Smelling the Architecture: An Action Research Study on Improving Practitioners' Architectural Practices

Theodor Risager, Fatima Osman Aden Mohamed
Department of Computer Science
Aalborg University, Denmark
{trisag19, fmoham19}@student.aau.dk

May 2024

Summary

High-quality software architecture is crucial for modern systems, as emphasized by Martin Fowler. While a well-defined architecture is essential, excessive upfront design can hinder agility. Conversely, neglecting architecture leads to Technical Debt (TD), making systems complex and difficult to maintain. TD, a concept introduced by Cunningham in 1992, accrues when suboptimal decisions are made for short-term gains. This debt increases over time if left unaddressed, potentially hindering development progress. Architectural Debt (AD), a significant subcategory of TD, contributes substantially to maintenance efforts. Despite its impact, studies show that a majority of practitioners lack TD management practices.

This study investigates methods to improve architectural practices and mitigate AD creation. We apply the *Smell of the Week* activity, focusing on identifying and discussing architectural smells within the codebase. An architectural smell is a pattern within the architecture that indicates a potential problem, rather than being a problem itself. Prior research on TD management has primarily focused on frameworks for the identification, prioritization, and rectification of TD items. Our study builds upon this work by exploring a novel approach to AD management and its impact on team communication and architectural competencies.

We employ Action Research (AR) to bridge theory and practice, aiming to both test existing theory in real-world settings and contribute to the body of knowledge through practical insights. The study is conducted in collaboration with a team of six developers at the IT services department of a Danish university, who we refer to as ITS. Our AR study consists of two intervention cycles. The first cycle focuses on diagnosing the team's architectural challenges, identifying a need for AD management, and a strong emphasis on maintainability. To address this, we introduce the *Smell of the Week* activity, which entails weekly meetings to discuss and identify specific architectural smells within their codebase. This activity shows initial promise, leading to a second cycle where we adjust the frequency to align with the team's three-week sprint cycle. Through both interventions, we derive three key lessons, that evolving architectures require a team of aligned and competent developers, that the *Smell of the Week* activity effectively initiates architectural discussions, and that architectural smells can be categorized as *Sneaky* or *Fix-it-Once* smells.

Motivation

Software architecture is fundamental for high-quality software. Without sound architecture, software can become difficult to maintain, extend, and upgrade, potentially leading to increased operational costs. A compelling example of the impact of architectural choices is provided by Amazon [22], where a service initially implemented as a microservice was later transformed into a monolithic application, resulting in a 90 % reduction in running costs.

Inspired by the significance of architecture, our research aims to understand the processes, decision frameworks, and challenges faced by practitioners in choosing and maintaining effective architectures. This knowledge is sought to enhance our own development capabilities. We have chosen action research as our methodology, enabling us to both explore existing literature and apply theoretical concepts in a practical setting. This approach allows us to evaluate the real-world effectiveness of practices considered "good" in the literature.

Our research context is the IT service department (ITS) of a Danish University. Initial interviews with ITS revealed the architecture of their systems, highlighting issues such as technical debt, misalignment, and maintenance challenges. Given the team's prioritization of maintainability, our research focuses on evolving architecture, defined as an architecture that can adapt to changing requirements while remaining maintainable.

This leads to the following research question:

RQ How can architectural practices be improved in teams that develop and maintain evolving systems with a high risk of accumulating Architectural Debt?

Our 9th semester concluded that the *Smell of the Week* practice showed promise and that further investigation was warranted. We therefore continued investigating it throughout the 10th semester. Initially, we considered expanding the project to include additional teams from other organizations. However, this was deemed infeasible due to our focus on the long-term effects of the practice. Onboarding a new team at this late would have limited our ability to gather comprehensive data on these effects. Furthermore, incorporating another team would have diluted our attention and potentially compromised the depth of our observations.

Improving Architectural Practices to Mitigate Architectural Debt in Evolving Systems: An Action Research Study

Theodor Risager, Fatima Osman Aden Mohamed
Department of Computer Science
Aalborg University, Denmark
{trisag19, fmoham19}@student.aau.dk

May 2024

Abstract

Architectural Debt (AD) is the largest contributor to Technical Debt (TD), a prevalent issue in most IT organizations. AD involves short-sighted decisions, often due to time constraints, that reduce efficiency and increase maintenance costs over time. If unresolved, these decisions accumulate until development grinds to a halt. This Action Research (AR) study examines the effectiveness of the *Smell of the Week* practice in managing AD in a practical context. *Smell of the Week* involves selecting a subset of architectural smells to identify and discuss during meetings. Our findings reveal that this practice is useful for initiating discussions about architectural smells and formulating management strategies. Additionally, we introduce a new categorization of architectural smells as *Sneaky* and *Fix-it-Once* to further the understanding of their management. Furthermore, we emphasize that evolving architectures require aligned and competent developers capable of distributed decision-making.

1 Introduction

High-quality software is a cornerstone to success for modern systems. As Martin Fowler emphasizes, the initial investment in high-quality is ultimately justified by the long-term benefits of a more maintainable and scalable system [20]. A critical factor in achieving this quality is the software's architecture, the foundational design decisions that determine its long-term viability [20].

While a well-defined architecture established early on in the development lifecycle is crucial [4], an overly rigorous upfront design process lowers the team's agility [45]. Conversely, neglecting architecture altogether leads to Technical Debt (TD), making the system increasingly complex and difficult to maintain.

Cunningham introduced the concept of TD in 1992 [14], and accumulates throughout the development lifecycle when suboptimal decisions are made to prioritize short-term gains, such as faster time to production. While these decisions may be justifiable at the moment, they incur a "cost" in the form of future complexity and maintenance effort. This cost, like financial debt, grows over time if left unaddressed. Unmanageable TD can significantly impede development progress, potentially bringing entire organizations to a standstill.

TD can be divided into subcategories, with Architectural Debt (AD) being a significant contributor to maintenance efforts within software projects [50, 32, 39, 17, 51]. In [50], they identify five different types of AD in open source projects. Their findings indicate that AD accounts for 85 % of maintenance ef-

fort. This is backed up by [39], which found AD to be responsible for 54 % of the maintenance efforts while only being present in 14 % of the source files. This underscores the significant impact that AD has on maintenance. Furthermore, [17] found that 65 % of practitioners do not have any TD management practices, further underlining the need for improvement.

AD research can be broadly categorized into two key areas: identification and management. Identification techniques focus on developing methods and tools to pinpoint locations within the codebase that may contain AD. Automated approaches leverage code analysis tools to identify files or classes exhibiting architectural weaknesses [51, 11, 12, 49]. These characteristics may include cyclic dependencies, scattered functionality, or God components [2]. Practitioner-oriented frameworks offer methodologies and guidance to software engineers for systematically evaluating their codebase and manually identifying potential architectural issues [26].

AD management research efforts explore various strategies to analyze the impact of AD and prioritize its repayment. One line of research describes different types of AD and the potential "interest" (i.e., negative impact) they incur over time [28, 27, 51]. This allows development teams to prioritize debt repayment based on its projected future burden on maintenance efforts. Another area focuses on frameworks that facilitate prioritizing debt items based on various factors such as severity, effort required for repayment, and potential impact on future development [28].

AD is a practical reality, not a theoretical ideal. Unlike perfect systems in theory, real-world development involves deadlines, resource constraints, and varying levels of expertise. These factors often necessitate compromises, leading to the accumulation of AD. Therefore, it is crucial to investigate AD within a practical context. Action Research (AR) is particularly well-suited for this purpose. AR fosters collaboration between researchers and practitioners, bridging the gap between theory and practice [41, p. 3].

Within the broad definition of TD, there exists plenty of research covering various areas. Borup et al. [8] and Oliveira et al. [33] utilize AR to investigate managing TD within teams. Both papers utilize Guo and Seaman's TD management framework [37], focusing on the identification, prioritization and rectification of TD items. Similarly in [52], an AR is conducted on a large company to develop processes to identify, document, and prioritize TD. The IT company subsequently adopted a prioritized TD backlog to increase visibility and manageability. However, the body of AR specifically investigating AD remains limited, highlighting a significant gap in this area of research.

Our goal is to contribute to the ongoing AR on AD, with a focus on practitioner-oriented analysis in contrast to the prevalent use of automated tools in the existing literature. We aim to investigate how to minimize AD creation throughout the development lifecycle, utilizing AR to intervene and evaluate the effectiveness of AD management techniques. Effectiveness, in this context, is the ability of the technique to promote productive architectural discussions, as this fosters knowledge sharing, team alignment, and education.

This collaborative effort involves a small team, of six developers, within the IT service department of a Danish university responsible for the development and maintenance of 10 internal software systems. These systems undergo frequent requirement changes, necessitating the continuous evolution of the software. A well-defined and well-maintained architecture is crucial in this context, facilitating future system evolution. More importantly for the team, it lowers the amount of resources needed to change and maintain the systems. However, the team has not been able to achieve this ideal. These systems have been developed over an extended period by various developers, including both internal developers and external contractors. This heterogeneous development history has led to an inconsistent and suboptimal architecture. Furthermore, resource constraints within the IT department and bureaucracy make large-scale refactoring initiatives impractical.

