



On the correlation between architectural smells and static analysis warnings

Matteo Esposito¹ · Mikel Robredo¹ · Francesca Arcelli Fontana² ·
Valentina Lenarduzzi¹

Received: 11 February 2025 / Accepted: 1 September 2025 / Published online: 13 October 2025
© The Author(s) 2025

Abstract

Software quality assurance is essential during software development and maintenance. Static Analysis Tools (SAT) are widely used for assessing code quality. Architectural smells are becoming more daunting to address and evaluate among quality issues. We aim to understand the relationships between Static Analysis Warnings (“warnings”) and Architectural Smells (“smell”) to guide developers/maintainers in focusing their effort on warnings more prone to co-occurring with smell. We performed an empirical study on 103 Java projects totaling 72 million LOC belonging to projects from a vast set of domains, and 785 warnings were detected by three SAT, Checkstyle, Findbugs, PMD, SonarQube, and 4 architectural smells were detected by the ARCAN tool. We analyzed how warnings influence smell presence. Finally, we proposed a smell remediation effort prioritization based on warning severity and warning proneness to specific smells. Our study reveals a moderate correlation between warnings and smells. Different combinations of SATs and warnings significantly affect smell occurrence, with certain warnings more Likely to co-occur with specific smells. Conversely, 33.79% of warnings are “non-co-occurring” with any of the smells in our dataset. This provides an early indicator for potential architectural concerns before resource-intensive architectural analysis is performed. Practitioners can ignore about a third of warnings and focus on those most likely to be associated with smells. Prioritizing smell remediation based on warning severity or warning proneness to specific smells results in effective rankings like those based on smell severity. While not a substitute for specialized tools like ARCAN, warning-based prioritization provides a pragmatic bridge between low-level warnings and high-level architectural issues, particularly useful in contexts lacking full architectural visibility.

Keywords Static analysis tool · Software quality warnings · Architectural smells · Technical debt

1 Introduction

The risk of architectural erosion (Wan et al., 2023; Perry & Wolf, 1992) and disruption for critical services is exponentially increasing due to the pervasive nature of computing and the widespread adoption of digital services in all areas of daily life (Esposito & Falessi, 2023). To create a common definition for quality issues, the literature provides the concept of *smells* (Lenarduzzi et al., 2019b), referring to quality issues in code (Taibi et al., 2017). These smells are usually categorized as code smells, low-level coding issues (Fowler & Back, 1999; Lenarduzzi et al., 2019b), and, recently, architectural smells, higher-level abstract issues that concern the software architecture and the software components (Arcelli Fontana et al., 2017).

A popular technique to detect quality issues is to leverage Static Analysis Tools (SAT) (Vassallo et al., 2019; Avgeriou et al., 2021) which analyze the source code without executing it (thus static analyzers), and produce a report containing all the violations, i.e., Static Analysis Warnings (from now on “warnings”) that the code presents. More specifically, warnings refer to code violations with a lower-level granularity compared to smells. Therefore, warnings do not consider the software architecture of the system under development, commonly focusing on class or method levels (Esposito et al., 2024).

Despite the large body of work on smells detection techniques using architectural reconstruction, graph-based models, or clustering heuristics (Mumtaz et al., 2021a), these methods are often resource-intensive, demand explicit architectural views, or require specialized tooling that may not be readily integrated into developers’ daily workflows.

In contrast, warnings are pervasive in software engineering practice. They are automatically produced, tightly coupled with development environments, and consulted regularly by practitioners. This makes warnings a pragmatic and low-barrier data source, yet their potential to act as early indicators of architectural erosion remains largely underexplored.

While multiple studies successfully exploited SATs to remediate common quality issues (Esposito et al., 2024; Palomba et al., 2016; Taibi et al., 2017; Lenarduzzi et al., 2020c; Arcelli Fontana & Zanoni, 2011), only a few studies focused on developing SATs to detect smells (Arcelli Fontana et al., 2017). Moreover, some studies have investigated the interrelations between code anomalies (Macia et al., 2012b, a) or code smells (Fontana et al., 2019) and smells (Macia et al., 2012b, a). However, to our knowledge, no previous study has analyzed the correlations between warnings and smells.

Furthermore, to ensure the safety and stability of services, during development and maintenance, smell remediation should be prioritized (Nord et al., 2012; Rachow & Riebisch, 2022). Hence, the portion of code affected by smells is the primary remediation target to prevent critical quality issues.

This work investigates whether warnings, widely available and frequently ignored, can serve as useful signals to identify or prioritize architectural remediation efforts. By doing so, we aim to bridge the gap between commonly available code-level signals and hard-to-detect architectural issues, providing practical insights for both researchers and practitioners.

Therefore, the need to investigate how practitioners and researchers can exploit warnings to detect and remediate smells is evident. Hence, we designed and performed a large empirical study on the possible correlations between four smells and warnings detected by three tools largely adopted by developers (Esposito et al., 2024).

Our study assesses the correlation between warning and higher-level smells. We did not consider correlations between defects and warnings, smells, these have been extensively studied in the literature. Specifically, we investigate the correlation between warning and smells and how warning can prioritize smells.

Past studies have struggled to show an obvious relationship between warnings and smells (Pigazzini et al., 2021). Nonetheless, discovering a correlation between smells and warnings would enable the use of existing SAT smells as an effective tool for detecting issues in software architecture. Importantly, this approach requires no additional tooling or modeling effort, making it highly attractive in industrial contexts. For practitioners, linking warning to architectural smells can reduce the need for refactoring, thereby saving both cost and effort (Desai & Parmar, 2016; Deeb et al., 2021).

Therefore, we can summarize our main contributions as follows:

- To our knowledge, we performed the first large-scale study, i.e., 103 projects, on the correlation of smells and warnings. Other studies have considered correlations with more general architectural problems (Macia et al., 2012a, b) or the correlation between smells and code smells (Fontana et al., 2019).
- We investigated the contribution of the most correlated warnings with smells.
- We proposed and evaluated smells remediation effort prioritization via warnings severity.

Our study focused on 103 Java projects from the Qualitas Corpus dataset (QCD) (Tempero et al., 2010), totaling 72 million Lines of code analysis and 66,036 packages, 808,208 classes, and 88,460 interfaces examined across various contexts, including development platforms, multimedia, gaming, middleware, compilers, and testing frameworks. We considered a large set of warnings (1142) and four smells based on dependency issues that can critically influence the quality of a software project and its progressive architecture erosion (Fontana et al., 2016; Li et al., 2021; Mumtaz et al., 2021b).

Contrary to past studies, our findings reveal a moderate, yet statistically significant, correlation between warnings and smells. Moreover, different combinations of SATs and warnings significantly influence the occurrence of smells. Therefore, specific warnings are more prone to co-occur with specific smells. Conversely, we also identified that warnings non-co-occurring with any smells represent 33.79% of our dataset. Our study shows that practitioners can safely drop a third of the warnings in our context and focus on the most smell-prone areas to address smells without prior smell severity or discovery knowledge effectively. Our study highlighted that prioritizing smell remediation efforts based on warning severity, or warning proneness to specific smells with a limited inspection window, results in rankings similar to the optimal one, i.e., ordered by smell severity. Therefore, we suggest that warnings can moderately influence the presence of smells. Prioritizing warnings based on occurrence probability effectively guides remediation. Future research should improve the detection of smell-prone warnings and explore deeper connections between warnings and smells for better understanding and prioritization of remediation efforts.

