

Example. To illustrate our approach, we revisit our motivating scenario described in Section 3.3, focusing on the Purple Team services and the refactorings needed by them. Initially, the Product Owner of the Lakeside Mutual MSA provides the necessary data during the Specification Phase. During the Filtering Phase, the TriSS-generated urgency codes, as exemplified in Table 3, are used to populate the Kanban-inspired view, organizing refactorings by their associated urgency and implementation effort. Table 4 illustrates the concept

Table 4

Conceptual Kanban-inspired view illustrating refactoring urgency versus implementation effort.

Urgency level	High effort	Medium effort	Low effort
High	Use OpenID Connect to Policy Management Backend	Add an API Gateway to Policy Management Backend Affected teams: Yellow and Blue	
	Use mutual TLS to Policy Management Backend Affected teams: Yellow and Blue		
Medium to High		Encrypt secrets at rest to Policy Management Backend	
		Use OAuth 2.0 to Policy Management Backend Affected teams: Yellow and Blue	
		Use mutual TLS to Policy Management Frontend	
Medium			Use OpenID Connect to Policy Management Frontend Add an API Gateway to Policy Management Frontend
Low to Medium			
Low			
None to Low			
None			

of our Kanban-inspired view. Urgency levels are displayed along the rows, ranging from High to None, while effort levels are shown in the columns, divided into High, Medium, and Low effort. This structured representation supports teams in visualizing the relative urgency and effort of the refactorings needed, ultimately enabling practitioners to take more informed decisions.

Suppose that for the current Filtering Phase, the Purple Team can allocate resources for refactorings requiring up to Medium effort. They are also able to manage interactions with other teams and should focus on refactorings with Medium-to-High or High urgency. Under these constraints, even though the most urgent refactorings involve adding *OpenID Connect* and *Mutual TLS* to the *Policy Management Backend* service (as shown in the top-right corner of Table 4), the Purple Team might defer these actions to future sprints when additional resources and time are available. For the Decision Phase, the ATAM-inspired view illustrated in Fig. 6 is generated, containing the four refactorings that meet the Purple Team Filtering Phase constraints. The left section displays the services managed by the team, namely the *Policy Management Frontend* and *Policy Management Backend* services. The central section highlights the key quality attributes and their importance, as defined by the Product Owner in Table 2. The right section lists the refactorings, their urgency, and estimated effort, linking each to the impacted quality attributes.

This visualization provides the Purple Team with a holistic perspective of refactoring needs, the urgency of the associated security smells, and whether collaboration with other teams is required. For instance, refactorings targeting the *Policy Management Backend* service impact services owned by the Yellow and Blue Teams. Consequently, interactions with both the Yellow and Blue Teams would be required to address certain smells affecting the *Policy Management Backend* service. In contrast, the *Use Mutual TLS* refactoring for the *Policy Management Frontend* does not impact other teams, eliminating the need for cross-team collaboration. Given that these four refactorings satisfy the constraints imposed for the next sprint, the “most convenient” decision may lie between *Use Mutual TLS* for the *Policy Management Frontend* service and *Add an API Gateway* for the *Policy Management Backend* service.

Now suppose that during a new Filtering Phase, the Purple Team still has resources for Medium-effort refactorings but does not have the budget to set up cross-team projects, thus not being able to manage interactions with other teams. The view illustrated in

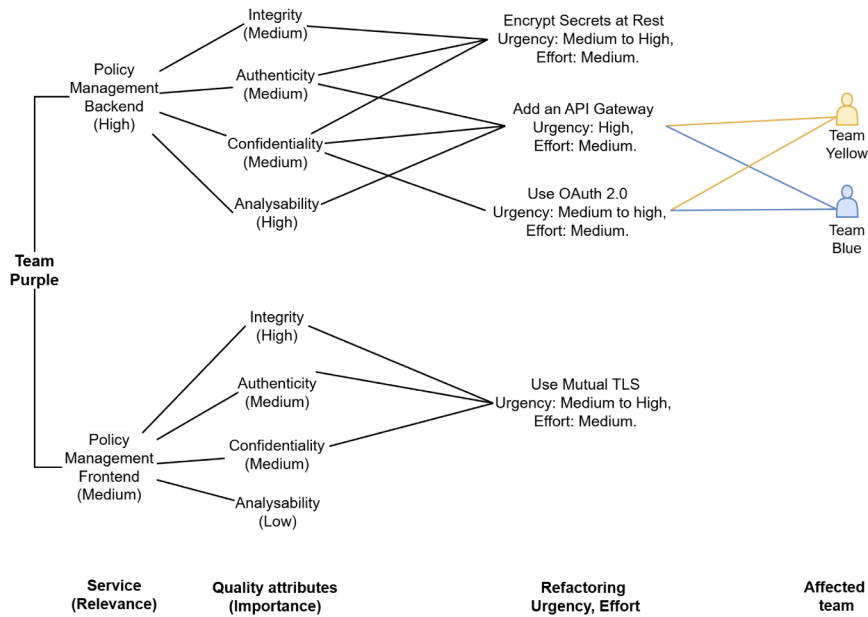


Fig. 6. ATAM-inspired view generated for Medium or lower effort, interactions allowed, and Medium to high or High urgency.

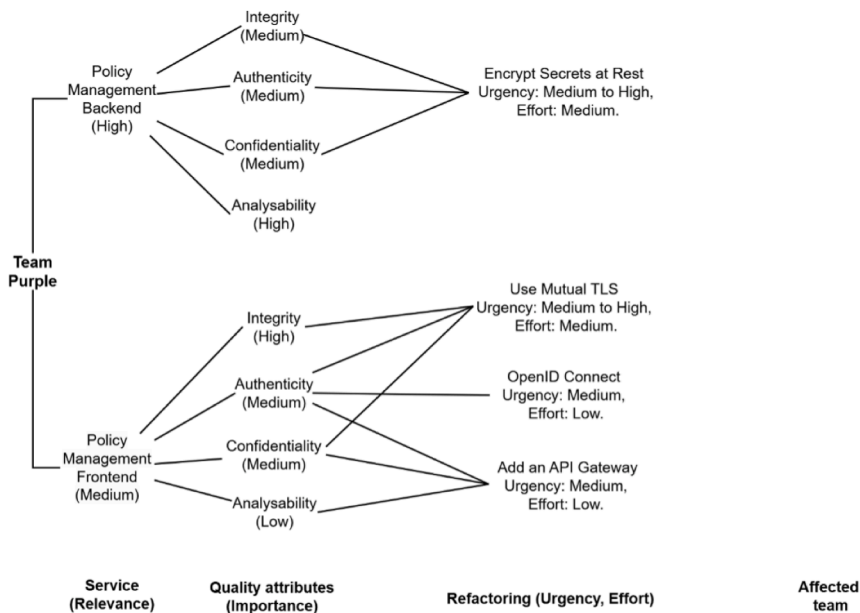


Fig. 7. ATAM-inspired view generated for Medium or lower effort, disallowed interactions, and any urgency.