Through AR, we aim to: (1) improve the team's architectural practices and (2) contribute valuable empirical evidence on the effectiveness of these architectural practices. This dual focus is guided by the following research question:

RQ How can architectural practices be improved in teams that develop and maintain evolving systems with a high risk of accumulating Architectural Debt?

By answering this question, our research will contribute to the literature in three ways: (1) shedding light on how these architectural practices can be implemented in practice, (2) evaluating their effectiveness in fostering productive architectural discussions within the chosen team, and (3) evaluating its transferability for future collaborations.

2 Theory

To ensure a shared understanding, this section draws upon existing research to define the key terms: software architecture, Architectural Debt (AD), architecture smells, and the architect's role in managing this.

2.1 Software Architecture

Architecture, as Martin Fowler puts it, embodies "the decisions that you wish you could get right early in a project." representing the important decisions that require focus in the early stages of development [18]. As Fowler also states, architecture deals with "the important stuff", namely the components and modules that would be costly to change. As defined by Bass et al, software architecture also refers to the arrangement of elements in a system that are needed to understand and reason about the system [3]. Philippe Krutchen adds that "Architecture is what remains when you remove everything unnecessary from the system, leaving only what's essential to describe the system's functionality"[24].

Architecture operates across multiple abstraction levels. For instance, at a macro level, a system may consist of numerous subsystems, each serving a distinct specialized functionality. Alternatively, the system might use a microservices architecture, where it is split into separate services. The architecture then defines how these services are organized and how they interact with each other. Furthermore, architecture also extends to not only the relationship between classes but also the underlying structure of each class. However, the level most appropriate for the architecture is the level understood by all the developers [18].

Software architecture serves as a design blueprint, providing an abstraction to manage the system's complexity. It describes how system elements interact, fit together, and fulfill requirements, defining each element's responsibilities and their interactions with the system and its environment. Acting as a bridge between system requirements and implementation, it addresses nonfunctional requirements or quality attributes, anticipating necessary adaptations for the system's evolution [21, p. 4-6].

Software architecture quality is evaluated through the measurement of various attributes. A key determinant of high-quality architecture lies in attributes like maintainability and evolvability. Maintainability involves the ease of identifying and rectifying system defects, or addressing bugs efficiently [29]. Evolvability, on the other hand, refers to the ease of integrating new requirements [26]. These attributes are important in defining what constitutes high-quality architecture, as they directly contribute to the system's long-term sustainability, flexibility, and ability to meet evolving demands. These system quality attributes are often sacrificed to fulfill immediate business demands such as time-to-market and development costs [25].

2.2 Role of the Architect

Architects are often stereotyped as high-level 'astronauts' with overly abstract designs, 'techno-geeks' who disregard stakeholder needs and business concerns, or detail-oriented 'dwarfs' focused on minute details [10]. In contrast to limiting archetypes, the pragmatic architect, as introduced by Buschmann [10], embodies a versatile blend of skills. They balance abstract vision with technical expertise and possess strong communication skills to bridge the gap between stakeholders and developers. This includes mentoring developers, effectively communicating architectural concepts, creating balanced architectural documentation, and understanding the pros and cons of different architectural choices.

Fowler follows this idea by emphasizing a shift

in architectural focus away from top-down decision-making and towards empowering development teams [18]. A core function of the architect becomes mentoring, ensuring that developers are equipped to make informed architectural choices aligned with the system's requirements. This fosters an environment where "hands-on architects" work directly with the codebase, reducing bugs and promoting architectural best practices [35].

In this collaborative model, we propose that a project's architectural responsibilities encompass several key aspects. Architects guide developers, sharing architectural knowledge and principles, and negotiating new requirements while ensuring they are fully understood and reflected in the architecture. They facilitate clear and consistent communication between stakeholders and developers. Further, they strive for a balanced architecture that is neither overly abstract nor excessively granular. Most importantly, architects prioritize maintainability and evolvability, ensuring the system can easily accommodate future changes and bug fixes.

This definition is complemented by Boehm and Turner in [7] where they present the five dimensions affecting method selection (agile vs disciplined), two of which are criticality and dynamism. Criticality is defined as "Loss due to the impact of defects" and in order to be agile this should be as little as possible. Dynamism is the percentage change in requirements, where a large percentage of changes requires an agile method. In order to manage these changes the team needs to have a culture, where they are comfortable with this chaos. This further necessitates a professional and experienced team, consisting primarily of what Boehm [7] describes as level 2 and 3 developers. These developers are "able to [tailor or] revise a method (break its rules) to fit an unprecedented new situation".

In teams with constrained resources, like the one we are collaborating with, the concept of a dedicated architect might be impractical. By adopting a collective responsibility model, everyone on the team shares ownership of the architecture. This distributes architectural knowledge, reduces single-person dependencies, and promotes a more sustainable approach in the event of personnel changes.

2.3 Architectural Debt & Architecture Smells

Ernst et al. states that architectural issues are the most significant source of TD [17]. Consequently, managing AD becomes a paramount responsibility for architects. Unlike other forms of TD, AD focuses on structural deficiencies within the system's architecture, impacting internal quality and remaining invisible to end-users [23]. AD often stems from developers implementing sub-optimal solutions due to time constraints, particularly tight deadlines.

The intended lifespan of a system directly influences the effort that should be dedicated to managing AD. Short-lived systems may not manifest the negative consequences of AD, whereas long-lived systems will inevitably face these challenges [44]. For long-lived systems, the management of AD is an ongoing process, as even after resolving existing debt, new architectural shortcomings may be introduced over time.

Architectural smells serve as a means to identify potential areas where AD may reside. These indicators highlight opportunities for improvement; however, their presence does not inherently signify sub-optimal architectural decisions [19]. Context is essential, as specific structural choices may have been implemented deliberately due to external constraints or requirements.

Architectural smells can impact different quality attributes. For instance, the Empty Semi Trucks smell, characterized by an excessive number of requests to complete a task, negatively affects performance. Similarly, the Too Many Standards smell, where different languages, protocols, and frameworks are used, hinders comprehensibility and subsequently impacts maintainability [31].

Given that architectural smells serve as indicators of potential improvement areas, we will leverage them to enhance architectural practices at ITS.

3 Action Research Method

To understand and improve architectural practices, we employ Action Research (AR), a collaborative

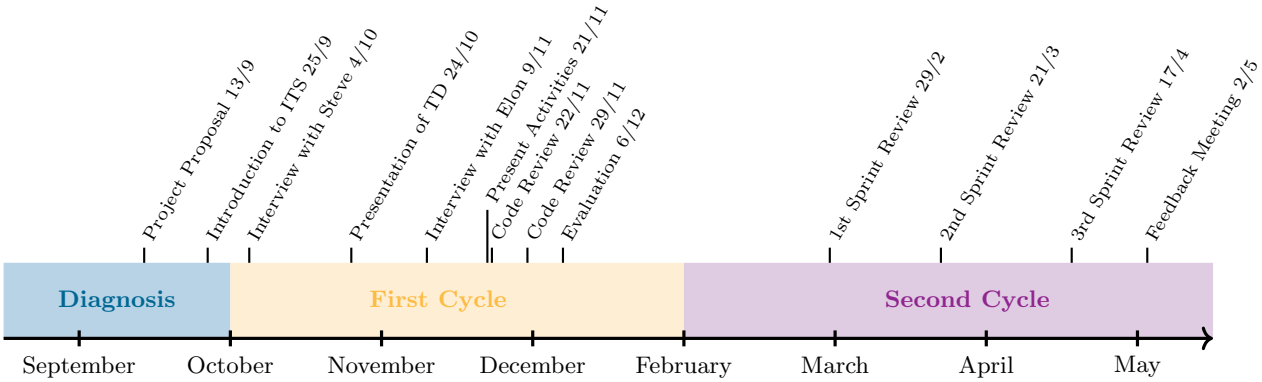

Figure 1: Action Research study timeline

methodology rooted in social science that aims to effect change while conducting research [47]. AR bridges research with action through cyclical phases of diagnosing, planning, acting, evaluating, and learning [41]. In our study, this involved investigating practitioners' architectural practices to identify areas of improvement using architectural smells, while also providing feedback to the literature on the practical utility of these techniques.

This collaborative study involves ITS, an IT department with 150 employees serving an organization of 3,500+ employees and 20,000+ users, where one of the authors is employed. This is a well-established IT organisation that manages many different evolving systems, therefore making it an interesting context for this research. We collaborate with a small development team within ITS, consisting of one team lead and six developers working on 10 distinct systems, separate from the author's team. To maintain participant anonymity, all developers will be identified using pseudonyms throughout this study. The members of the team have reviewed and approved this paper for accuracy and the absence of misconceptions. Subsequent references to ITS refer to this specific team.

Interviews with the ITS development team followed a semi-structured format [5, p. 143], also known as an interview guide [34, p. 438], combining pre-determined questions and research goals with opportunities for open-ended discussion. This approach facilitated the exploration of both the primary research topics and supplementary insights. All interviews were audio-recorded and transcribed using Goodtape[1]. The resulting recordings and transcripts, a total of 150 pages, served as primary data sources throughout the project. Immediately after each interview, the recordings underwent multiple reviews to extract relevant information. Additionally, at the end of each research cycle, all interview data was re-examined. This iterative review process allowed for the refinement of interpretations based on evolving understandings and the acquisition of new knowledge.