Paper Structure Section 2 describes the background to which our paper is based. In Section 3, we present the study design. In Section 4, we show the results obtained and discuss them in Section 5. Section 6 focuses on threats to the validity of our study. Section 7

discusses related work and in Section 8, we draw conclusions and outline future research directions.

Definitions In the remainder of this paper, we refer to static analysis warnings as ‘warnings’ and architectural smells as ‘smells’.

2 Background

This section introduces the static analysis tools used, the types of architectural smells considered, and the effort-aware metrics applied to evaluate prioritization.

2.1 Static analysis tools and warnings

This section introduces the tools we employed in our study and the motivation for their selection. SAT analyzes software without running it by inspecting the source code, thus discovering potential quality issues in the code base (Ernst et al., 2015). Recently, the readily availability of SATs and the subsequent inclusion in Continuous Improvement (CI) practices (Zampetti et al., 2017) are boosting their popularity. Among the available SATs, we selected the most widely used according to state of the art in detecting warning (Vassallo et al., 2019; Esposito et al., 2024) such as smells, Sonarqube, Checkstyle, Findbugs, and PMD. According to architectural issues, our study focuses on architectural smells. Therefore, we also included ARCAN based on our previous experience (Arcelli Fontana et al., 2017). In Table 1, we summarize the characteristics of each of the following SATs.

More specifically, regarding SATs, **Checkstyle**¹ is designed to assess Java code quality. The tool uses “rules” according to a set of *checks* for analyzing the code base. Checkstyle has two predefined rule configurations: Google Java Style and Sun Java Style. Static violations of the checks are grouped under two severity levels: error and warning.

FindBugs² examine Java bytecode. The tool detects *bug patterns* which are caused by, but not limited to, difficult language features and misunderstood API features. Bug patterns are ranked on an ordinal scale from one to 20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group.

PMD³ can inspect eight programming languages, including Java. PMD uses a set of *rules* to assess code quality. There are a total of 33 different rule-set configurations for Java projects. Rules are classified into 8 categories. The tool measures rule violations on an ordinal scale of one to five. One is associated with the most severe violations, and 5 is the least severe.

SonarQube⁴ is one of the most common SATs for detecting code quality issues. SonarQube measures several aspects of the source code, such as smells, several lines of code and

¹<https://checkstyle.org>

²<http://findbugs.sourceforge.net>

³<https://pmd.github.io/>

⁴<http://www.sonarsource.org/>

Table 1 Summary of static analysis tools warning (warnings)

SAT	Checkstyle	Findbugs	PMD	SonarQube
Warnings Types	14 types	9 types	8 types	3 types
Total Warnings	173	424	305	413
Types	Annotations, Block Checks, Class Design, Coding, Headers, Imports, Javac Comments, Metrics, Miscellaneous, Modifiers, Naming Conventions, Regexp, Size Violations, Whitespace	Bad practice, Correctness, Experimental, Internationalization, Malicious code vulnerability, Multithreaded correctness, Performance, Security, Dodgy code	Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance, Security	Bugs, Code Smells, Vulnerabilities
Severity Levels	Error, Ignore, Info, Warning	1-20 ranking: Scariest, Scary, Troubling, Concern	Change required, Change suggested, Change optional, Change highly recommended, Change highly optional	Blocker, Critical, Major, Minor, Info

code complexity, and verifies code compliance with a specific set of *coding rules* defined for most common programming languages. Violations of coding rules are reported smells *issues* in SonarQube. The estimated time to remove these issues, i.e., remediation effort, is used to compute the remediation cost. SonarQube analysis encompasses quality aspects such smells reliability, maintainability, and security. Reliability rules, also named *Bugs*, create quality issues that “represent something wrong in the code” and that will soon be reflected in a bug. *Code smells* are considered “maintainability-related issues” in the code that decrease code readability and modifiability. It is important to note that the term “code smells” adopted in SonarQube does not refer to the commonly known code smells term defined by Fowler and Back (1999), but to a different set of rules (Table 2).

2.2 Architectural smells

On the other hand, we employed **ARCAN**⁵ that detects different smells based on dependency issues by computing different metrics proposed by Martin (1995), such smells those related to instability issues. We refer to software instability smells the inability to make changes without impacting the entire system or a large part.

The tool relies on graph database technology to perform graph queries, which enables higher scalability during the detection process and management of many dependencies. The detailed detection techniques for smells have been described in previous works (Arcelli Fontana et al., 2017). Moreover, the tool allows the computation of an architectural debt (Roveda et al., 2018) index based on the number of smells detected in a project and the criticality of the smells, evaluated according to each type of smell. The tool’s results were validated on ten open source systems and two industry projects with a high precision value of 100% in the results and 63% recall (Arcelli Fontana et al., 2017). Moreover, the results of ARCAN were validated using practitioners’ feedback on four industry projects (Martini et al., 2018).

⁵<https://www.arcan.tech>

Table 2 Static analysis warnings

Tool	Detection Capability		
	# rule	# type	# severity
Checkstyle	173	14	4
Findbugs	424	9	4
PMD	305	8	5
SonarQube	413	3	5

A research version of the ARCAN tool has been used for this study, while the latest Arcan commercial version is available at ⁶.

In our study, we have considered the following smells: **Unstable Dependency (UD)** An *Unstable Dependency* occurs when a stable component depends on a less stable component. Martin (2007). This can lead to a ripple effect when the unstable component changes. Arcelli Fontana et al. (2017). Detected in packages. **Example:** A stable `PaymentProcessor` component depending on an unstable `BetaFeature`.

```

1  class BetaFeature {
2      public void experimentalMethod() { /* Unstable feature */ }
3  }
4
5  class PaymentProcessor {
6      private BetaFeature feature; // Depends on unstable module
7
8      public PaymentProcessor(BetaFeature feature) {
9          this.feature = feature;
10     }
11 }

```

Hub-Like Dependency (HD) A *Hub-Like Dependency* occurs when an abstraction has many incoming and outgoing dependencies, making it a critical component that is difficult to modify. Suryanarayana et al. (2015). Detected in classes and packages.

Example A `DatabaseHelper` class used across multiple services.

```

1  class DatabaseHelper {
2      public void connect() { /* Connect to DB */ }
3      public void executeQuery(String query) { /* Run Query */ }
4  }
5
6  class UserService {
7      private DatabaseHelper dbHelper = new DatabaseHelper();
8  }
9
10 class ProductService {
11     private DatabaseHelper dbHelper = new DatabaseHelper();
12 }

```

Cyclic Dependency (CD) A *Cyclic Dependency* occurs when multiple components depend on each other in a cycle, making independent modifications and releases difficult. The

⁶<https://www.arcan.tech/>

subsystems involved in a dependency cycle can hardly be released, maintained, or reused in isolation. Detected in classes and packages. The *Cyclic Dependency* smells is detected according to different shapes (Al-Mutawa et al., 2014) smells described in Arcelli Fontana et al. (2017). **Example A** dependency cycle between A, B, and C.

```

1  class A {
2      private B b;
3  }
4
5  class B {
6      private C c;
7  }
8
9  class C {
10     private A a; // Cycle formed: A -> B -> C -> A
11 }

```

We focused on the above smells based on dependency issues since dependencies are very important in software architecture. Components with many dependencies can be considered more critical and expensive to maintain. Moreover, we defined smells “non-co-occurring” the instances without smells, i.e., a software entity affected by one or more warnings but without any of the ASs considered in our study.