Fig. 7 is then generated to satisfy these constraints. Here, the decision taken by the Purple Team should shift toward refactorings that avoid cross-team coordination. Options include *OpenID Connect* for the *Policy Management Frontend*, which requires low effort, or more urgent refactorings such as *Encrypt Secrets at Rest* for the *Policy Management Backend*.

These contrasting scenarios illustrate how the proposed method supports practitioners in taking context-aware decisions. While the examples are based on a specific set of sprint constraints, different contexts may lead to different decisions. Ultimately, our approach is designed to guide practitioners toward the “most convenient” decision by helping them balance trade-offs among refactoring effort, cross-team interactions, sprint commitments, ongoing feature development, and strict delivery deadlines.

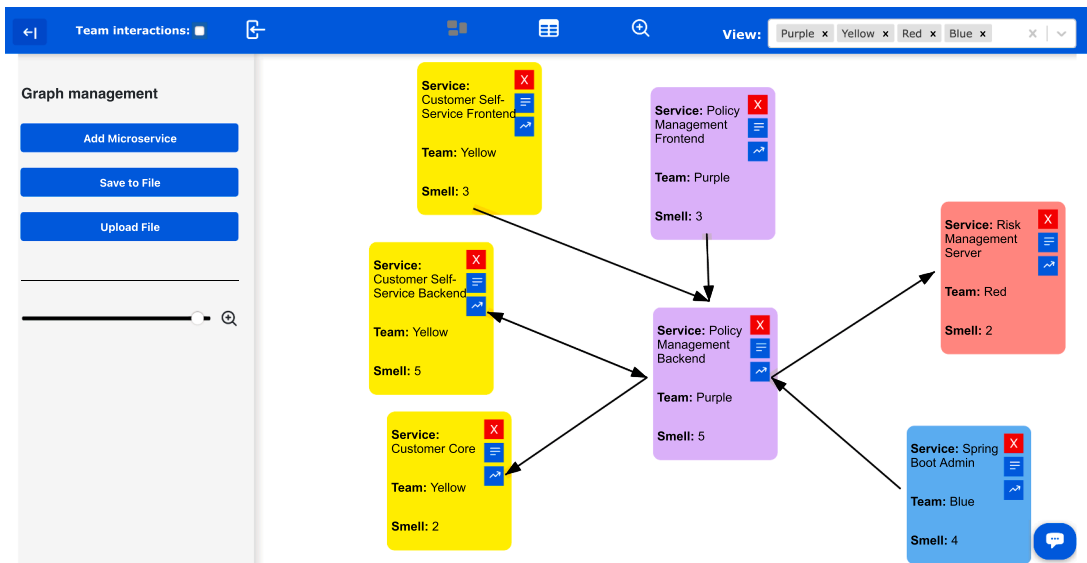


Fig. 8. Architectural view used during the specification phase, captured from the prototype tool, showing a services subset of Lakeside Mutual MSA.

5. Prototype

To assess the practical applicability of our proposal, we developed a prototype tool that automates the decision support method and the three phases described in Section 4. The prototype is open-source and publicly available on GitHub at <https://ms-security.github.io/SmellAnalyzer>. The tool provides a graphical environment where practitioners can navigate an architectural view, the Kanban-inspired view, and the ATAM-inspired view generated for each team.

Fig. 8 illustrates the architectural view that is used by the Product Owner during the Specification Phase. This interface enables Product Owners to define the individual services composing the MSA, either by entering them manually, importing data from a JSON file, or exporting the defined architecture for reuse. When creating a service, the Product Owner can specify details such as service name, the team responsible for its development and maintenance, and its relevance for the business. The interface also supports defining service interactions to capture dependencies across the MSA, as well as documenting current security smell instances affecting each service. This architectural view also allows the definition of the importance of several quality attributes for each service, providing a clearer view of its role and critical concerns within the MSA. The capture shown in Fig. 8 was generated using the information in Fig. 4, Tables 1 and 2, which allows us to configure the motivating scenario within the prototype tool.

Fig. 9 illustrates the Kanban-inspired view used by teams during the Filtering Phase. In this interface, teams can specify the estimated effort required to implement each refactoring associated with the identified security smell instances affecting the MSA. Once the effort values are provided, the tool automatically populates the Kanban-inspired view, positioning refactorings based on both their inherited urgency and estimated implementation effort. Urgency is systematically calculated using the TriSS method, which relies on data entered by the Product Owner during the Specification Phase, including service relevance, quality-attribute importance, and security smell instances. Teams can also apply filters, for example, to display only refactorings of Medium effort. These filters directly influence the view, enabling teams to focus on refactorings that are most relevant to their available time and resources. The capture in Fig. 9 was automatically generated from the input data defined for our motivating scenario described in Section 3.3, with urgency levels for the Purple Team computed using the TriSS method, as detailed in Table 3.

Fig. 10 presents the ATAM-inspired view that is used by teams during the Decision Phase. This interface, as introduced for Figs. 6 and 7, is automatically constructed from the data provided in the architectural and Kanban-inspired views. This holistic visualization highlights the refactorings that align with the constraints and preferences defined by teams, including effort thresholds, urgency levels, service relevance, and inter-service dependencies. By integrating these factors, the ATAM-inspired view provides teams with actionable decision guidance tailored to their specific sprint conditions.

5.1. Implementation

The prototype is implemented as a web application, making the tool accessible from any device without installation and reducing infrastructural cost and complexity. This is particularly relevant in our setting, where teams need to collaborate on and share updated representations of the logical architecture of the same MSA. The tool is built using React, with Vite as the development toolchain and Firebase as a Backend-as-a-Service platform for authentication and database functionality. This architecture avoids the need to

Urgency \ Effort	High effort	Medium effort	Low effort
High	Use of Mutual Transport Layer Security OpenID Connect	Add an API Gateway Use OAuth 2.0	
Medium to High		Use of Mutual Transport Layer Security	
Medium			OpenID Connect Add an API Gateway
Low to Medium			
Low			
None to Low			
None			

Fig. 9. Snapshot of the prototype tool interface, showing the Kanban-inspired view for the Purple Team services.