The following sections detail two cycles of action research, following the diagnose, plan, act, evaluate, and learn phases outlined in [41]. Each act cycle includes sub-cycles adhering to these phases.

3.1 Weekly Architecture Reviews

In this initial cycle, we focused on diagnosing the problematic situation at ITS through a series of five meetings over two months. This first cycle included just the team lead and three of the six developers. Initial interactions involved an introductory meeting, an interview with a developer, and a presentation highlighting the system's shortcomings and proposing three project focuses: technical debt, mainte-

[1] `https://goodtape.io`

nance, and misalignment. Based on the feedback we gained from the meeting, and the team's emphasis on maintainability, we changed the direction of the project towards evolutionary architecture.

To understand their practices, we conducted an interview with a developer, revealing the need for a common understanding of architecture and the architect's role. We then presented our literature-based definition, presented in Section 2, to the team.

The action plan involved integrating the *Smell of the Week* intervention into their weekly code review. This activity was chosen because of its relevance to evolutionary architecture and the team's perceived excitement for it. It includes selecting a subset of architectural smells from Appendix A for the developers to identify in their codebase and discuss. We participated as participant observers [38] in three one-hour code reviews over the following month, which concluded with half of the last review allocated for feedback. Continuous refinements were made to the structure of the activity to enhance its effectiveness.

The evaluation phase involved recording, transcribing, and revisiting interviews for insights, as presented in Section 4. Following each review, the researchers discussed its effectiveness in facilitating productive architectural discussions.

All findings for this cycle were first formalized in preparation for our semester evaluation, which includes our supervisor and an external censor, and again for our final thesis exam. The primary finding was the *Smell of the Week* activity's potential to promote architectural discussions and shared responsibility within the team. Section 4.1 and Section 4.2 detail our findings.

3.2 Sprint Architecture Reviews

Building on the learnings from Cycle 1, we refined the intervention for Cycle 2. The frequency was adjusted to align with the team's three-week sprint review cycle, and participation was expanded to include the entire development team. Given the limited time per review (30 minutes), we prioritized a single architectural smell per session, leveraging the framework in [6] to identify smells with the highest potential negative impact on maintainability, a key concern identified in Cycle 1. Additionally, to avoid overwhelming the team, the analysis was restricted to code developed within the sprint.

Over three months, three 30-minute sprint architecture reviews were conducted. A final 30-minute feedback session with all six developers concluded the cycle. As in Cycle 1, all reviews were recorded, transcribed, and revisited for insights presented in Section 4. After each review, the researchers again discussed its effectiveness in facilitating productive architectural discussions.

All findings for this cycle were formalized in preparation for our thesis exam, which includes our supervisor and the external censor from our semester evaluation following the first cycle. The primary finding from Cycle 2 was that the Smell of the Week activity, while valuable, required a significant time investment relative to its perceived value. Integrating the activity into existing code reviews emerged as a more practical approach. Section 4.3 details these findings.

4 Findings

In this section, we present our findings from the problematic situation and our two interventions.

4.1 The Problematic Situation

The five interviews with the developers at ITS revealed the accumulation of significant TD and AD within their Gateway. This debt impedes maintenance efforts and hinders their ability to accommodate new development as their team manages a growing portfolio of systems.

The development team attributed the rise in technical and architectural debt within the Gateway project to several specific causes. Each cause is explored in the subsections below.

4.1.1 Unclear Feature Requirements

Within the organization, there exist many different departments, such as study administration, finance, and the various faculties. Whenever stakeholders from these departments request a new feature, the

requirements often lack clarity. There tends to be a gap in considering the feature's complete lifecycle, including its creation, maintenance, and eventual deletion. Consequently, developers must invest considerable time and resources in uncovering the full scope of the requirements. Additionally, the feature might impact or be relevant to other departments, necessitating broader discussions to identify any further implications.

As Elon with seven years of experience expressed:

> *“ I could wonder why there isn't anyone [...] who, across all teams, says, 'Now, those people over in student services want to do this branch thing. Maybe we should [...] ask these stakeholders, or talk to them.' Something new is coming in student services, branches are coming in education programs. It works like this and that. You can have something like this. Legally, it works like this. What does it mean for you? And then you go back and implement [the system]. ”*
>
> *– Elon*

The development team highlights the lack of an enterprise architect at ITS, a gap that hinders their ability to obtain a holistic organizational perspective. This absence impedes the comprehensive definition of features, as it limits the consultation of all relevant stakeholders.

4.1.2 Requirements Always Change

The development team also highlighted the challenge presented by frequent and sometimes sweeping changes in requirements. These changes could stem from new regulations imposed by policymakers, to which the team is legally obligated to adhere, or from evolving stakeholder needs.

As Elon noted:

> *“ So, we are unfortunately bound by legislation. And there are no politicians sitting and thinking, 'What if we mess with this? How will it affect the flow of various IT systems at universities? ”*
>
> *– Elon*

This external imposition of requirements can lead to situations where fundamental assumptions about the system are altered. This necessitates significant rework, often involving suboptimal architectural decisions due to incomplete or unclear requirements and limited time or resources.

Elon goes on to describe this challenge:

> *“ So now you take a system with several thousand lines of code. It has a lot of validation and logic to ensure that you are not allowed to tamper with things. And then you say, 'Now we would like to be able to tamper with things.' ”*
>
> *– Elon*

The team further expressed concern about the marginalization of their input and opinions regarding such requirement changes, even when these decisions significantly impact their work.

4.1.3 Too Specific Requirements

The development team expressed concern over the challenge of non-technical stakeholders sometimes dictating specific implementation details. This can hinder their ability to design a coherent system architecture, as decisions should be driven by a holistic understanding of requirements rather than isolated preferences.

As Elon stated:

> *“ ... some people have just heard that microservices are really smart. So can't you just make it? ... some people are getting involved in how [it should be done], who shouldn't be involved because then you can't have this coherent architecture. ”*
>
> *– Elon*

While pushing back on unsuitable requirements is a potential solution, the team acknowledges limitations in exercising this control. Their ultimate responsibility lies in implementation, not requirement definition.

> *“ But that's where the responsibility for [the gateway] isn't necessarily something we determine. Someone could come along and say, 'It would be bloody brilliant if this gateway could just make my coffee.' And then there should be someone saying, 'No, that's not the specification for the gateway.' ”*
>
> *– Elon*

This problem extends to the broader system landscape. Elon described a situation where an architect, despite advocating for the streamlining of ID usage across systems, lacked the authority to implement the change due to not being the product owner.

> *“ Well, couldn't we just decide that wherever the most complete set of IDs is, that's the system? They're on nine characters or whatever. Can't we just say that's the ID now? Then fix the other systems. ... So it's the ID. But the architect couldn't decide that, because he's not the product owner of these things. ”*
>
> *– Elon*

The ability to challenge or shape requirements is constrained by a lack of ownership over their definition, potentially leading to architectural compromises. This highlights a potential disconnect between technical expertise and decision-making authority within the organization.

4.1.4 Limited Resources

The development team faces a significant challenge in securing stakeholder support for addressing TD. Since these improvements will not yield immediate, tangible benefits like new features or enhanced performance, stakeholders tend to deprioritize them in favor of projects with more visible outcomes.

As explained by Elon:

> *“ But it's also about selling it, because how do we sell it? You get exactly the same piece of software as you had before. It looks a little different. It costs just half a year. So we'd rather have some new features. And that's difficult because then you have to sell it with [other features]. ”*
>
> *– Elon*

Consequently, the team resorts to addressing maintenance issues by strategically integrating them into the development of new features. This allows them to undertake essential upkeep while still delivering the additions stakeholders request.

However, this approach has a significant drawback: its limited ability to thoroughly address the existing debt. This results in the ongoing accumulation of technical and architectural debt, a problem that has now reached a point where implementing even minor new features takes days rather than hours, highlighting the consequences of this neglect.

4.1.5 Primary Findings from the Problematic Situation

The development team faces the ongoing challenge of frequent and sometimes sweeping changes in requirements, while not having the authority to challenge these requirements, which over time have resulted in the accumulation of large amounts of TD.

To effectively accommodate these changes, the architecture must be highly evolvable, as outlined in Section 2.1. This evolvability necessitates highly competent architects who can implement new requirements while maintaining overall quality, as emphasized in Section 2.2.

Based on this we will employ the *Smell of the Week* activity introduced by Martin Fowler [19] for our first intervention. This activity involves selecting a specific architectural smell for weekly architecture reviews. The team will collaboratively analyze the codebase for the selected smells and discuss potential solutions.

The activity will foster a shared architectural vocabulary while strengthening the team's architectural competencies, leading to proactive codebase improvements and reduced the likelihood of introducing future debt.

4.2 The Intervention - Weekly Architecture Review

We implemented the *Smell of the Week* activity into the team's existing weekly code review meetings, participating in three reviews. For the first review, we selected 15 relevant code smells from [31] (re-represented in Appendix A). The goal of this first review was to introduce the concept of smells to the team. During the first review, we collaboratively searched the codebase for these smells together with the development team. While discussions initially deviated from the selected smell, they proved valuable. We analysed various code issues, linking them to relevant architectural smells. This exposure to the concept of smells sparked a meaningful discussion. As a result, the team gained greater awareness of existing AD present in the codebase.