2.3 Effort aware metrics

Çarka et al. (2022) investigated effort-aware metrics (EAM) to improve defect prioritization. In our study, we take inspiration from the definition of PofB and PopT to create three different prioritization rankers for smells. We then evaluate our ranker in a way that is similar to the interpretation of the PofB results. Therefore, we briefly discuss EAMs; according to Çarka et al. (2022), two different EAMs exist: normalized by size or not by size. The most known not-normalized EAM is called *PofB* (Chen et al., 2017; Wang et al., 2020; Xia et al., 2016; Tu et al., 2020) PofB, or Proportion of Defects Identified by the top x% of the codebase, is a measure derived from a prediction model’s rankings. It signifies the percentage of defective entities discovered when analyzing the initial x% of the code. Higher PofB indicates more effective rankings, thereby enhancing testing support. For instance, a PofB 10 of 30% implies that 30% of defective entities were detected by analyzing 10% of the codebase according to the method’s ranking.

Comparing a prediction model’s ranking with a perfect one sheds light on its performance. Mende and Koschke (2009), drawing inspiration from Arisholm et al. (2007), introduced “PopT” to gauge this accuracy. It assesses how the prediction model deviates from perfection while outperforming random rankings. PopT quantifies the area Δ_{opt} between the ideal and predicted models. In the optimal scenario, instances are ordered by decreasing fault density, while in the predicted model, they’re ordered by decreasing predicted defectiveness. The equation for PopT, $1 - \Delta_{opt}$, indicates a larger value denotes a smaller gap between the ideal and predicted models (Yu et al., 2019).

PopT and PofB are distinct metrics, each capturing different facets of model accuracy. They employ different ranking methods: PopT ranks entities based on bug density (bug probability divided by entity size), while PofB ranks them according to bug probability. Consequently, classifiers ranked by PopT and PofB may differ. PopT, grounded in bug density,

offers a more realistic perspective than PofB, which relies solely on probability. However, interpreting Popt is more challenging; a classifier with double the Popt value doesn't necessarily yield double the benefits to the user.

3 Empirical study design

This section presents the empirical study's goal, research questions, metrics, and hypotheses. Moreover, we describe the study context, the data collection, and the data analysis procedures. We designed and conducted our empirical study according to the guidelines defined by Wohlin et al. (2012).

3.1 Goal, research questions, metrics, and hypothesis

We formalized the **goal** of this study according to the Goal Question Metric (GQM) approach (Basili et al., 1994) as follows:

The *goal* of our empirical study is to investigate the relationship between architectural smells and static analysis warnings in the context of software quality assurance. Our *perspective* is of practitioners and researchers seeking to assess the relationship between high-level architectural issues and low-level code quality problems.

Based on our goal, we formulate the following Research Questions (RQ_s)

RQ₁.

Are warnings and smells correlated?

Investigating the correlation between smells and warnings enables us to assess the relationship between high-level architectural issues and low-level code quality problems. We aim to determine whether smells are correlated with warnings and the strength of this correlation. To address RQ₁, we collected warning and smell presence per software package. To assess statistical significance, we conjectured a family of hypotheses, one for each warning-smells pair **hypothesis** (H_1) as follows:

$H_{11(w,s)}$ *There is a correlation between the number of instances of warning w and the number of instances of architectural smell s , averaged at the package level.*

Hence, we defined the **null hypothesis** (H_0) as follows:

$H_{01(w,s)}$ *There is no correlation between the number of instances of warning w and the number of instances of architectural smell s , averaged at the package level.*

Moreover, different static analyzer warnings can lead to specific architectural smells. Therefore, assessing whether a specific warning's presence is a possible indicator of the presence of specific smells is essential. Therefore, we ask:

RQ₂.

Do specific types of warnings co-occur with specific types of smells?

In the long run, architectural smell can silently deteriorate complex software architecture. Hence, we investigate the characteristics of warnings affecting software projects and leading to smells to identify common traits among projects prone to smells and assess further factors contributing to the emergence and persistence of smells. To address RQ₂, we analyzed the co-occurrence of warnings and smells. Let σ be a warning and α be a smell, and η be a software entity. We state that σ co-occur with α , if given η such that σ and α affect contemporary η . To assess statistically significance, we conjectured two **hypotheses** (H_2 and H_3) as follows:

$H_{12(w,s)}$ *There is a difference in the co-occurrence of warning w with smell s compared to its co-occurrence with other smells.*

$H_{13(w,s)}$ *There is a difference in the co-occurrence of warning w with smell s compared to its co-occurrence with other smells by SAT.*

Hence, we defined the **null hypotheses** (H_0) as follows:

$H_{02(w,s)}$ *There is no difference in the co-occurrence of warning w with smell s compared to its co-occurrence with other smells.*

$H_{03(w,s)}$ *There is no difference in the co-occurrence of warning w with smell s compared to its co-occurrence with other smells by SAT.*

We computed the co-occurrence of each warning and smell. For each project package, we counted the number of non-co-occurring warnings (**NCO**) and those affected by CDs, UD_s, and HL_s (and combinations of them). We tested H_2 and H_3 with the Wilcoxon Paired Signed Rank Test (WPT) (Section 3.4).

Finally, assessing the correlation between warnings and smells and understanding how this relationship influences the presence of smells in software projects merely scratches the surface. Indeed, it falls short, considering practitioners' limited time for remediation efforts. Therefore, prioritizing these efforts becomes paramount. Thus, we ask:

RQ₃

Are warnings effective for smell remediation effort prioritization?

Effort prioritization is essential to avoid wasting resources on noncritical tasks (Jorgensen & Shepperd, 2006). No prior study has investigated smell prioritization via warning severity. According to the “no free lunch theorem”, currently, there is no default severity for code quality checks universally accepted by our community (Esposito et al., 2023). Therefore, in our study, we investigated exploiting warning and warning attributes detected from SAT, such as smells SonarQube, i.e., rule severity and cost estimation, and the prioritization effort for smell severity according to Arcelli Fontana et al. (2017). We referred to recent empirical studies to assess the accepted severity of smells (Fontana et al., 2020; Pigazzini et al., 2021; Sas & Avgeriou, 2023). More specifically, Sas and Avgeriou (2023) proposes, on average, an ordinal scale of one to ten, where one is the least severe smell and ten is the highest. Therefore, we adopt the average smell severity values, CD (5), UD (7), and HL (9), smells proposed by Sas and Avgeriou (2023) who empirically derived these ratings based on practitioner feedback and analysis of the impact of smells on maintainability and evolution. Moreover, regarding warning severity, we compiled a comprehensive dataset in our previous study (Lenarduzzi et al., 2019b) focused on SonarQube. Without the risk of a generalizability issue (Esposito et al., 2023), we focused this RQ on SonarQube-specific

warnings. To address RQ₃, we quantify the **probability of a warning inducing smells** (\mathcal{P}) smells the relative frequency at which the particular warning contributed to the occurrence of smells. To assess statistical significance, we conjectured one **hypothesis** (H_4) as follows:

H_{14} *There is a difference in the prioritized effort between the three rankers.*

Hence, we defined the **null hypothesis** (H_0) as follows:

H_{04} *There is no difference in the prioritized effort between the three rankers*

We computed \mathcal{P} for each SAT, warning, and smell. We ranked the warnings based on their severity (Lenarduzzi et al., 2019b), \mathcal{P} , and smells severity (Sas & Avgeriou, 2023). We tested H_4 with WPT (see Section 3.4).