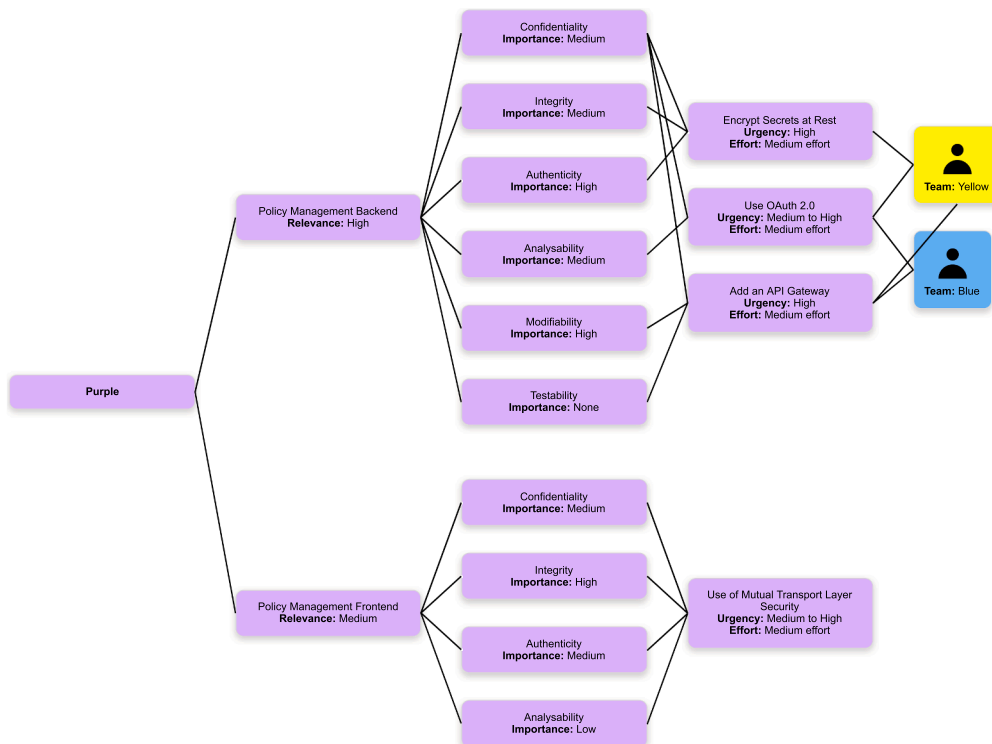


Fig. 10. Snapshot of the prototype tool interface, showing an ATAM-inspired view for the Purple Team.

configure and maintain dedicated servers while providing persistence and authentication mechanisms. The prototype is structured as a Single Page Application using React Router, which manages navigation between the login page, account management page, and the three main views described previously. The interface is organized into reusable components, including those responsible for rendering microservices and their interactions, editing microservice information, and providing the chatbot described in [Section 5.1.2](#).

5.1.1. Configuration files and data entities

The catalogs used by the method are externalized into JSON configuration files, allowing the tool to be adapted without modifying its source code. The information contained in those files is used to derive the cross-team dependencies represented in the Kanban-inspired and ATAM-inspired views. Externalizing these catalogs makes the tool more extensible, adaptable, and maintainable, since new smells, refactorings, and quality attributes can be added without modifying the code. It also allows the configurations to be inspected and edited independently and potentially reused by other tools.

5.1.2. Chatbot

The prototype integrates an experimental chatbot, accessible through the chat icon in the bottom-right corner of each interface, which provides a conversational entry point to the tool's functionalities. Practitioners can therefore access these functionalities without necessarily navigating through the main views, potentially reducing the learning curve associated with the interface.

We implemented the chatbot as a finite state machine rather than using machine learning models. Although machine learning could support more natural and free-form conversations, it would require substantial training data, more sophisticated management of conversational context, and could introduce less predictable behaviour. These characteristics were considered unsuitable for a prototype operating within a restricted usage domain. In our implementation, each interaction is associated with a well-defined conversational state, and transitions occur in response to supported inputs identified through predefined keywords. This design makes conversational paths explicit, which improves the predictability and verifiability of the chatbot, allows individual states to be tested in isolation, avoids the computational overhead associated with complex language processing, and enables new conversational flows to be added by defining additional states and transitions without modifying existing ones.

Because relying exclusively on exact keywords would impose a rigid input syntax, we complement the finite state machine with a natural language processing library that normalizes and classifies user input through semantic parsing and keyword extraction. This allows natural-language messages to be mapped to valid state transitions without requiring full linguistic understanding. Incoming messages are first normalized by converting them to lowercase and passing them to the NLP library. The conversational state is represented by a function. A null state indicates that the chatbot is awaiting a new request, whereas a non-null state delegates the input to the handler associated with the current conversational flow, thereby supporting multi-turn interactions. When the active handler does not consume the input, the message is matched against the keyword patterns associated with the supported intents. For example, a message containing *do*, *suggest*, or *help* is mapped to the *suggest* intent. When a team name is also recognized, the intent is parameterized with that team; otherwise, it remains generic. The parsing stage consequently produces an intent object containing the information required to execute the corresponding command. Ambiguous, incomplete, or out-of-context inputs are handled through an explicit fallback that returns a neutral message asking the practitioner to reformulate the request. This mechanism keeps conversational control within the finite state machine and prevents interaction deadlocks.

6. Evaluation

To evaluate the perceived usefulness of the proposed decision support method, we conducted a preliminary empirical study following a one-group pretest-posttest design. A replication package containing the information about the study can be found here <https://zenodo.org/records/17295441>. The pretest-posttest study consisted of asking subjects to complete two tasks in which they needed to decide what refactoring to implement under certain constraints. The ultimate goal was to extract answers to the following research questions:

- RQ_1 : Does the proposed method facilitate the decision process when selecting which refactoring to implement?
- RQ_2 : Does the proposed method increase subjects' confidence in their selected refactorings?