In the second review, we shifted to a checklist-based approach, where one of the developers was responsible for sequentially going through the list. Each of the smells was then discussed in depth by all developers, with the aim of identifying problems and discussing corresponding solutions. This proved significantly more effective, allowing for focused analysis of individual smells.

In the third review, a single smell was selected, and the team was prompted to document the problem, how they would fix it if they were to start over, and how they would reach that goal. This documentation process ensured a shared understanding of the problem and its potential resolutions. Additionally, it improved discussion focus and helped the team stay on track.

We concluded the third review by soliciting developer feedback on the intervention. The following subsections will detail this feedback.

4.2.1 Discovering Hidden Parts of the System

Feedback highlighted the effectiveness of the *Smell of the Week* activity in exploring various system components that might otherwise have been overlooked. Helping to raise awareness of the system's current state and associated concerns.

> *“ It's a very interesting way, trying in some logical manner to go through [the system] ... One might end up in some corners of the system that one normally doesn't touch or hasn't thought about. ”*
>
> *– Elon*

As Elon remarked, the process provided a logical method for delving into potential areas of improvement, even those typically overlooked or not previously considered. This holistic examination prompted the team to engage in collaborative discussions about addressing identified issues and brainstorming potential solutions.

4.2.2 Continuous Maintenance

The team sees this activity as a valuable tool for ongoing maintenance of both legacy systems and systems under active development. This stems from its ability to systematically uncover architectural issues that can be addressed in subsequent sprints.

As Bill, a developer with 12 years of experience, stated:

> *“ We are thinking that maybe we could use this as a tool to trigger some ongoing maintenance tasks. ”*
>
> *– Bill*

By identifying architectural weaknesses, the activity prompts the creation of specific maintenance tasks that can be integrated into future development sprints. This proactive approach ensures that architectural issues are not overlooked.

While the team recognized the benefits of continuous maintenance, Elon raised a valid concern:

> *“ Could it be demotivating because you're discussing something you can't do anything about? Because existing systems, it's kind of like they work, but... ”*
>
> *– Elon*

Their concern highlights the potential for demotivation when confronted with overwhelming AD or not enough resources for its management.

However, the team still recognizes the value of *Smell of the Week* as a tool for identifying and managing AD.

> *“ If someone from above said that you have x amount of time each month for maintenance, then it could have been a good tool to grab some things, [...], something needs to be done here. ”*
>
> *– Bill*

As the developer noted, if they were allocated resources for the management of their AD, the *Smell of the Week* activity would be a tool they would use.

4.2.3 Primary Findings From Cycle 1

The team suggests two primary use cases for *Smell of the Week*: the evaluation of legacy systems and the assessment of newly created code within actively developed systems.

For the continuous development of systems, they suggest using it for the evaluation of newly created components. A suitable time for this could be during, e.g. sprint reviews for SCRUM teams. This evaluation would examine how well the new component integrates with the existing architecture, whether it has introduced any new architectural smells, and if broader refactoring is needed to maintain a high-quality architecture of the system. This proactive approach promotes ongoing architectural awareness throughout the development process, preventing suboptimal patterns from becoming entrenched and ensuring timely refactoring to maintain the quality of the architecture.

Based on this, our next intervention will apply *Smell of the Week* in the context of continuous development, by implementing it into their triweekly sprints as an addition to their sprint review.

4.3 The Intervention - Sprint Architecture Review

We implemented the *Smell of the Week* activity into the team's triweekly sprint review. All the sprint reviews followed the same structure. We began by presenting the selected smells and their descriptions. We did not provide our interpretation of the descriptions, we left this for the team to discuss. When they had come to a common understanding of the smell, we went on analysing the code for it.

During the second and third reviews, by request of the team, we sent the selected smells one week prior to the review. This ensured that each developer had time to find relevant code snippets for our review. This increased the effectiveness of the review, as we spent less time on analysing the code, and more on discussing it. From this intervention, we gained the following findings.

4.3.1 Categorizing Smells

During the feedback session, a distinction emerged between two different categories of architectural smells:

> *“ Seriously. It's typically something that sneaks in after the application is built, where a strange requirement might come up. It's a kind of sneaky land. ”*
>
> *– Elon*

Sneaky smells, as noted by Elon, these smells often manifest gradually over time due to evolving requirements or incremental changes. God classes, for instance, do not emerge suddenly but typically grow in size and complexity as responsibilities are added. Contrary to this, we have *Fix-it-Once* smells, such as Too Many Standards, that often can be addressed with a single concentrated intervention and subsequently maintained with relative ease.

> *“ I would say that Separation of Concerns, Bloated Service, and Circuitous Treasure Hunt are probably the most relevant. Because we have pretty good control over our standards regarding which languages, tools, frameworks, and so on we use. ”*
>
> *– Alan*

Their SCRUM master with four years at ITS, Alan, added that the *Sneaky* smells are more relevant throughout the evolution of the system, whereas the *Fix-it-Once* smells only need attention once in a while.

4.3.2 Improving Code Review

Alan mentions that a dedicated *Smell of the Week* activity might be excessive, as its benefits could be covered by implementing it as part of their weekly code review.

> *“ I might see it more as something we could do in connection with Code Reviews, for example. It's probably not necessary to have it as a recurring activity beyond Code Reviews if it's already being done there. ”*
>
> *– Alan*

He suggests that rotating the selected smell for each code review would help address more potential issues.

Bill further highlights the potential of utilizing smells during code reviews to elevate the analysis from granular code details to a broader architectural perspective.

> *“ Doing a Code Review, you often end up diving deep into some functions. You can use these as cues to take a step back and get a higher-level perspective, I think. I believe it can be used for that. So we get a more helicopter view of what we're doing. ”*
>
> *– Bill*

This, he suggests, would enhance the code review process by preventing reviewers from getting bogged down in minute details.

4.3.3 Resource Management

The Cloud Developer Steve with more than 20 years of experience, highlights the perceived lack of value as a key barrier to adopting *Smell of the Week* as a regular practice. He suggests that the time invested in identifying and addressing smells may not be perceived as generating sufficient value for stakeholders, potentially leading to resistance from management.

> *“ Because it costs too much. And then we become unpopular with the bosses. [...] It also tends to become an academic exercise. Where the value does not outweigh the time spent on it. ”*
>
> *– Steve*

Bill offers a counterpoint, proposing that the *Smell of the Week* activity could serve as a valuable tool for identifying areas of interest, with regard to refactoring efforts:

> *“ So it can be used to figure out, okay, where should we intervene? ”*
>
> *– Bill*

He suggests that regular smell detection could inform prioritization and resource allocation for architectural improvements.

4.3.4 Quality Attributes

The team acknowledged that while their team lead might be receptive to the idea of allocating resources for debt management, broader stakeholder understanding and support posed a challenge. As they expressed, stakeholders might struggle to grasp the long-term benefits of AD management and its impact on the system.

Elon further elaborated on this perception gap, highlighting that stakeholders' priorities tend to shift towards maintenance concerns only when development grinds to a halt:

> *“ So when it gets so bad that it becomes unmaintainable, then people start to be receptive to it. But when it's just something like... Ah, it might take maybe 40% less time to do things in the future if we did this. And it would be much nicer to work with, it's a bit hard to sell. ”*
>
> *– Elon*

He emphasizes that stringent requirements, such as 99.99 % uptime, would create a strong incentive to proactively address AD.

> *“ If the requirement was for 99.99% uptime, and we needed to be able to deliver such and such and respond to request-service and so on, then there was a business case for getting these things in order. But when we*

> *don't actually have very stringent uptime requirements and such, then it's more of a wish for us perhaps, that it would be nicer, that it would be easier to maintain this.* ”
>
> – *Elon*

However, the current lack of such performance requirements results in stakeholders tolerating system failures, provided they are resolved within a reasonable time.

4.3.5 Shared Vocabulary

During the reviews and last feedback session, we observed that they started using different smells during our discussions. While team members expressed reservations about whether their architectural competencies have improved, they acknowledged that this practice had expanded their vocabulary for articulating architectural concerns.

Alan highlights the effectiveness of the *smell* metaphor for communicating architectural issues.

> “ *Yeah, I actually think it's a good metaphor. Also because we are used to using the metaphor of code smell from a static analysis tool we use, which can come and tell us something like, hey, it looks like there's a code smell here. There are a lot of code smells, they are probably a bit more code-centric than architects.* ”
>
> – *Alan*

He notes that the team's existing familiarity with the concept of code smells facilitated their adoption and understanding of architectural smells.

4.4 Lessons Learned

As highlighted in Section 2, the successful implementation of an evolving architecture, capable of adapting to changing requirements, necessitates skilled developers at levels two and three, as defined by Boehm [7]. These developers possess the expertise to navigate unforeseen challenges and construct systems that accommodate change.