3.2 Study context

We selected projects from the Qualitas Corpus (QCD) collection of software systems (Terra et al., 2013), using the compiled version of the QCD (Tempero et al., 2010). Table 3 details metrics on the QCD.

3.3 Study setup and data collection

This section presents our data collection methodology. Figure 1 presents the study workflow. Our data collection comprised three steps:

1. We analyzed the QCD with three SATs and averaged the results for project packages;
2. We separately analyzed QCD with the Arcan tool;
3. To get our study results, we analyzed the finalized dataset, comprising all warning detected by the three SATs and the smells detected by the Arcan tool grouped by software package (see Section 3.4).

We analyzed the entire data set (RepoCount Java projects) from the Qualitas Corpus (Terra et al., 2013).

Table 3 Summary of qualitas corpus metrics

Metric	Value
Number of Projects	112
Lines of Code (LOC)	>18 million
Packages	16,509
Classes Analyzed	202,052
Interfaces Analyzed	22,115
Methods Analyzed	464,893
Contexts Covered	IDEs, SDKs, databases, 3D/graphics/ media, diagram/visualization libraries and tools, games, middlewares, parsers/generators/make tools, programming language compilers, testing libraries, and tools.
Available Metrics	LOC, NOP, NOCL, NOI, NOM, NOA, NORM, PAR, NSM, NSA. CK Metrics: WMC, DIT, NOC, LCOM HS. - MLOC, SIX, VG, NBD, RMD. - CA, CE, I, A.



Fig. 1 Data collection workflow

Table 4 Spearman’s ρ interpretation (RQ₁)

Interpretation	Perfect	Strong			Moderate			Weak		Zero	
Correlation	+ 1	+ 0.9	+ 0.8	+ 0.7	+ 0.6	+ 0.5	+ 0.4	+ 0.3	+ 0.2	+ 0.1	0
	- 1	- 0.9	- 0.8	- 0.7	- 0.6	- 0.5	- 0.4	- 0.3	- 0.2	- 0.1	0

3.4 Data analysis

We considered the packages affected by at least a smell and one static analysis warning at the class or package levels. One or more static analysis warnings could affect the same set of packages for each smell. This is because an smells may involve more than one class, while static analysis warnings involve only one class. In the case of projects not infected by static analysis warnings or smells, we did not consider them for the analysis.

To answer **RQ₁**, we collected the QCD projects and ran the SAT and ARCAN. We merged the outputs of the tool. In our context, the dependent variable is the smell, and the independent variables are the warning. To compute the correlation coefficients, we need to assess the distributions of our variable. Therefore we conjectured one **hypothesis** ($H_{\mathcal{N}}$) as follows:

$H_{1\mathcal{N}}$ *The collected warning and smells metrics do not follow a normal distribution.*

Hence, we defined the **null hypothesis** (H_0) as follows:

$H_{0\mathcal{N}}$ *The collected warning and smells metrics follow a normal distribution.*

We tested for normality ($H_{\mathcal{N}}$) using the Anderson–Darling (AD) test (Stephens, 1974), considered one of the most powerful methods for detecting deviations from normality (Stephens, 2017). Since normality could not be assumed, we used Spearman’s ρ (Spearman, 1904) instead of Pearson’s correlation coefficient (Pearson, 1895).

Table 4 presents ρ values interpretation according to Dancey and Reidy (2007).

Therefore, we tested $H_{01(w,s)}$ with Spearman’s ρ for each warning-smells pair, i.e., warning w and smell s , resulting in 663 distinct tests. Each test measured the strength of correlation between the number of instances of w and s at the package level.

To answer **RQ₂**, we computed Spearman’s ρ for each warning-smells combination. We used the interquartile ratio (IQR) method (Hojo & Pearson, 1931) to select the top correlated warning for each SAT and smells, i.e., we computed the quartiles. We selected only the warnings with their Spearman’s ρ value higher than or equal to the third quartile (Q3), thus selecting the top 25%. We computed the \mathcal{P} as follows: Let Σ_{SAT_i} represent the set of Static Analysis Warnings specific to a particular Static Analysis Tool SAT_i . Then, $\sigma \in \Sigma_{SAT_i}$ signifies that σ belongs to the set of warnings specific to SAT_i ; thus σ_j is a specific warning and $\alpha \in smells$ represents a specific smells.. Therefore, we define the relative frequency $\mathcal{P}(\sigma_j, \alpha)$ smells:

$$\mathcal{P}(\sigma_j, \alpha) = \frac{|\sigma_j \text{ in } \alpha|}{|\sigma \text{ in } \alpha|}$$

Where:

- The numerator represents the number of times the specific warning, $\sigma_j \in \Sigma_{\text{SAT}_i}$, appears within the given smells, α .
- The denominator represents the total count of all warnings belonging to a specific SAT, σ , in the given smells, α .

Informally, $\mathcal{P}(\sigma_j, \alpha)$ expresses how likely a specific warning σ_j is to appear within the context of packages affected by a given smell type α . This helps identify which warnings are most frequently associated with particular architectural issues.

More specifically, we computed \mathcal{P} for each possible combination of smells. Therefore, our data computed a \mathcal{P} -value for the 3 smells, the non-co-occurring instances, and for combinations of smells. We tested H_2 , and H_3 via the WPT (Conover, 1999), which is a non-parametric statistical test that compares two related samples or paired data. WPT uses the absolute difference between the two observations to classify and then compare the sum of the positive and negative differences. The test statistic is the lowest of both. We selected WPT to test H_2 and H_3 because the data was not normally distributed; hence, we used it instead of the paired t-test, which assumes a normal data distribution.

Moreover, due to the high number of statistical tests performed, we needed to apply p-value correction. In place of the usual level of significance ($\alpha = 0.05$), we need to correct p-values since it is essential in multiple hypothesis testing to control the increased risk of Type I errors. When multiple tests are conducted simultaneously, the probability of incorrectly rejecting at least one true null hypothesis increases. A classical p-value correction technique is the Bonferroni correction. However, Bonferroni is known to offer a conservative correction (Hochberg, 1988). Therefore, we decided to use the Benjamini & Hochberg (BH) p-value correction (Benjamini & Hochberg, 1995), which uses the false discovery rate (FDR) when performing multiple hypothesis tests, smells follows (Emmert-Streib & Dehmer, 2019):

1. **Rank the p-values:** Sort the p-values obtained from multiple tests in ascending order, denoted smells $p_{(1)} \leq p_{(2)} \dots \leq p_{(m)}$ for their respectively defined hypotheses $H_{(1)}, H_{(2)}, \dots, H_{(m)}$.
2. **Assign ranks:** Assign each p-value a rank, i , with $p_{(1)}$ being the smallest p-value and $p_{(m)}$ being the largest.
3. **Calculate the corrected p-values:** For each p-value $p_{(i)}$, calculate the BH-corrected p-value using the formula:

$$q_{(i)} = \min_{j \in \{1, \dots, m\}} \left\{ \frac{m \cdot p_{(j)}}{j}, 1 \right\}$$

where m is the total number of tests, $q_{(i)}$ is the corrected probability for the hypothesis at moment i and $p_{(j)}$ given the order rank $j \in \{1, \dots, m\}$ of p-values, which stands smells the probability at the rank position j .