6.1. Pretest-posttest study design

Building on the motivating scenario described in Section 3.3, subjects were asked to complete two tasks, each requiring them to assume the role of a Purple Team leader and to decide which refactoring to implement in the upcoming project sprint. Each task was carried out twice: (a) *without* the decision support method, and (b) *with* the method. The study unfolded in four structured steps:

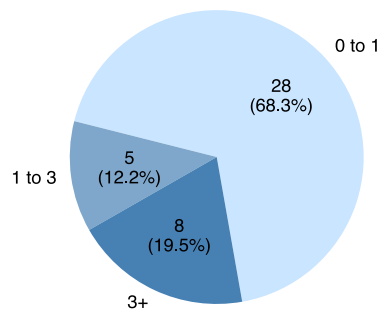
1. **Introduction:** Subjects attended a seminar on microservice security smells, their impact on security properties, and other quality attributes. They were also introduced to the Lakeside Mutual MSA and the information described in Section 3.3.
2. **Initial decision:** The subjects executed tasks 1.1 and 1.2 described in Table 5 *without* the decision support method. In doing so, they were asked to consider the urgency of each security smell instance, the relevance of the affected service, and the importance of quality attributes. After completing the tasks, subjects filled out a questionnaire that measured both their confidence in the chosen refactoring and their perceived ease of the decision process, using a 1–5 Likert scale (1 = Very difficult, 2 = Difficult, 3 = Neither easy nor difficult, 4 = Easy, 5 = Very easy).
3. **Method introduction:** Subjects were introduced to the proposed decision support method described in Section 4.
4. **Second decision:** The subjects executed tasks 2.1 and 2.2 described in Table 5 *with* our decision support method. After completing the tasks, subjects completed a second questionnaire to assess their confidence and perceived ease in the decision processes. At the end of this stage, subjects were also asked whether and how their confidence in the previous decision had changed.

The questionnaires allowed us to get answers for the research questions RQ_1 and RQ_2 , previously introduced:

Table 5

Tasks defined for the pretest-posttest study.

Task	Description
1.1	Scenario 1: For the upcoming sprint, you can allocate a medium level of effort and cannot commit to interactions with other teams in the organization. Based on these constraints, which refactoring would you implement during such a sprint? Setup: Without method, effort medium or lower, interactions not allowed, any urgency.
1.2	Scenario 2: For the upcoming sprint, you can again allocate a medium level of effort, but this time, focus only on those refactorings with a Medium to High urgency or greater. Additionally, now your team is open to cooperating with other teams in the organization. Based on these constraints, which refactoring would you implement during such a sprint? Setup: Without method, effort medium or lower, interactions allowed, urgency medium to high or high.
2.1	You are now back in Scenario 1, but this time you will have a method to support you during the decision process. For the upcoming sprint, you can allocate a medium level of effort and cannot commit to interactions with other teams in the organization. Based on these constraints, which refactoring would you implement during such a sprint? Setup: With method, effort medium or lower, interactions not allowed, any urgency.
2.2	You are now back in Scenario 2, but this time you will have a method to support you during the decision process. For the upcoming sprint, you can again allocate a medium level of effort, but this time, focus only on those refactorings with a Medium to High urgency or greater. Additionally, now your team is open to cooperating with other teams in the organization. Based on these constraints, which refactoring would you implement during such a sprint? Setup: With method, effort medium or lower, interactions allowed, urgency medium to high or high.

**Fig. 11.** Subjects' years of experience in software design and development.

- RQ_1 is addressed by comparing the perceived ease of the decision processes *with vs. without* the decision support method.
- RQ_2 is addressed by assessing subjects' confidence concerning the chosen refactoring *with vs. without* the method.

6.2. Pretest-posttest study execution

The study was conducted across two sessions, held in November and December 2024. A total of 41 subjects with varying levels of experience in software design and development took part (Fig. 11). In particular, the majority of subjects, 28 out of 41 (68.3%), reported having between 0 and 1 year of experience, indicating a predominantly novice group. In contrast, 8 subjects (19.5%) reported more than 3 years of experience, whereas the remaining 5 subjects (12.2%) had 1–3 years of experience.

The subjects were asked to complete the four activities outlined in Table 5, deciding which refactoring to implement under specific constraints, considering the urgency of each security smell instance, the affected service relevance, and the quality attributes data (Table 2). During Tasks 1.1 and 1.2, subjects were provided with all the information outlined in our motivating scenario. This included the architecture of the Lakeside Mutual MSA (as shown in Fig. 4), details about the business relevance of each service and the assigned importance of its quality attributes (refer to Table 2), as well as a list of identified security smell instances with their urgency codes calculated using the TriSS method (see Table 3). Additionally, subjects were provided with the list of corresponding refactorings, the estimated effort required to implement each one, and the SIGs available online in <https://ms-security.github.io>.

During Tasks 2.1 and 2.2, the subjects were given the same set of information as in the earlier tasks, including the MSA architecture, service relevance, quality attribute importance, the smell instances with associated urgency codes, and the possible refactorings with estimated effort. In addition, the subjects were given the option to visualize the Kanban-inspired and ATAM-inspired views of our decision support method, instantiated with data related to the Purple Team in our motivating scenario.

6.3. Decisions with and without the method

Fig. 12a and b illustrate the distribution of subjects' responses to Tasks 1.1 and 1.2, that is, both tasks that were completed *without* using the proposed decision support method. Each bar represents a specific refactoring decision, and bars are segmented by