A unified team can effectively challenge unfeasible stakeholder requirements, leading to enhanced requirement quality and streamlined management of the evolving architecture. Furthermore, team alignment enables the distribution of architectural decision-making, empowering individual developers with the necessary skills to make informed choices, outlined in Lesson 1:

Lesson 1 An evolving architecture distributes architectural decision-making, necessitating aligned and competent developers.

As discussed in Section 4.3.1, the final feedback session of Cycle 2 established a classification of architectural smells into two categories, as articulated in Lesson 2. This categorization differentiates between *Fix-it-Once* smells, which only require periodic reevaluation, and *Sneaky* smells, which necessitate ongoing management due to their tendency to gradually sneak into the codebase during its evolution.

Lesson 2 Architectural smells can be categorized as either: *Fix-it-Once* or *Sneaky* smells.

During Cycle 1, we found that, if the team was allocated time for AD management, then *Smell of the Week* would be a useful tool to identify areas of improvement, as captured by Lesson 3. This could be implemented either as part of ongoing development or dedicated refactoring efforts.

Lesson 3 Smell of the week can be utilized for team-based Architectural Debt management.

In the context of ongoing development, the team expressed, in Cycle 2, a desire to allocate a few minutes during their weekly code reviews to discuss a subset of smells, keeping these smells in mind throughout the review process. For dedicated refactoring efforts, the team suggested that dedicated *Smell of the Week* sessions could systematically identify areas for improvement by focusing on one smell at a time.

5 Discussion

In this section, we address the research question as presented in Section 1.

Lessons	Examples	Related Research
Lesson 1: An evolving architecture distributes architectural decision-making, necessitating aligned and competent developers.	During development, a developer faces different architectural decisions, e.g. whether to keep a class or divide it, to eliminate God Objects.	Supports decentralizing architectural responsibilities [16]. And extends agile methods, with an increased focus on architecture [4, 46].
Lesson 2: Architectural smells can be categorized as either: *Fix-it-Once* or *Sneaky* smells.	A Golden Hammer can be resolved once and not need attention for a long time, whereas God Objects sneak into the codebase over time.	We expand on the existing categorization of architectural smells [2] by introducing an additional temporal classification.
Lesson 3: Smell of the week can be utilized for team-based Architectural Debt management.	When a developer wants to improve the architecture, looking for God Objects will help guide their refactoring efforts.	Supports that smells can be used to identify refactoring opportunities [40].

Table 1: Key lessons of the study

RQ How can architectural practices be improved in teams that develop and maintain evolving systems with a high risk of accumulating Architectural Debt?

We first outline our contributions to the literature both in theory and practice. Additionally, we address the potential limitations of the project and suggest ways to mitigate them. Finally, we conclude by discussing areas of improvement for future research.

5.1 Contribution

Section 4.4 present our primary findings, which are discussed below. Table 1 summarizes these lessons and provides an example for each.

The evolvable architecture outlined in Section 2.1 necessitates skilled developers, as described in Section 2.2, capable of adapting to changing requirements. Lesson 1 encapsulates this, extending the concept of shared code ownership in Extreme Programming [46] to include shared architectural responsibility, aligning with decentralized decision-making proposed by [16].

Existing classifications of architectural smells are categorised based on their impact on modularity, hierarchy, and dependency structures [2] or components and non-functional requirements [31]. Lesson 2 introduces a temporal categorization, differentiating between *Sneaky* and *Fix-it-Once* smells. This distinction has practical implications for prioritization and scheduling of remediation efforts: *Sneaky* smells require ongoing vigilance, while *Fix-it-Once* smells can be addressed early or periodically in a project. The Too Many Standards, for instance, a *Fix-it-Once* smell, can be resolved early on for known technology stacks while addressing the issue in a prototype can be postponed to a later time.

Lesson 3 underscores the utility of combining *Smell of the Week* with architectural smells to facilitate architectural improvement. Smells act as indicators of potential issues [2, 19], and different patterns aid in identifying refactoring opportunities [40] and motivate architectural improvements. Regular practice expands architectural vocabulary and enhances recall through spaced repetition [9], ultimately improving team effectiveness.

This *Smell of the Week* practice, introduced as a recurring activity, where a subset of architectural smells selected from a predefined list Appendix A are discussed, followed by code analysis to identify their

presence. This adaptable approach can be adjusted to fit various contexts, with time allocation dependent on available resources.

Our objective was to enhance the team's architectural vocabulary and competencies, as written in Section 4.1.5. We observed the development of a shared vocabulary, facilitating discussions about AD and solutions, as well as increased alignment on the system's architectural deficiencies. These observations suggest improved competencies, despite the team's self-reported lack of significant change.

However, concerns about the practice's potentially demotivating effects, particularly when addressing issues without the necessary resources for resolution, were raised. This is likely influenced by the organizational context at ITS, where non-functional requirements are not prioritized, hindering support for AD management. The team's perceived lack of value suggests that a different organizational environment, one that prioritizes such requirements, might perceive the benefits of the practice differently.

5.2 Limitations

As earlier mentioned one of the authors' employment at ITS introduces potential bias, stemming from implicit knowledge about the organization, potentially leading to unconscious projections onto the studied team. Additionally, the AR methodology employed in this study, while valuable for its dual focus on aiding practitioners and contributing to academia [1, 30], is inherently susceptible to researcher bias during result interpretation [48]. To mitigate this, we have ensured transparency by explicitly stating the rationale behind our conclusions and using direct quotes where possible.

To distinguish this work from mere consultancy [1, 30], we highlight the dual purpose of AR: providing practical solutions for the ITS team while contributing to broader research on architectural debt management. Our three lessons learned demonstrate this commitment to generating new knowledge that can benefit both practitioners and the research community. Finally, we argue that the *Smell of the Week* intervention is transferable as it could be beneficial in other organizations due to the widespread nature of suboptimal architecture [20] and the intervention's relative ease of implementation.

5.3 Future Research

The team's interest in incorporating the *Smell of the Week* practice as a minor part of their existing code reviews warrants further investigation as a logical next step in evaluating the practice's effectiveness.

The practice was identified to help systematically explore various system components and could facilitate knowledge transfer when a new team takes over a project. A focused smell identification session could prompt original developers to share their design rationale and help the new team quickly understand the existing architecture, including any intentional smells.

Given the inverse relationship between team size and agility [7], further research could investigate the practice's applicability in larger organizations. We propose creating smaller review groups across teams to facilitate knowledge sharing and inter-team communication. Conversely, studying the practice in small, agile organizations could illuminate its effectiveness in environments where rapid decision-making and close collaboration are possible.

6 Conclusion

Our AR study investigated how a small IT team within a Danish university could improve its architectural practices. Over eight months, we conducted interviews, participant observations, and implemented the *Smell of the Week* activity, focusing on its integration into existing code reviews. This practice facilitated productive discussions on architectural issues and fostered a shared vocabulary for addressing architectural debt.

Key contributions include recognizing the need for aligned and competent developers in evolving architectures, where the architect role is a shared team responsibility. Additionally, we identified a new categorization of smells: *Sneaky* (requiring ongoing attention) and *Fix-it-Once* (addressable at specific times), extending existing classifications.

Acknowledgements

The authors would like to thank John Stouby Persson for his excellent supervision and insightful guidance throughout this research. We also acknowledge the invaluable collaboration of the ITS team and express our gratitude for their openness and willingness to share their expertise and perspectives, which have been instrumental in shaping this research.

References

[1] Avison, D., Davison, R., Malaurent, J.: Information systems action research: Debunking myths and overcoming barriers. Information & Management **55**(2), 177–187 (2018). https://doi.org/10.1016/j.im.2017.05.004, https://www.sciencedirect.com/science/article/pii/S0378720617300605

[2] Azadi, U., Fontana, F.A., Taibi, D.: Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 88–97 (2019). https://doi.org/10.1109/TechDebt.2019.00027

[3] Bass, L., Clements, P., Kazman, R.: Software Architecture In Practice. Pearson, New York (2013)

[4] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Principles behind the Agile Manifesto. https://agilemanifesto.org/principles.html (2001), accessed: 17-01-2024

[5] Benyon, D.: Designing Interactive Systems. No. ISBN: 978-1 -4479-2011 -3 in Paper, PEARSON (2005)

[6] Besker, T., Martini, A., Bosch, J.: Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software **135**, 1–16 (2018). https://doi.org/10.1016/j.jss.2017.09.025, https://www.sciencedirect.com/science/article/pii/S0164121217302121

[7] Boehm, B., Turner, R.: Observations on balancing discipline and agility. In: Proceedings of the Agile Development Conference, 2003. ADC 2003. pp. 32–39 (2003). https://doi.org/10.1109/ADC.2003.1231450

[8] Borup, N., Christiansen, A., Tovgaard, S., Persson, J.: Deliberative Technical Debt Management: An Action Research Study, pp. 50–65. International Conference on Software Business (11 2021). https://doi.org/10.1007/978-3-030-91983-2_5

[9] Branwen, G.: Spaced repetition. https://gwern.net/spaced-repetition (2019), accessed: 14/5-2024)

[10] Buschmann, F.: Introducing the pragmatic architect. IEEE Software **26**(5), 10–11 (2009). https://doi.org/10.1109/MS.2009.130

[11] Cai, Y., Kazman, R.: Software architecture health monitor. In: 2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE). pp. 18–21 (2016). https://doi.org/10.1145/2896935.2896940