4. **Determine correction index:** The detailed procedure identified the most extensive index k for which BH holds and rejects null hypotheses as follows:

$$k = \max \left\{ i \in 1, \dots, m \mid p_{(i)} \leq i \frac{\alpha}{m} \right\}$$

Which is automatically computed using existing analytical software.

5. **Corrected hypothesis testing:** Compare the corrected p-values $q_{(i)}$ from each hypothesis $H_{(1)}, H_{(2)}, \dots, H_{(m)}$ against the k index, that is:

$$p_{(i)} \leq i \frac{\alpha}{m}$$

To determine which hypotheses to reject, to the desired critical value, α (e.g., 0.05).

Since we performed the BH correction on the p-values of the defined hypotheses, we maintained the standard value of the critical value α at 0.05, hence gaining power in controlling for error Type I and Type II (Emmert-Streib & Dehmer, 2019).

Nevertheless, while BH correction mitigates false discoveries in large-scale testing, it may also suppress weaker, yet practically relevant, correlations. Future studies may benefit from exploring complementary methods such as smells, Bayesian correlation inference, or effect-size filtering to balance statistical rigor with exploratory sensitivity better.

To answer **RQ₃**, we leverage our definition of \mathcal{P} , the default severity associated with the specific warning, and the severity associated with the particular smells to build three different rankers. The ranking idea stemmed from previous research by Çarka et al. (2022), in which PofB and PopT metrics were normalized and used to evaluate classifier prediction concerning inspection effort (see Section 2.3).

Following the definition of PofB and Popt, we designed three rankers. More specifically, we ordered the data according to:

- warning severity: we used SonarQube severity associated with each warning.
- \mathcal{P} : we used \mathcal{P} and created separate rankings one for each smells combination.
- smells severity: we used the known smell severity to order the data.

We note that the smells severity-based order is a special case; by design, that ranker is the *optimal* one because it uses the smells severity. **RQ₃** aims to use the other two rankers to prioritize warnings that are more prone to induce smells. Therefore, the optima ranker is needed to compare the results of the other two. According to the interpretation of PofB and PopT, we selected the first x% of the data from the three rankings, with x ranging between 10 and 100 with increments of 10, e.g., top 10% ordered according to \mathcal{P} .

Therefore, for each selected percentage of data, we computed how many Critical, High, and Medium smell-prone warnings we would capture.

4 Results

In this section, we reported the results we obtained to answer our RQs.

4.1 Static analysis warnings and architectural smell correlation (RQ₁)

We tested $H_{11}(w, s)$, excluding all cross-correlations among warnings and focused on the correlations between warning and smell (Figure 2), Resulting in the rejection of the null hypothesis for each of the 663 warning-smell pair we measured the correlation of. Hence, we can affirm that **there is a statistically significant correlation between the presence of static analyzer warnings and the presence of architectural smells**. More specifically, warnings were weakly to moderately correlated with smell for each smell across the SAT.

Moreover, According to Table 5, we observe that the strongest correlations are associated with CheckStyle and FindBugs, with CheckStyle’s javadoc and sizes rules correlating with CD at ρ values above 0.47, and FindBugs’ The Bloaters reaching a peak correlation of $\rho = 0.51$ with CD. CD generally shows the highest correlations across tools, followed by HL and UD. PMD also shows consistent correlations, particularly for CD, with warnings such as Bean Members Should Serialize and Error Prone. These findings reinforce the idea that specific warnings, particularly those associated with structural or stylistic concerns, are statistically aligned with the presence of architectural erosion symptoms. Indeed, while the overall correlation is moderate, distinct and tool-specific warnings exhibit a stronger association with architectural smells.

4.2 Static analysis warnings inducing architectural smell (RQ₂)

We tested H_2 with WPT. We rejected the null hypothesis for 243,258 out of 275,380 warning-smell pairs, i.e., 88%. Therefore, on average, we can affirm that **there is a statistically significant difference in the co-occurrence of pairs of warning and smell**. More specifically, Figure 3 presents the distribution of H_2 test results. It is worth noticing that in the case of UD, there was almost an equal probability of rejecting or accepting the null hypothesis across warnings. Moreover, we note that there is an equal probability of rejecting the null hypothesis in the non-co-occurring cases and those affected by all three smells.

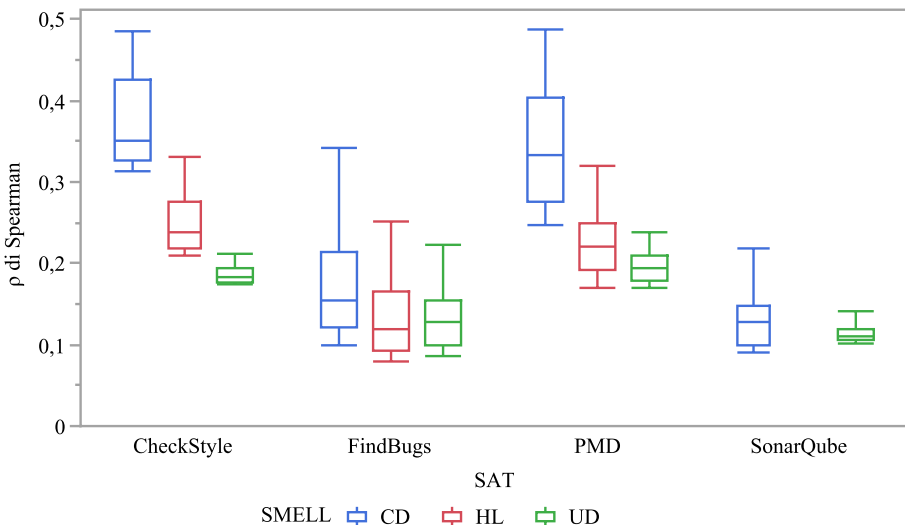


Fig. 2 Spearman’s ρ average distribution for warning by smell and SAT (RQ₁)