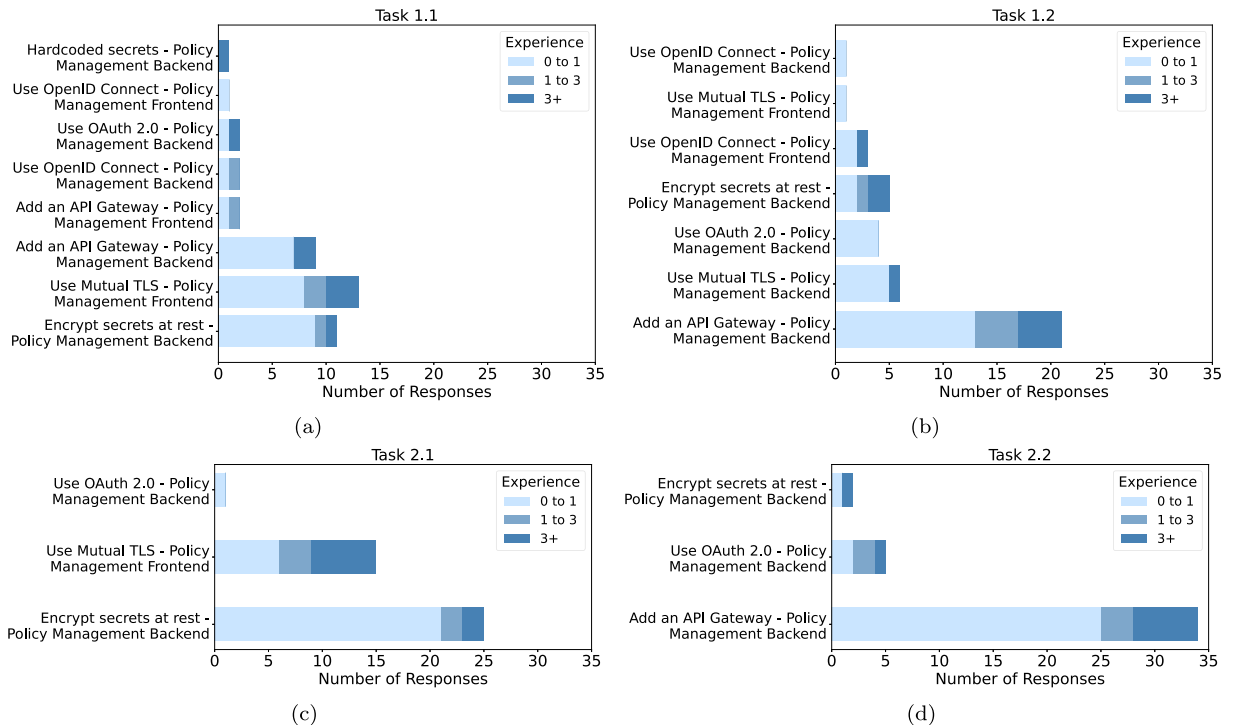


Fig. 12. Subjects' answers for tasks (a) 1.1, (b) 1.2, (c) 2.1, and (d) 2.2, showing software design and development experience.

experience levels: 0 to 1 year (light blue), 1 to 3 years (blue), and 3+ years (dark blue). As shown in Fig. 12a, responses to Task 1.1 spanned a wide range of options, suggesting that subjects explored multiple refactorings without reaching a clear consensus. This dispersion may reflect uncertainty or a lack of structured guidance when evaluating alternatives. Nevertheless, a few decisions emerged more frequently across all experience levels: *Use Mutual TLS for Policy Management Frontend*, *Encrypt secrets at rest for Policy Management Backend*, and *Add an API Gateway for Policy Management Backend*. During Task 1.2, shown in Fig. 12b, the responses are again distributed across several options, but were generally more concentrated, with *Add an API Gateway to Policy Management Backend* receiving the highest number of total responses and substantial contributions across all experience levels. Other notable decisions included *Use Mutual TLS to Policy Management Backend* and *Encrypt secrets at rest to Policy Management Backend*. The broad distribution of answers in both tasks suggests that, in the absence of decision support, subjects often struggled to decide among competing refactorings. This lack of convergence underscores the challenge of decision-making, particularly when practitioners are unfamiliar with trade-offs among effort, urgency, and quality attributes.

Fig. 12c and d, illustrate the distribution of subjects' answers to Task 2.1 and Task 2.2, where subjects used our proposed decision support method. Again, each bar represents a selected refactoring option, segmented by subjects' experience in software design and development. As shown in Fig. 12c, decisions in Task 2.1 demonstrate a marked convergence compared to Task 1.1. Among less experienced subjects, 21 of the 28 (75%) decided to implement the *Encrypt secrets at rest for Policy Management Backend* refactoring. Similarly, among the more experienced subjects, 6 of 8 (also 75%) decided to implement the *Use Mutual TLS for Policy Management Frontend* refactoring, reflecting a strong consensus even within this smaller group. Notably, these two groups converged internally but diverged in focus. Novices tended to select a refactoring with Medium-to-High urgency and Medium effort, targeting quality attributes of Medium importance within a service defined as highly relevant to the business. By contrast, the experienced group focused on a refactoring of equivalent urgency and effort but affecting a quality attribute classified as highly important within a service of Medium business relevance. This contrast suggests that less-experienced practitioners placed greater weight on business relevance, whereas more experienced practitioners emphasized the importance of quality attributes when making decisions.

For Task 2.2, as shown in Fig. 12d, the convergence on a decision was even more pronounced. Regardless of experience level, 34 of the 41 subjects (83%) decided to implement the *Add an API Gateway to Policy Management Backend* refactoring. In this task, both less experienced and more experienced groups converged on the same decision, that is, a refactoring with High urgency and Medium effort, affecting quality attributes of Medium and High importance, within a service defined as highly relevant to the business. However, across both Tasks 2.1 and 2.2, the number of distinct choices selected was substantially lower than in Tasks 1.1 and 1.2. After being introduced to the method, subjects' decisions showed greater convergence, and subjects reported greater ease and confidence, as we discuss hereafter.

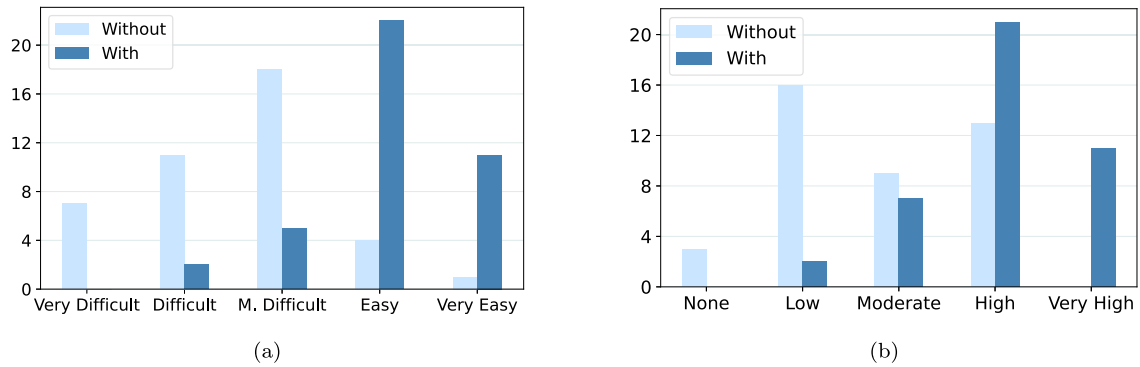


Fig. 13. Subjects' answers on (a) ease of the decision process and (b) confidence in the chosen refactoring, both *with* vs. *without* the decision support method.