[12] Cai, Y., Xiao, L., Kazman, R., Mo, R., Feng, Q.: Design rule spaces: A new model for representing and analyzing software architecture. IEEE Transactions on Software Engineering **45**(7), 657–682 (2019). https://doi.org/10.1109/TSE.2018.2797899

[13] Canfora, G., Mancini, L., Tortorella, M.: A workbench for program comprehension during software maintenance. In: WPC '96. 4th Workshop on Program Comprehension. pp. 30–39 (1996). https://doi.org/10.1109/WPC.1996.501118

[14] Cunningham, W.: The wycash portfolio management system. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum). p. 29–30. OOPSLA '92, Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/157709.157715, https://doi.org/10.1145/157709.157715

[15] De Lucia, A., Fasolino, A., Munro, M.: Understanding function behaviors through program slicing. In: WPC '96. 4th Workshop on Program Comprehension. pp. 9–18 (1996). https://doi.org/10.1109/WPC.1996.501116

[16] Erder, M., Pureur, P.: What's the architect's role in an agile, cloud-centric world? IEEE Software **33**(5), 30–33 (2016). https://doi.org/10.1109/MS.2016.119

[17] Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I.: Measure it? manage it? ignore it? software practitioners and technical debt. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 50–60. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786848, https://doi-org.zorac.aub.aau.dk/10.1145/2786805.2786848

[18] Fowler, M.: Who needs an architect? Tech. rep., Institute of Electrical and Electronics Engineers (2003)

[19] Fowler, M.: Code Smell. https://martinfowler.com/bliki/CodeSmell.html (2006), accessed on 29/01-2024

[20] Fowler, M.: Is High Quality Software Worth the Cost? (2019), https://martinfowler.com/articles/is-quality-worth-cost.html, accessed on 12/10-2023

[21] Hofmeister, C., Nord, R., Soni, D.: Applied software architecture. Addison-Wesley Longman Publishing Co., Inc., USA (1999)

[22] Kolny, M.: Scaling up the prime video monitoring service and reducing costs by 90%. Tech. rep., TECH (2023)

[23] Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012). https://doi.org/10.1109/MS.2012.167

[24] Leffingwell, D.: Principles of Agile Architecture. https://scalingsoftwareagility.files.wordpress.com/2008/08/principles_agile_architecture.pdf (2006), accessed on 29/01-2024

[25] Li, Z., Liang, P., Avgeriou, P.: Architectural Debt Management in Value-Oriented Architecting (01 2013). https://doi.org/10.1016/B978-0-12-410464-8.00009-X

[26] Li, Z., Liang, P., Avgeriou, P.: Architectural technical debt identification based on architecture decisions and change scenarios. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. pp. 65–74 (2015). https://doi.org/10.1109/WICSA.2015.19

[27] Martini, A., Bosch, J.: The danger of architectural technical debt: Contagious debt and vicious circles. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. pp. 1–10 (2015). https://doi.org/10.1109/WICSA.2015.31

[28] Martini, A., Bosch, J.: An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). pp. 31–40 (2016)

[29] Mathiassen, L., Munk-Madsen, A., Nielsen, P.A., Stage, J.: Object Oriented Analysis & Design. No. ISBN: 978-87-970693-0-1 in Paperback, Metodica ApS (2018)

[30] Mckay, J., Marshall, P.: The dual imperatives of action research. Information Technology and People **14**, 46–59 (02 2001). https://doi.org/10.1108/09593840110384771

[31] Mumtaz, H., Singh, P., Blincoe, K.: A systematic mapping study on architectural smells detection. Journal of Systems and Software **173**, 110885 (2021). https://doi.org/10.1016/j.jss.2020.110885, https://www.sciencedirect.com/science/article/pii/S0164121220302752

[32] Nayebi, M., Cai, Y., Kazman, R., Ruhe, G., Feng, Q., Carlson, C., Chew, F.: A longitudinal study of identifying and paying down architecture debt. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 171–180 (2019). https://doi.org/10.1109/ICSE-SEIP.2019.00026

[33] Oliveira, F., Goldman, A., Santos, V.: Managing technical debt in software projects using scrum: An action research. In: 2015 Agile Conference. pp. 50–59 (2015). https://doi.org/10.1109/Agile.2015.7

[34] Patton, M.Q.: Qualitative Research & Evaluation Methods. Fourth Edition. Sage Publications. Thousands Oaks, California. (2015)

[35] Rehman, I., Mirakhorli, M., Nagappan, M., Aralbay Uulu, A., Thornton, M.: Roles and impacts of hands-on software architects in five industrial case studies. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 117–127 (2018). https://doi.org/10.1145/3180155.3180234

[36] Risager, T., Mohamed, F.O.A.: Mitigating architectural debt in evolving systems: An action research study

[37] Seaman, C., Guo, Y.: Chapter 2 - Measuring and Monitoring Technical Debt (2011). https://doi.org/https://doi.org/10.1016/B978-0-12-385512-1.00002-5, https://www.sciencedirect.com/science/article/pii/B9780123855121000025

[38] Sharpe, D.: Participant Observant. https://research.utoronto.ca/participant-observation, accessed on 21/02-2024

[39] Snipes, W., Karlekar, S., Mo, R.: A case study of the effects of architecture debt on software evolution effort. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 400–403 (2018). https://doi.org/10.1109/SEAA.2018.00071

[40] Sousa, L., Oizumi, W., Garcia, A., Oliveira, A., Cedrim, D., Lucena, C.: When are smells indicators of architectural refactoring opportunities: A study of 50 software projects. In: Proceedings of the 28th International Conference on Program Comprehension. p. 354–365. ICPC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387904.3389276, https://doi.org/10.1145/3387904.3389276

[41] Staron, M.: Action Research in Software Engineering (01 2020). https://doi.org/10.1007/978-3-030-32610-4

[42] Storey, M.A., Fracchia, F., Müller, H.: Cognitive design elements to support the construction of a mental model during software exploration. Journal of Systems and Software **44**(3), 171–185 (1999). https://doi.org/https://doi.org/10.1016/S0164-1212(98)10055-9, https://www.sciencedirect.com/science/article/pii/S0164121298100559

[43] Taibi, D., Janes, A., Lenarduzzi, V.: How developers perceive smells in source code: A replicated study. Information and Software Technology **92**, 223–235 (2017). https://doi.org/10.1016/j.infsof.2017.08.008, https://www.sciencedirect.com/science/article/pii/S0950584916304128

[44] Verdecchia, R., Kruchten, P., Lago, P.: Architectural technical debt: A grounded theory. In: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (eds.) Software Architecture. pp. 202–219. Springer International Publishing, Cham (2020)

[45] Waterman, Michael: Agility, risk, and uncertainty, part 1: Designing an agile architecture. IEEE Software **35**(2), 99–101 (2018). https://doi.org/10.1109/MS.2018.1661335

[46] Wells, D.: Extreme Programming. `http://www.extremeprogramming.org` (2013), accessed on 14-05-2025

[47] Wohlin, C., Runeson, P.: Guiding the selection of research methodology in industry–academia collaboration in software engineering. Tech. rep., Information and Software Technology (2021)

[48] Wynekoop, J.L., Conger, S.A.: A review of computer aided software engineering research methods. The Information Systems Research Area Of The 90's **I**, p. 130–154 (1990)

[49] Xiao, L., Cai, Y., Kazman, R.: Design rule spaces: a new form of architecture insight. International Conference on Software Engineering **36**, 967–977 (2014). https://doi.org/M 978-1-4503-2756-5/14/05

[50] Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q.: Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering. p. 488–498. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884822, `https://doi.org/10.1145/2884781.2884822`

[51] Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q.: Detecting the locations and predicting the maintenance costs of compound architectural debts. IEEE Transactions on Software Engineering **48**(9), 3686–3715 (2022). https://doi.org/10.1109/TSE.2021.3102221

[52] Yli-Huumo, J., Maglyas, A., Smolander, K., Haller, J., Törnroos, H.: Developing processes to increase technical debt visibility and manageability – an action research study in industry. In: Abrahamsson, P., Jedlitschka, A., Nguyen Duc, A., Felderer, M., Amasaki, S., Mikkonen, T. (eds.) Product-Focused Software Process Improvement. pp. 368–378. Springer International Publishing, Cham (2016)

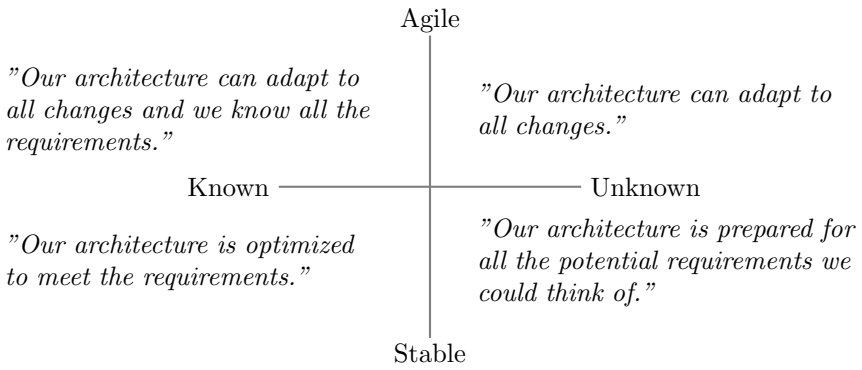

Figure 2: Agile Architecture Quadrant: The x-axis represents known/unknown Architectural Significant Requirements, the y-axis represents the agility of the architecture. (Re-representation from 9th semester [36])

7 Further Findings

This section outlines additional findings that, while relevant to the research, were not explored in depth.