Table 5 Top 5 Most Significant Correlations Between warnings and smell by SAT

SAT	WARNING	SMELL	ρ	SAT	WARNING	SMELL	ρ
SonarQube	squid:S1066	HL	0,250915	CheckStyle	javadoc	CD	0,485681
SonarQube	squid:S1118	HL	0,248187	CheckStyle	sizes	CD	0,471169
SonarQube	squid:S1155	HL	0,234255	CheckStyle	Line Length Check	CD	0,466939
SonarQube	squid:S1226	HL	0,228857	CheckStyle	checks	CD	0,461382
SonarQube	squid:S1135	HL	0,228094	CheckStyle	Final Parameters Check	CD	0,450387
SonarQube	squid:Method Cyclomatic Complexity	CD	0,223376	CheckStyle	CD	HL	0,574997
SonarQube	squid:S134	CD	0,219365	CheckStyle	javadoc	HL	0,332039
SonarQube	squid:S1066	CD	0,187805	CheckStyle	checks	HL	0,308395
SonarQube	squid:S1188	CD	0,173311	CheckStyle	The Bloaters	HL	0,305625
SonarQube	squid:S1226	CD	0,165946	CheckStyle	Final Parameters Check	HL	0,298404
SonarQube	squid:Class Cyclomatic Complexity	UD	0,191145	CheckStyle	Overload Methods Declaration Order Check	UD	0,212942
SonarQube	squid:S1226	UD	0,141715	CheckStyle	javadoc	UD	0,207462
SonarQube	squid:S1067	UD	0,139032	CheckStyle	checks	UD	0,206098
SonarQube	squid:S134	UD	0,136966	CheckStyle	The Bloaters	UD	0,20556
SonarQube	squid:S1066	UD	0,136421	CheckStyle	Hide Utility Class Constructor Check	UD	0,204744
FindBugs	The Bloaters	CD	0,510642	PMD	Bean Members Should Serialize	CD	0,488396
FindBugs	The Dispensables	CD	0,400668	PMD	Error Prone	CD	0,486998
FindBugs	STYLE	CD	0,341812	PMD	Only One Return	CD	0,482231
FindBugs	PERFORMANCE	CD	0,292369	PMD	Code Style	CD	0,471469
FindBugs	The Object - Orientation Abusers	CD	0,282612	PMD	Documentation	CD	0,465746
FindBugs	The Bloaters	HL	0,396364	PMD	Documentation	HL	0,319861
FindBugs	The Dispensables	HL	0,303299	PMD	The Bloaters	HL	0,309873
FindBugs	The Encapsulators	HL	0,250777	PMD	Long Variable	HL	0,305122
FindBugs	code_smells:baseclass_abstract	HL	0,228306	PMD	Method Argument Could BeFinal	HL	0,298117
FindBugs	STYLE	HL	0,218801	PMD	Code Style	HL	0,294548

Table 5 (continued)

SAT	WARNING	SMELL	ρ	SAT	WARNING	SMELL	ρ
FindBugs	The Bloaters	UD	0,257588	PMD	Excessive Public Count	UD	0,265235
FindBugs	The Object - Orientation Abusers	UD	0,254754	PMD	GodClass	UD	0,237967
FindBugs	code_smells:baseclass_abstract	UD	0,22347	PMD	Excessive ClassLength	UD	0,228839
FindBugs	The Dispensables	UD	0,211089	PMD	Too Many Methods	UD	0,221366
FindBugs	The Encapsulators	UD	0,206185	PMD	Null Assignment	UD	0,217341

Furthermore, we tested H_3 with WPT. We can reject the null hypothesis in 30 out of 42 cases, i.e., 72% of the cases. Table 6 shows the pair of SAT by smell presence for which we could not reject the null hypothesis. Therefore, we can affirm that **there is a statistically significant difference in the co-occurrence of SAT-specific warnings and smell presence**.

Table 7 presents the top five warning co-occurring with specific smell. We selected the top five warnings that co-occur the most for each smell, including the “non-co-occurring”. We note that the most co-occurring warnings belong to FindBugs. Similarly, FindBugs also has the most “non-co-occurring” co-occurring warnings. Moreover, where three smells affect a single package, the FindBugs warning co-occurs each time. Furthermore, the least co-occurring smell is UD.

4.3 Architectural smell prioritization (RQ₃)

Figure 4 presents the distribution of \mathcal{P} averaged across warnings across SAT per smell, i.e., the percentage of occurrences of warnings related to the occurrence of one or more smells or their absence (i.e., non-co-occurring). On average, warnings belonging to SonarQube are more likely to raise warnings about non-co-occurring packages and classes. Most warnings co-occur with CD smell and have a similar presence among the SATs. Moreover, second to CD, warnings co-occurred with all three smells. In this context, FindBugs’ warnings exhibit an average higher co-occurrence.

Figure 5 presents the warning prioritization based on warning Severity-based, \mathcal{P} -based, and optimal. We computed how many medium, high, or critical smell could be discovered by inspecting the first x% of data ranked by one of the three methods. According to its design, the optimal ranking prioritized the warning co-occurring with high and critical smell. The warning severity-based and UD+CD rankers demonstrate the most balanced performance, while the ranking based on warning co-occurring with CD favors moderately severe smell.

Finally, we tested H_4 with the WPT. We could reject the null hypothesis in all cases for high and critical smell severities, in which the optimal ranker was compared with the warning-severity-based and the \mathcal{P} based rankers. Conversely, we could reject the hypothesis when comparing the warning-severity-based and the \mathcal{P} based rankers, but only in the specific case when comparing the UD + HL + CD proneness based on \mathcal{P} . Furthermore, when the difference was statistically significant, the gains were in favor of the optimal rankers, as expected. Due to space constraints, we included the whole table and the graphical repre-

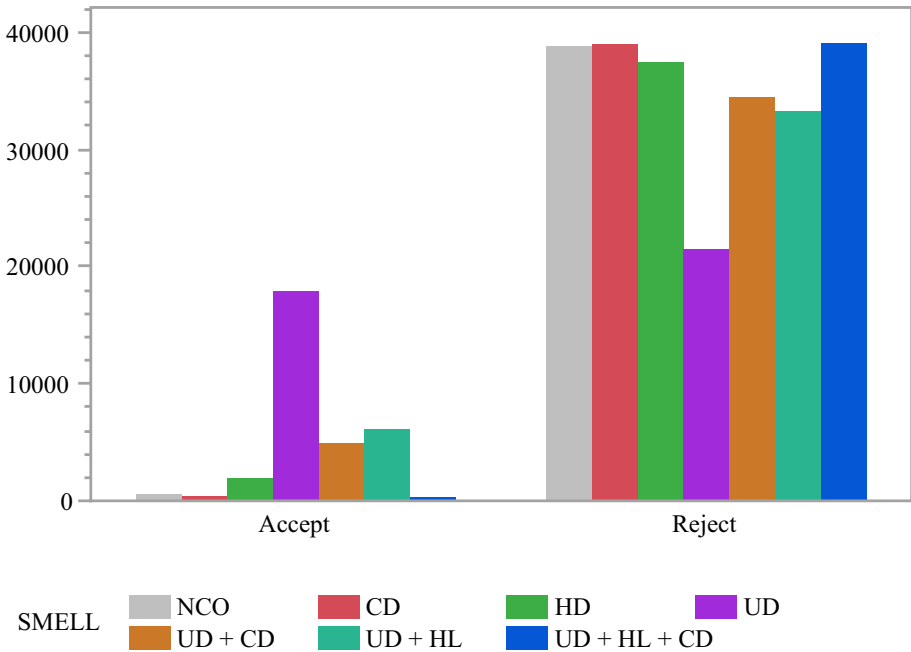


Fig. 3 Distribution of H_2 test results (RQ₂)

Table 6 Wilcoxon All Pair Test, i.e., H_{03} p-value (RQ₂)

SMELL	NCO	CD	CD	CD	CD	CD
SAT	PMD	PMD	FindBugs	PMD	SonarQube	SonarQube
Compared with	CheckStyle	CheckStyle	CheckStyle	FindBugs	CheckStyle	FindBugs
p-value	0,76	0,25	0,44	0,85	0,58	0,20
AS	CD	UD	UD + CD	HL	UD + HL + CD	
SAT	SonarQube	FindBugs	PMD	PMD	FindBugs	PMD
Compared with	PMD	CheckStyle	CheckStyle	CheckStyle	CheckStyle	CheckStyle
p-value	0,02	1,00	0,14	0,40	0,03	0,28

sensation of the gains among the approaches in the replication package. Moreover, we can **not reject the null hypothesis** for the medium risk for each ranker. Therefore, the optimal ranker behaves similarly to the other two rankers.