6.4. Answers to research questions

Fig. 13 summarizes the findings regarding the perceived ease of the decision process and confidence in the selected refactoring. The histogram in Fig. 13a addresses RQ_1 : *Does the proposed method facilitate the decision process when selecting which refactoring to implement?* Before being introduced to our decision support method, most subjects described the tasks as moderately difficult or difficult. After being introduced to the method, the same tasks were predominantly described as easy or very easy.

Fig. 13b addresses RQ_2 : *Does the proposed method increase subjects' confidence in their selected refactorings?* Subjects generally reported greater confidence in the refactorings selected after being introduced to our decision support method. Most subjects reported high or very high confidence in the selected refactoring with the method, whereas confidence in the selected refactoring without the method was generally lower.

Taken together, these observations suggest that, after being introduced to our decision support method, subjects perceived the process of balancing the urgency of security smell instances, the affected quality attributes, and refactoring effort as easier. Subjects also reported greater confidence in the refactorings they selected, indicating that the decision support method was associated with greater assurance in their choices when considering the urgency of the security smell, the affected quality attributes, and the associated implementation effort.

7. Threats to validity

Based on the taxonomy developed by Wohlin et al. [39], four potential threats to validity may apply to our study, namely threats to external, construct, internal, and conclusion validity. We considered these threats during the design, execution, and analysis of our evaluation, and we adopted mitigation strategies to reduce their impact wherever this was possible. Our study, however, follows a one-group pretest-posttest design, in which the same subjects performed the same two scenarios first without and then with our decision support method. Some of the threats discussed below are inherent to this design and could not be mitigated by the measures we adopted. We therefore report them openly and qualify our claims accordingly, presenting our findings as preliminary evidence of the perceived usefulness of our method rather than as a demonstration of its effect on decision outcomes.

External Validity. External validity concerns the extent to which study results can be generalized to broader contexts [39]. Our evaluation relies on a single object of study, the Lakeside Mutual MSA, and a single focal team. We mitigate this threat by using a third-party application that we did not develop, is publicly available, and has been widely used in MSA-related research [11,35–37]. Moreover, its source code contains 32 instances of five distinct security smells, providing a decision problem of non-trivial size. A second threat concerns the subject population. We mitigate this threat by recruiting subjects with different levels of experience and reporting the results by experience band. This disaggregation allows us to examine whether experience influences the decisions made, as illustrated by the different refactorings selected by the more and less experienced subgroups in Task 2.1. Nevertheless, the limited and non-industrial nature of the subject population constrains the generalizability of our findings. Finally, the realism of the tasks represents a further threat. Although the scenarios are based on a realistic MSA and involve non-trivial architectural decisions, they cannot fully reproduce the complexity and contextual factors of industrial settings. We therefore acknowledge that generalization to real-world practice remains limited. As future work, we plan to evaluate our method with experienced practitioners using real project data and to investigate its applicability to larger MSAs, thereby providing broader evidence of its external validity.

Construct and Internal Validity. Wohlin et al. [39] characterize internal validity as concerning the methods employed to conduct a study and analyze its data, including potential sources of bias. They define construct validity in terms of the extent to which the constructs under study are adequately operationalized and measured. The primary threat to the internal validity of our study arises from its pretest-posttest design. All subjects first performed the tasks without our method and subsequently repeated them with the method, having been introduced to it between the two rounds. Consequently, by the second round, subjects were already familiar with the architecture of Lakeside Mutual, the service relevance and quality attribute importance data, the urgency codes, and the

catalog of candidate refactorings and their estimated effort. The tasks may therefore have been perceived as easier, and subjects may have approached them with greater confidence, as a result of repetition and increased familiarity with the problem rather than the use of our method. We partially mitigate this threat by providing subjects with the same information in both rounds, ensuring that the primary difference between the two conditions is the availability of our Kanban-inspired and ATAM-inspired views. We also conducted both sessions using the same script, slides, task descriptions, and instructor, thereby minimizing potential differences in how the tasks and study materials were presented.

We further observe that the change in reported ease and confidence between Tasks 1.1 and 1.2, both performed without our method, provides an indication of the extent to which repetition alone may account for the observed changes in our setting. These measures reduce, but do not eliminate, the threat. Since the treatment order was not counterbalanced and no control group repeated the tasks without being exposed to the method, the effects of our method cannot be statistically separated from those of learning and maturation. We therefore do not attribute the observed changes directly to the method. Throughout [Section 6](#), we instead report that subjects described the tasks as easier and expressed greater confidence *after* being introduced to our method.

Regarding construct validity, our study examines the perceived ease of the decision process and subjects' confidence in their decisions. Ease and confidence were measured through self-report and do not fully capture the underlying constructs. Perceived ease does not necessarily reflect the effort expended, while confidence may accompany either a good or a poor decision. A further threat concerns potential response bias. Because the method was introduced by its authors and the purpose of the comparison was apparent from the task structure, subjects may have inferred the expected outcome or over-reported ease. We mitigate these threats by collecting responses anonymously, informing subjects that their answers would not be assessed, and framing the questionnaire around the perceived difficulty of the tasks rather than the perceived value of the method.

Conclusion Validity. Conclusion validity concerns the extent to which the conclusions drawn from a study are reasonably supported by the available data [39]. Our analysis of the collected responses is descriptive. We did not formulate statistical hypotheses before data collection, nor did we perform statistical hypothesis testing on the questionnaire responses. Consequently, we do not quantify the uncertainty surrounding the differences observed between the two conditions, and we cannot rule out the possibility that some of these differences occurred by chance. We mitigate this threat by reporting the complete response distributions for both the perceived ease of the decision process and confidence in the selected refactoring, rather than relying solely on summary measures. This allows readers to inspect the data underlying our observations. In addition, we release the questionnaires and raw per-subject responses as part of the replication package, supporting independent inspection and replication. Finally, throughout the evaluation, we frame our findings as tendencies observed within our sample rather than as statistically established effects.