7.1 Skinny Repository

During Cycle 1, the team encountered a specific instance of AR that deviated from the expected pattern. While the Fat Repository smell, characterized by repositories responsible for too many entities, was included in our initial list, the team identified the opposite issue in their codebase: Skinny Repositories, which manage too few entities. To address this, they proposed merging multiple skinny repositories into one, potentially simplifying the system's architecture. This finding presents a novel contribution to the existing body of knowledge on AD and expands the potential range of recognized smells.

We consider this a minor finding that is not directly relevant to the primary research objective of improving the team's architectural practices. The discovery of these additional smells occurred during the *Smell of the Week* reviews and reflects the practitioners' own perception of architectural issues, which did not always align with the predefined lists of smells found in the literature. This observation aligns with previous research highlighting the divergence between smells in theory and smells in practice [43].

7.2 Agile Architecture Quadrant

To analyze system agility, we propose a conceptual framework, depicted in Figure 2, consisting of two axes: system architectural agility and requirements stability. Architectural agility refers to the ability of the architecture to adapt to evolving requirements. Requirements can be classified as either known or unknown, depending on their degree of change over time.

Figure 2 illustrates the relationship between requirement stability and architectural agility. In environments with highly dynamic requirements, a team capable of managing change can adapt the architecture accordingly. Conversely, less agile teams may need to invest in highly flexible architectures upfront to accommodate future changes.

This framework was presented to ITS during the third review of Cycle 1, prompting a discussion about their Gateway system's current and desired position within the quadrant. Interestingly, the team noted that the system's position could vary depending on the level of abstraction considered.

While the team unanimously agreed that practical experience is necessary to fully assess the quadrant's utility, they acknowledged its potential to help identify the frequency and nature of requirement changes, leading to discussions on how to address them, as described by Bill.

> *“ . . . one is somewhat forced to take a position on how the requirements will evolve in the future. [. . .] What do we think will happen? What will they come up with? What might they come up with? And should we think about it now? [. . .] how will we handle requirement changes in the future? ”*
>
> *– Bill*

Ultimately, the team recognized that this framework could assist in managing requirement changes and guiding future system evolution.

As our primary research objective is to enhance practitioners' architectural practices, this finding was not included in the main body of the paper. The proposed framework, while useful for describing a system's nature and requirements, does not directly offer actionable strategies for architectural improvement. Therefore, it was deemed less relevant to our central research focus, which centers on practical techniques that practitioners can employ to enhance their architectural decision-making and processes.

7.3 Impact on Non-Functional Requirements

Following Besker et al.'s [6] unified model for architectural debt, we investigate the negative impact of architectural smells on a system's non-functional requirements, also known as quality attributes. Using the 12 attributes outlined in [29], a systematic evaluation of each smell reveals a predominant impact on comprehensibility and maintainability, with some affecting efficiency, testability, and flexibility.

Interestingly, 40 % of smells impacting comprehensibility also indirectly affected maintainability, a correlation supported by existing research [42, 15, 13]. This suggests that these attributes should be considered correlated rather than distinct. Furthermore, the strong emphasis on comprehensibility underscores its importance in software architecture.

To maximize the value of our limited time with ITS, we sought to prioritize architectural smells relevant to their focus on maintainability. Therefore, we undertook an exercise to classify smells based on their impact on non-functional requirements, anticipating that this would help narrow our selection. Instead, we found that 60 % of the smells had a direct impact on maintainability, 55 % on comprehensibility, and those having an impact on the latter also had an impact on the former. While this analysis provided the relationship between architectural smells and quality attributes, it did not significantly reduce the number of smells to consider for the *Smell of the Week* practice. Because of this and the low relevance to our research goal, as also mentioned in Section 7.2, this finding was removed from the main paper.

8 Additional Areas for Potential Investigation

This section will outline some of the directions we could have taken after investigating the initial problem situation of Cycle 1.

While our primary focus centered on architectural smells and debt, our collaboration with ITS revealed several additional areas warranting further exploration. Maintaining their expanding system portfolio poses

a significant challenge for the small team, particularly due to the lack of uniformity and diverse technology stacks across gateways, leading to knowledge fragmentation and complex maintenance. Cascading changes, where modifications in one area necessitate changes elsewhere, further compound this issue.

Beyond architecture, ITS faces TD manifested in insufficient test coverage, unimplemented requirements, and code complexity with potential code smells. Additionally, organizational misalignment stemming from differing interpretations of master data and technical terms, as well as varying technology choices across teams, creates knowledge silos that impede collaboration. Incorrect domain separation and cyclic dependencies also pose challenges, necessitating careful API versioning and management to avoid livelocks. Finally, infrastructure challenges, such as ensuring system availability while adapting to cloud platform updates and complying with data privacy regulations e.g. GDPR, highlight the need for a flexible and compliant architecture.

9 Interview Guide

The following sections detail the interviews conducted with ITS, outlining the questions posed to provide insights into our research methodology. Initially, given the exploratory nature of the project's early stages, interviews were open-ended to gain familiarity with ITS systems and challenges. As the research progressed and a specific focus emerged, questions became more targeted, aiming to evaluate the effectiveness of our intervention.

9.1 Meeting with Steve

1. Present architecture
   (a) What is your architecture called?
2. Present problems
   (a) Relevance
   (b) Are there other problems
   (c) Ask about azure cloud
3. Ask about hosting (only if there is time)
4. Flexibility, comprehensible, interoperable

9.2 Meeting with Elon

1. Who defines requirements?
2. What do you do when requirements change/new requirements arise?
3. Who creates the design/architecture?
4. How are decisions made?
5. When are decisions made?
6. Architecture vs. Requirements
7. When are decisions made about refactoring?

9.3 Code Reviews

The initial review consisted of a discussion between researchers and developers, aimed at increasing the researchers' understanding of the Gateway system and introducing the concept of architectural smells to the developers.

Subsequent code reviews were conducted by providing developers with a list of selected architectural smells, which they were tasked with identifying and discussing within the system's codebase. The second review contained 15 smells and the third contained a single smell.

9.4 Code Review Feedback

1. Document a single Smell and its solution
2. What is the problem?
3. How would you solve it if you could start over?
4. How/can you get there today?
    (a) Draw the solution (optional)
    (b) Write it down
5. Interest
    (a) Probability
    (b) Amount
    (c) Principal (Cost to fix)
6. Present Agile Architecture Quadrant
    (a) Where are you?
    (b) Where do you want to be?
    (c) Review the usefulness
7. Smell of the week
    (a) Pros/Cons
    (b) Why/Why not
8. Will you use this in the future? If not, how would you handle TD?
9. Agile Architecture Quadrant
    (a) Is it a good way to think about the problem?

9.5 Sprint Reviews

All the sprint reviews followed the same structure. We began by presenting the selected smells and their descriptions. We did not provide our interpretation of the descriptions, which we left for the team to discuss. When they had come to a common understanding of the smell, we went on to analyse the code for it. In some instances, a developer was tasked with documenting these identified smells, utilizing the following reference points:

1. Filename and Line number
2. What is the problem?
3. Why has it been introduced?
4. What is its severity (Low/High)?
5. How do we manage/resolve it?
6. How do we make sure this does not happen again?

During the second and third reviews, by request of the team, we sent the selected smells one week before the review. This ensured that each developer had time to find relevant code snippets for our review. This increased the effectiveness of the review, as we spent less time on analysing the code, and more on discussing it.

9.6 Sprint Review Feedback

1. How much of the code should be reviewed?
2. How many Smells can we look at at a time?
3. What about the smells that are not so relevant (you don't have any of)?
4. How often should the activity be performed?
5. Which smells should be looked at?
6. Has it made you as a team better at working with architecture?
7. What long-term pros/cons can you see with the activity?
8. Are you producing better architecture?
9. Does it help to write your thoughts down?
10. Is this something you would like to use?
11. Why/Why not?
12. Have you changed the way you interact with your stakeholders?
13. Are you more aligned around what can and cannot be done?

14. Do you put the requirements up for discussion?
15. Is AD on the agenda?
16. Is the smell concept a good metaphor for talking about technical debt?
17. Are some smells more relevant than others?
18. How have you looked for smells?
19. How much time have you spent on it?
20. Would you have liked more time?
21. Have you fixed/prioritized any of the shortcomings we found during the reviews?
22. Have you created any sprint items for your backlog?
23. Have you fixed any of them in your sprints?
24. What have you gained from this process?
25. Have you gained a better understanding of TD/AD?