Hence, we can affirm that **there is a statistically significant difference in the prioritized effort between the three rankers**.

Nevertheless, based on the results, we can affirm that **warning prioritization based on \mathcal{P} can effectively guide smell remediation effort**.

5 Discussion

This section reflects on our findings, their practical implications, and how they compare to existing literature.

Table 7 The top five warning co-occurring with specific smell

SAT	WARNING	SMELL	% co-occurring
FindBugs	AT_OPERATION_SEQUENCE_ON_CONCURRENT_ABSTRACTION	UD + HL + CD	100
FindBugs	DMI_EMPTY_DB_PASSWORD		100
FindBugs	DMI_ENTRY_SETS_MAY_REUSE_ENTRY_OBJECTS		100
FindBugs	DMI_RANDOM_USED_ONLY_ONCE		100
FindBugs	EQ_COMPARING_CLASS_NAMES		100
PMD	Use String Buffer For String Appends	UD + HL	6,132
PMD	Assignment In Operand		4,856
PMD	code_smells:spaghetti_code		3,448
PMD	Use Collection Is Empty		2,435
PMD	Boolean Get Method Name		1,904
SonarQube	squid:Class Cyclomatic Complexity	UD + CD	3,418
PMD	Avoid Prefixing Method Parameters		3,222
SonarQube	squid:S1197		2,300
FindBugs	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE		2,270
FindBugs	BC_UNCONFIRMED_CAST		2,178
SonarQube	S2386	UD	0,429
SonarQube	S1444		0,294
SonarQube	S1141		0,267
SonarQube	S1118		0,242
SonarQube	S1226		0,223
FindBugs	DMI_COLLECTION_OF_URLS	HL	40,000
FindBugs	OBL_UNSATISFIED_OBLIGATION		33,871
CheckStyle	No White space After Check		31,748
FindBugs	EQ_DOESNT_OVERRIDE_EQUALS		21,739
PMD	LongVariable		18,457
FindBugs	NS_NON_SHORT_CIRCUIT	CD	88,235
FindBugs	ICAST_INTEGER_MULTIPLY_CAST_TO_LONG		87,500
FindBugs	RV_CHECK_FOR_POSITIVE_INDEXOF		80,000
FindBugs	DM_GC		78,571
FindBugs	VA_FORMAT_STRING_USES_NEWLINE		75,000
FindBugs	NM_METHOD_NAMING_CONVENTION	NCO	89,865
FindBugs	RI_REDUNDANT_INTERFACES		86,029
FindBugs	SIO_SUPERFLUOUS_INSTANCEOF		70,130
FindBugs	EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS		68,421
PMD	Compare Objects With Equals		66,212

1. Why Not Just Use Arcan?

SATs are ubiquitous in development environments, while smells detection tools like ARCAN require architectural context, often unavailable or costly to extract. Our warning-based method offers a low-cost, always-on signal to prioritize remediation. It does not replace smells tools, but complements them in tooling-constrained or high-velocity settings.

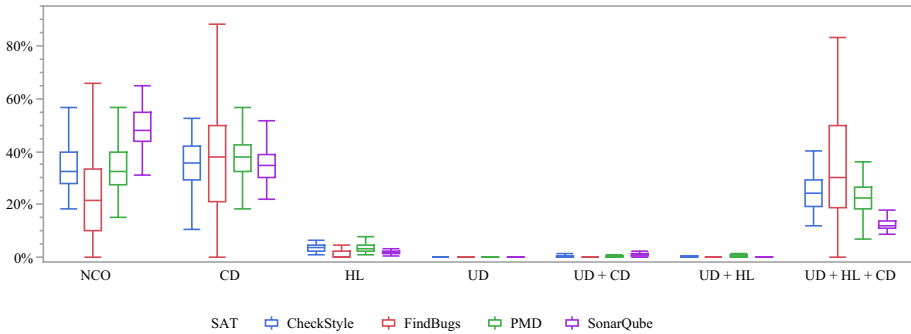


Fig. 4 Distribution of \mathcal{P} averaged among warning across SAT per smell (RQ₂)

Our analysis revealed a moderate but statistically significant correlation between warnings and smells. While not strong enough to determine causation, the presence of certain warnings consistently aligned with specific smells across multiple systems. Interestingly, the strength of this correlation varied depending on the SAT and the warning type. FindBugs, in particular, produced a disproportionately high number of warnings that co-occurred with smells, suggesting that its rule set is more sensitive to architectural-level concerns.

Among the smells we studied, UD showed the least co-occurrence with warnings. This suggests that UD issues may lie outside the detection capabilities of standard SA tools or may manifest in ways not captured by low-level violations. Future research should explore alternative indicators or more abstract static metrics for such architectural erosion patterns (Li et al., 2022; Bass et al., 2021).

2. Detect Hard-to-Find Smells

Like Unstable Dependency (UD), which co-occur less frequently with warnings, may require complementary indicators or higher-level abstractions. Exploring combinations of warnings, architectural metrics, or dynamic signals could improve coverage.

Interestingly, a handful of static analysis warnings surfaced as canaries in the coal mine, i.e., co-occurring with all three architectural smells (UD, HL, and CD). One such case is `DMI_EMPTY_DB_PASSWORD`, which flags hardcoded empty database credentials. Beyond being a glaring security flaw, this warning hints at systemic issues: poor layering, tangled dependencies, and low cohesion. Another is `EQ_COMPARING_CLASS_NAMES`, where equality checks rely on class names rather than proper instance comparisons—often a sign of brittle design and misuse of reflection or custom loading logic. Finally, `DMI_ENTRY_SETS_MAY_REUSE_ENTRY_OBJECTS` may appear trivial at first glance, but its misuse of shared objects can break encapsulation and introduce tight coupling across layers. While these individual violations may seem minor, their alignment with architectural smells hints they often live in “bad neighborhoods” of the codebase, regions suffering from architectural erosion on multiple fronts. Spotting these outliers early could help teams prevent cascading issues and prioritize fixes in areas that are both structurally weak and operationally risky.



Fig. 5 Comparison of warning prioritization based on warning Severity-based, \mathcal{P} -based, and optimal

3. Surface Deeper Architecture Erosion

A small set of warnings consistently co-occurred with all three smells, hinting at concentrated zones of architectural erosion. These high-signal warnings may serve as early alerts for deeply problematic components. Teams should treat such violations as gateways to uncover broader design flaws and prioritize their remediation accordingly.