8. Conclusions

This article addresses the challenge of deciding which security smell-related refactorings to apply first, a decision complicated by inter-service dependencies, varying quality attribute requirements, varying refactoring effort, and cross-team coordination. To support practitioners operating under these complex and dynamic conditions, we introduced a decision support method that accounts for urgency, implementation feasibility, and team dependencies. Our approach integrates three complementary visualizations: (i) an architectural view that allows practitioners to introduce the MSA information needed, (ii) a Kanban-inspired view that organizes refactorings based on urgency and implementation effort, and (iii) an ATAM-inspired view that highlights business relevance, key quality attributes, and required refactorings for each service.

We also presented a prototype tool that implements our method and assessed its perceived usefulness through a pretest-posttest study. After being introduced to the decision support method, participants perceived the process of balancing urgency, impacted quality attributes, and implementation effort as easier and reported greater confidence in their selected refactorings. This suggests that practitioners could leverage our proposed method, which provides structured, sprint-aligned guidance for making refactoring decisions in the presence of multiple interdependent refactorings, thereby helping them balance urgency, quality impacts, and coordination needs.

Future work will integrate the prototype tool with automated security smell detection mechanisms, such as KubeHound [15]. We will also extend the method to cover additional types of smells and refactorings relevant to MSAs, including architectural smells as proposed by Neri et al. [40]. Finally, we aim to validate the decision support method in an industrial context through experimentation, ensuring its applicability and effectiveness in real-world development settings.

CRedit authorship contribution statement

Francisco Ponce: Writing – original draft, Methodology, Investigation; **Jacopo Soldani:** Writing – original draft, Methodology, Investigation; **Gabriele Squarciapino:** Writing – original draft, Investigation; **Hernán Astudillo:** Writing – original draft, Methodology, Investigation; **Antonio Brogi:** Writing – original draft, Methodology, Investigation.

Dedication

This article is dedicated to Gian-Luigi Ferrari, “Giangi” for many of us. His inspiration, encouragement, enthusiasm, and generous support reinforced our beliefs in working on microservices security from a software engineering perspective, ultimately resulting in the decision support presented in this paper. This article hence stands as a witness of our gratitude and affection for Gangi.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly supported by the projects: *FREEDA* (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU; *SERICS* (PE000000014 - CUP H73C2200089001), under the National Recovery and Resilience Plan (NRRP), funded by the European Union - [NextGenerationEU](#).

Data availability

Research link provided: <https://zenodo.org/records/17295441>.

References

- [1] N. Kratzke, P.-C. Quint, Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study, *J. Syst. Softw.* 126 (2017) 1–16. <https://doi.org/10.1016/j.jss.2017.01.001>
- [2] Y. Wang, H. Kadiyala, J. Rubin, Promises and challenges of microservices: an exploratory study, *Empir. Softw. Eng.* 26 (4) (2021) 63. <https://doi.org/10.1007/s10664-020-09910-y>
- [3] O. Zimmermann, Microservices tenets, *Comput. Sci. - Res. Dev.* 32 (3) (2017) 301–310. <https://doi.org/10.1007/s00450-016-0337-0>
- [4] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables DevOps: migration to a cloud-native architecture, *IEEE Softw.* 33 (3) (2016) 42–52. <https://doi.org/10.1109/MS.2016.64>
- [5] P. Billawa, A. Bambhore Tukaram, N.E. Díaz Ferreyra, J.-P. Steghöfer, R. Scandariato, G. Simhandl, SoK: security of microservice applications: a Practitioners' perspective on challenges and best practices, in: Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22, ACM, Association for Computing Machinery, New York, NY, USA, 2022. <https://doi.org/10.1145/3538969.3538986>
- [6] J. Soldani, D.A. Tamburri, W.-J. Van Den Heuvel, The pains and gains of microservices: a systematic grey literature review, *J. Syst. Softw.* 146 (2018) 215–232. <https://doi.org/10.1016/j.jss.2018.09.082>
- [7] F. Ponce, J. Soldani, H. Astudillo, A. Brogi, Smells and refactorings for microservices security: a multivocal literature review, *J. Syst. Softw.* 192 (2022) 111393. <https://doi.org/10.1016/j.jss.2022.111393>
- [8] F. Ponce, J. Soldani, H. Astudillo, A. Brogi, Should microservice security smells stay or be refactored? Towards a trade-off analysis, in: Software Architecture, Springer International Publishing, Cham, 2022, p. 131–139. https://doi.org/10.1007/978-3-031-16697-6_9
- [9] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, A. Brogi, To security and beyond: on the impacts of microservice security smells and refactorings, in: 2023 XLIX Latin American Computer Conference (CLEI), IEEE, New York, USA, 2023, pp. 1–10. <https://doi.org/10.1109/CLEI60451.2023.10346146>
- [10] T. Besker, A. Martini, J. Bosch, Technical debt triage in backlog management, in: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE, New York, USA, 2019, pp. 13–22. <https://doi.org/10.1109/TechDebt.2019.00010>
- [11] P. Wizeny, F. Ponce, F. Rademacher, J. Soldani, H. Astudillo, A. Brogi, S. Sachweh, Towards resolving security smells in microservices, model-driven, in: 18th International Conference on Software Technologies (ICSOFT), INSTICC, SciTePress, Setúbal, Portugal, 2023, pp. 15–26. <https://doi.org/10.5220/0012049800003538>
- [12] F. Ponce, J. Soldani, H. Astudillo, A. Brogi, Towards teamwise informed decisions on microservice security smells, in: M. Galster, P. Scandurra, T. Mikkonen, P. Oliveira Antonino, E.Y. Nakagawa, E. Navarro (Eds.), Software Architecture, Springer Nature Switzerland, Cham, 2024, pp. 350–358. https://doi.org/10.1007/978-3-031-70797-1_23
- [13] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193), IEEE, 1998, pp. 68–78. <https://doi.org/10.1109/ICECCS.1998.706657>
- [14] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, F. Arcelli Fontana, A systematic literature review on technical debt prioritization: strategies, processes, factors, and tools, *J. Syst. Softw.* 171 (2021) 110827. <https://doi.org/10.1016/j.jss.2020.110827>
- [15] G. Dell'Immagine, J. Soldani, A. Brogi, KubeHound: detecting microservices' security smells in kubernetes deployments, *Future Internet* 15 (7) (2023). <https://doi.org/10.3390/fi15070228>
- [16] T. Cerny, A.S. Abdelfattah, A.A. Maruf, A. Janes, D. Taibi, Catalog and detection techniques of microservice anti-patterns and bad smells: a tertiary study, *J. Syst. Softw.* 206 (2023) 111829. <https://doi.org/10.1016/j.jss.2023.111829>
- [17] R. Verma, K. Kumar, H.K. Verma, Code smell prioritization in object-oriented software systems: a systematic literature review, *J. Softw.: Evol. Process* 35 (12) (2023) e2536. <https://doi.org/10.1002/smr.2536>
- [18] T. Haendler, S. Sobernig, M. Strembeck, Towards triaging code-smell candidates via runtime scenarios and method-call dependencies, in: Proceedings of the XP2017 Scientific Workshops, XP '17, Association for Computing Machinery, New York, NY, USA, 2017. <https://doi.org/10.1145/3120459.3120468>
- [19] A. Gupta, N.K. Chauhan, A severity-based classification assessment of code smells in kotlin and java application, *Arab. J. Sci. Eng.* 47 (2022). <https://doi.org/10.1007/s13369-021-06077-6>
- [20] R. Malhotra, P. Singh, Exploiting bad-smells and object-oriented characteristics to prioritize classes for refactoring, *Int. J. Syst. Assur. Eng. Manag.* 11 (S2) (2020) 133–144. <https://doi.org/10.1007/s13198-020-01001-x>
- [21] T. Alshammari, M. Alshayeb, Toward a software bad smell prioritization model for software maintainability, *Arab. J. Sci. Eng.* 46 (9) (2021) 9157–9177. <https://doi.org/10.1007/s13369-021-05766-6>
- [22] F. Arcelli Fontana, V. Ferme, M. Zanon, R. Roveda, Towards a prioritization of code debt: a code smell intensity index, in: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), IEEE, New York, USA, 2015, pp. 16–24. <https://doi.org/10.1109/MTD.2015.7332620>
- [23] T.W.W. Aung, Y. Wan, H. Huo, Y. Sui, Multi-triage: a multi-task learning framework for bug triage, *J. Syst. Softw.* 184 (2022) 111133. <https://doi.org/10.1016/j.jss.2021.111133>
- [24] S. Lim, S. Zaidi, H. Woo, C.-G. Lee, Toward an effective bug triage system using transformers to add new developers, *J. Sensons* 2022 (2022) 19. <https://doi.org/10.1155/2022/4347004>
- [25] F. Pecorelli, F. Palomba, F. Khomh, A. De Lucia, Developer-driven code smell prioritization, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 220–231. <https://doi.org/10.1145/3379597.3387457>
- [26] N. Sae-Lim, S. Hayashi, M. Saeki, Context-based code smells prioritization for prefactoring, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, New York, USA, 2016, pp. 1–10. <https://doi.org/10.1109/ICPC.2016.7503705>
- [27] N. Sae-Lim, S. Hayashi, M. Saeki, Revisiting context-based code smells prioritization: on supporting referred context, in: Proceedings of the XP2017 Scientific Workshops, XP '17, Association for Computing Machinery, New York, NY, USA, 2017. <https://doi.org/10.1145/3120459.3120463>
- [28] S.A. Vidal, C. Marcos, J.A. Díaz-Pace, An approach to prioritize code smells for refactoring, *Autom. Softw. Eng.* 23 (2016) 501–532. <https://doi.org/10.1007/s10515-014-0175-x>