A Architectural Smells

Architectural Smell	Description
Abstraction without decoupling	This smell occurs where a client class uses a service represented as an abstract type, but also a concrete implementation of this service, represented as a non-abstract subtype of the abstract type.
Ambiguous interface	This smell occurs when an abstraction (interface) is over-engineered by adding methods intended to accommodate potential future requirements but never used.
Ambiguous name	This smell occurs when developers use ambiguous or meaningless names for interfaces.
Anchor submission	This smell occurs when each file structurally depends on the anchor file, but each member historically dominates the anchor.
Anchor dominant	This smell occurs when each file structurally depends on the anchor file, and the anchor file historically dominates each member file.
API versioning	This smell occurs when APIs are not semantically versioned.
Architecture violation	This smell occurs when an intended architecture is different from its actual implementation.
Big bang	This smell occurs when an entire system is built at once.
Bottleneck service	This smell occurs when a service is highly used (high incoming and outgoing coupling) by other services.
Bloated service	This smell occurs when a service becomes a blob with one large interface and/or lots of parameters.
Blob or God object/component	This smell occurs when a component implements an excessive number of concerns.
Brain controller	This smell occurs when controllers have too much flow control.
Brain repository	This smell occurs when a complex logic is developed in the repository.
Circuitous treasure hunt	This smell occurs when an object looks in several places to find the information that it needs.
Chatty service	This smell occurs when a service has a high number of connections with other services.
Clique	This smell occurs when a group of files are tightly coupled by dependency cycles.
Co-change coupling	This smell occurs when changes to a component require changes in another component.
Concern overload	This smell occurs when a component implements an excessive number of concerns.
Connector envy	This smell occurs when components cover too much functionality with respect to connections.

Architectural Smell	Description
Crudy interface	This smell occurs when services show an RPC-like behavior by declaring CRUD-type operations.
Crudy URI	This smell occurs when crudy verbs (e.g., create, read, update, or delete) are used in the APIs.
Cyclic dependency	This smell occurs when two or more architecture components depend on each other directly or indirectly.
Cyclic hierarchy	This smell occurs when a direct referencing of a subtype from a supertype is created.
Cycles between namespaces	This smell occurs when two or more namespaces depend on each other directly or indirectly.
Data service	This smell occurs when a service has only accessor operations (getters and setters).
Degenerated inheritance	This smell occurs when there are multiple inheritance paths connecting subtypes with their supertypes or a concrete class with their abstractions (abstract classes or interfaces).
Dense structure	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes.
Duplicated service	This smell occurs when a set of highly similar services exists.
Empty semi-trucks	This smell occurs when an excessive number of requests is required to perform a task.
ESB usage	This smell occurs when micro-services communicate via an ESB (enterprise service bus)—it adds complexities for registering and de-registering services on it.
Excessive dynamic allocation	This smell occurs when an application unnecessarily creates and destroys large numbers of objects during its execution.
Extensive processing	This smell occurs when extensive processing impedes overall response time.
Fat repository	This smell occurs when a repository is managing too many entities.
Feature concentration	This smell occurs when different functionalities are implemented in a single design construct.
Forgetting hypermedia	This smell occurs when there is a lack of hypermedia (i.e., not linking resources).
Golden hammer	This smell occurs when familiar technologies are used as solutions to every problem.
Hard-coded endpoints	This smell occurs when micro-services are connected with hard-coded endpoints, making the change in their locations problematic.

Architectural Smell	Description
Hub-like dependency	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes.
Ignoring MIME types	This smell occurs when resources do not support multiple formats (e.g., XML, JSON, etc.).
Ignoring Caching	This smell occurs when developers avoid to implement the caching capability in the web applications.
Implicit cross-module dependency	This smell occurs when two or more architecture components depend on each other directly or indirectly.
Improper inheritance	This smell occurs when a parent class depends on its derived class or where a client depends on both the parent and derived classes.
Incomplete service	This smell occurs when the client is given the responsibility to complete the service.
Incomplete abstraction	This smell occurs when an abstraction does not support interrelated methods completely.
Interface violation	This smell occurs when components in an architecture communicate without their interfaces.
Knot service	This smell occurs when a set of very low cohesive services are tightly coupled.
Laborious repository method	This smell occurs when a repository method has multiple database actions.
Leaky encapsulation	This smell occurs when a class leaks implementation details because of its public implementation.
Link overload	This smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes.
Low cohesive operations	This smell occurs when developers place very low cohesive operations (not semantically related) in a single portType.
Maybe it is not RPC	This smell occurs when a service mainly provides CRUD-type (create, read, update, and delete) operations.
Meddling service	This smell occurs when services directly query the database.
Micro-service greedy	This smell generates an explosion of the number of micro-services composing a system.
Missing abstraction	This smell occurs when clumps of data are used instead of creating classes or interfaces.
Missing encapsulation	This smell occurs when classes are not encapsulated.
Misplaced component	This smell occurs when an architecture component is placed somewhere else other than the one it was intended for, resulting in undesired dependencies.

Architectural Smell	Description
More is less	This smell occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources.
Modularity violation	This smell occurs when an architecture violates the modularity principles.
Multi-service	This smell occurs when a service implements a multitude of methods related to different abstractions.
Multipath hierarchy	This smell occurs when there are multiple inheritance paths connecting subtypes with their supertypes or a concrete class with their abstractions.
Not having an API gateway	This smell occurs when service-consumers communicate directly with each micro-service.
Nobody home	This smell occurs when a service is defined but never used.
Non-transfer communication	This smell occurs when communication between components is not accomplished using transfer objects.
Nothing new	This smell occurs when inappropriate practices in object-oriented practices are attempted to apply in service-oriented.
No legacy	This smell occurs when a service provides limited standardized support of data types and interactions.
No subsystems	This smell occurs when a system has no subsystems.
One-lane bridge	This smell occurs when only one or a few processes can be executed concurrently.
Overgeneralized subsystems	This smell occurs when the generalization of the subsystems is overdone.
Overstandardized SOA	This smell occurs when all aspects and dimensions of SOA are overstandardized.
Package cycle	This smell occurs when two or more packages depend on each other directly or indirectly.
Package instability	This smell occurs when a package has many dependencies that frequently changes with other packages.
Missing package abstractness	This smell occurs when a package has unnecessary or missing abstraction.
Pipe and filter	This smell occurs when the slowest filter in the architecture results in low throughput.
Promiscuous controller	This smell occurs when controllers are offering too many actions.
Redundant portTypes	This smell occurs when multiple portTypes are duplicated with a similar set of operations.
Sand pile	This smell occurs when a service is composed of multiple smaller services sharing common data.
Scattered functionality	This smell occurs when a high-level concern is realized across multiple components.

Architectural Smell	Description
Security flaws	This smell occurs when critical information is disclosed or tampered, when confidentiality and integrity are not ensured in the architecture.
Separation of concerns	This smell occurs when the responsibilities of the components of an architecture are not appropriately separated.
Service Chain	This smell occurs when consecutive service invocations happen.
Shared libraries	This smell occurs when shared libraries between different micro-services are used.
Shared persistency	This smell occurs when different micro-services access the same relational database, reducing the service independence.
Shiny nickel	This smell occurs due to inflexibility to incorporate new technologies within service architecture.
Silver bullet	This smell occurs when unknown technologies are implemented where they are not required.
Sloppy delegation	This smell occurs when a component delegates the functionality to other components, which should be performed internally by that component.
Speculative hierarchy	This smell occurs when a hierarchy is created speculatively.
Subtype knowledge	This smell occurs when a direct referencing of a subtype from a supertype is created.
Tiny/nano/fine-grained service	This smell occurs when a service has only a few operations.
Ramp	This smell occurs when processing time increases as the system is used.
Too many standards	This smell occurs when different development languages, protocols, frameworks are used in micro-services.
Too small package	This smell occurs when a package has only one or two classes.
Too many subsystems	This smell occurs when a system consists of many subsystems.
Tower of babel	This smell occurs when processes excessively convert, parse, and translate internal data into a common exchange format.
Traffic jam	This smell occurs when one problem causes a backlog of jobs.
Unbalanced processing	This smell occurs when processing cannot make use of available processors.
Unauthorized dependency	This smell occurs when an unauthorized dependency exists between the components.
Unstable dependency	This smell occurs when a component depends on other components that are less stable than itself.
Unused package	This smell occurs when a package is no longer in use.
Unclear package name	This smell occurs when developers use ambiguous or meaningless names for packages.
Unbalanced package hierarchy	This smell occurs when the package structure is unbalanced.
Unauthorized call	This smell occurs when a calling component is not connected to the called component.

Architectural Smell	**Description**
Undercover transfer object	This smell occurs when transfer objects serve as data containers for the communication between components.
Unhealthy inheritance hierarchy	This smell occurs when a direct referencing of a subtype from a supertype is created.
Unstable interface	This smell occurs when an interface depends on other interfaces that are less stable than itself.
Unused interface	This smell occurs when an abstraction (interface) is over-engineered by adding methods intended to accommodate potential future requirements but never used.
Unutilized abstraction	This smell occurs when a direct referencing of a concrete class is created, instead of referencing one of its supertypes, from an abstract class.
Unnecessary hierarchy	This smell occurs when the inheritance hierarchy is unnecessarily created.
Wrong cuts	This smell occurs when micro-services are split based on technical layers instead of business capabilities .

Table 2: This table lists 108 architecture smells and is a re-presentation of table A.1 from [31]