- [29] A. Aldaej, C. Seaman, A rule-based decision model to support technical debt decisions: a multiple case study of web and mobile app startups, *Inf. Softw. Technol.* 175 (2024) 107542. <https://doi.org/10.1016/j.infsof.2024.107542>
- [30] C. Zhong, H. Huang, H. Zhang, S. Li, Impacts, causes, and solutions of architectural smells in microservices: an industrial investigation, *Softw.: Pract. Exp.* 52 (12) (2022) 2574–2597. <https://doi.org/10.1002/spe.3138>
- [31] P. Bacchiega, D. Rusconi, P. Mereghetti, F.A. Fontana, Refactoring of a microservices project driven by architectural smell detection, in: 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), IEEE, 2024, pp. 281–288. <https://doi.org/10.1109/ICSA-C63560.2024.00040>
- [32] M. Agostini, J. Soldani, A. Brogi, Detecting and resolving bad organisational smells for microservices, in: Proceedings of the 19th International Conference on Software Technologies - ICSOFT, INSTICC, SciTePress, 2024, pp. 67–78. <https://doi.org/10.5220/0012851200003753>
- [33] F. Ponce, J. Soldani, C. Taramasco, H. Astudillo, A. Brogi, Triaging microservice security smells, with triSS, in: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 698–706. <https://doi.org/10.1145/3661167.3661282>
- [34] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, The NFR Framework in Action, Springer US, Boston, MA, Boston, MA, 2000. https://doi.org/10.1007/978-1-4615-5269-7_2
- [35] S. Kapferer, O. Zimmermann, Domain-driven service design: context modeling, model refactoring and contract generation, in: Service-Oriented Computing, Springer International Publishing, Cham, 2020, pp. 189–208. https://doi.org/10.1007/978-3-030-64846-6_11
- [36] S. Panichella, M.I. Rahman, D. Taibi, Structural coupling for microservices, in: Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER, INSTICC, SciTePress, Setúbal, Portugal, 2021, pp. 280–287. <https://doi.org/10.5220/0010481902800287>
- [37] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, A. Zündorf, Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations: the case for microservice architecture, *SN Comput. Sci.* 2 (6) (2021) 459. <https://doi.org/10.1007/s42979-021-00825-z>
- [38] N.B. Moe, A. Aurum, T. Dybå, Challenges of shared decision-making: a multiple case study of agile software development, *Inf. Softw. Technol.* 54 (8) (2012) 853–865. Special Issue: Voice of the Editorial Board. <https://doi.org/10.1016/j.infsof.2011.11.006>
- [39] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, Kluwer Academic Publishers, Amsterdam, Netherlands, Amsterdam, Netherlands, 2000.
- [40] D. Neri, J. Soldani, O. Zimmermann, A. Brogi, Design principles, architectural smells and refactorings for microservices: a multivocal review, *SICS Softw.-Intensive Cyber-Phys. Syst.* 35 (1) (2020) 3–15. <https://doi.org/10.1007/s00450-019-00407-8>