Conversely, a large subset of warnings, 33.79% of our dataset, never co-occurred with any smells. We refer to these smells “non-co-occurring”. Since these warnings showed no statistical link to architectural erosion, they can be deprioritized when teams are constrained by time or resources. By focusing on the smell-prone subset of warnings, developers can streamline quality assurance efforts and reduce refactoring overhead.

4. Focus Refactoring Efforts

Teams can reduce noise and effort by filtering out 33.79 of warnings identified as “non-co-occurring smells”. These warnings showed no link to architectural smells and can be safely deprioritized when quality assurance resources are limited.

Anyhow, we could question whether the observed co-occurrence between warnings and smells is genuinely informative, or merely a consequence of the high volume of warnings emitted by SATs. After all, SATs are known to generate alerts across most code entities. While this observation is valid, our contribution lies in distinguishing signal from noise.

Consider a common real-world scenario. A team working on an extensive legacy system integrates SonarQube, FindBugs, and PMD into their CI/CD pipeline. Over time, they are flooded with warnings, naming issues, unused variables, null checks, and developers begin to ignore them. Now imagine this team learns that specific warnings, such as smells `NS_NON_SHORT_CIRCUIT` or `DM_GC`, appear in over 75% of cases where architectural smells like `CD` are also present. This reframes those warnings from low-priority noise to meaningful indicators of deeper architectural problems.

Therefore, while warnings may be unsurprising, the statistical associations we uncover, based on structured co-occurrence patterns, hypothesis testing, and severity-aware prioritization, provide valuable insights. We demonstrate that not all warnings are equally relevant: some consistently co-occur with smells and serve smells lightweight architectural risk indicators, while others (“non-co-occurring smells”) do not. This finding enables developers to focus their efforts more effectively, particularly in fast-paced settings where architectural tools like ARCAN are unavailable or impractical.

5. Not All Warnings Are Equal

Our findings show that only a subset of warnings consistently co-occur with architectural smells. Identifying these “high-risk” warnings helps developers prioritize attention and reduce alert fatigue caused by the broader volume of low-impact static analysis results.

5.1 Previous study results

From a practical perspective, these findings offer concrete guidance. Development teams can confidently filter out non-co-occurring smells and instead concentrate on warnings that frequently co-occur with smells. When explicit smell severity data is unavailable, they can prioritize remediation based on either warning severity levels or warning–smell co-occur-

rence probabilities. Both strategies, as shown in our ranking analysis (Figure 5), approximate the optimal remediation order based on smell severity.

6. Prioritize Using warning Signals

Without architectural smell severity data, developers can use either warning severity or warning–smells co-occurrence probability (\mathcal{P}) to prioritize refactoring. These strategies closely match the ideal orderings based on actual smell severity.

For example, SonarQube users could enrich their dashboards with co-occurrence probabilities to better triage warnings. At the same time, reviewers could flag high-risk FindBugs warnings, such as smells that are frequently linked to CD, and smells candidates for early intervention. Although this method does not replace architecture-specific analysis tools, it provides an accessible, lightweight alternative that integrates seamlessly into existing CI/CD workflows.

7. Improve Static Analysis Tools

SAT dashboards, such as those in SonarQube or FindBugs, could integrate smell-proneness scores derived from warning co-occurrence metrics. This would help developers triage high-risk violations and proactively prevent architecture erosion.

These insights directly address RQ_3 , which explores how warning-based indicators can support smells remediation. Our prioritization strategies, particularly those using warning severity and co-occurrence likelihood, produced rankings that closely matched smells-severity-based rankings. This shows that architectural concerns can be anticipated, at least in part, through patterns already embedded in developer workflows.

Compared to prior work such as smells (Fontana et al., 2019), which found weak correlations between code smells and smells, our study reveals that certain warnings, though more granular and often disregarded, carry meaningful signals about architectural issues. This distinction highlights our novel contribution: bridging SAT output with architectural-level insights, while maintaining compatibility with tools developers already use.

Our findings also align with those of Macia et al. (2012b), who showed that code anomalies introduced early in a project often lead to architectural erosion. Their work emphasized manually validated anomalies in a limited set of systems. By contrast, our study generalizes this observation across 103 projects, using automatically extracted warnings from widely adopted tools. Warnings offer a more representative and fine-grained lens into the types of quality issues developers regularly encounter.

We also extend the insights of Oizumi et al. (2016), who found that semantic aggregations of code anomalies better indicate architectural problems than individual instances. Our study confirms that aggregated co-occurrence patterns, especially from tools like FindBugs, are useful in estimating the likelihood and severity of smells. Moreover, we provide a prioritization framework grounded in these co-occurrence metrics, offering a more scalable and automation-friendly solution.

5.2 Implications for researchers

This study opens up several research directions. First, while we confirmed a moderate correlation between warnings and smells, the variation across tools and smells suggests that deeper semantic properties influence these links. Future work could explore contextual factors such as smells code structure, change history, or architectural annotations to improve prediction models.

Second, the concept of “non-co-occurring smells” introduces a novel dimension for understanding warning reliability. Investigating what makes these warnings architecturally benign, e.g., warning type, location, or context, could help refine rule sets for static analysis tools, minimizing false positives and increasing developer trust.

Finally, given the scalability of our approach across 72 million Lines of code and over 800,000 classes, future studies can explore transfer learning or AI-driven prioritization across repositories and ecosystems. This could lead to generalizable models for quality-aware development that adapt to project characteristics and team practices.

8. Build Smarter Prioritization Models

By studying what makes some warnings “non-co-occurring,” researchers can refine rule sets to reduce false positives. Moreover, large-scale, cross-system studies may support AI-driven prioritization systems that adapt to diverse project contexts.

Overall, our work serves smells a bridge between static analysis and architectural health monitoring, bringing together low-level tooling and high-level design quality in a way that supports both daily development and long-term maintainability.

Our finding also opens the door to a new class of tooling: one that connects co-occurrence signals across multiple warnings and smells to detect emergent architectural pathologies. For instance, a cluster of warnings repeatedly linked to UD, HL, and CD could indicate a nascent **God Component**, **Blob**, or **Lava Flow**, patterns that rarely surface through individual metrics alone. By layering co-occurrence with topology- or dependency-based metrics, future tools could provide early, lightweight detection of complex architectural **anti-patterns**. This would allow teams to intervene before such structures fully take shape and become costly to disentangle.

9. Spot Architectural Anti-Patterns Early

By tracking groups of warnings that consistently co-occur with multiple smells, teams can identify early symptoms of architectural anti-patterns like God Components or Lava Flows. These compound indicators support a new direction in static analysis—lightweight, design-aware smell detection without requiring full architecture recovery.

6 Threats to validity

In this section, we discuss the threats to the Validity of our study. We categorized the threats into Construct, Internal, External, and Conclusion, following the guidelines defined by Wohlin et al. (2012).

Construct validity Construct validity concerns how our measurements reflect what we claim to measure (Wohlin et al., 2012).

Our specific design choices may impact our results, including our measurement process and data filtering. To address this threat, we based our selection on past studies (Arcelli Fontana et al., 2017; Tempero et al., 2010; Sas & Avgeriou, 2023) and used well-established guidelines in designing our methodology (Runeson & Höst, 2009; Basili et al., 1994). More specifically, we used three static analysis tools considered to be the most adopted by developers to detect software quality warnings (Esposito et al., 2024) and ARCAN to detect the four architectural smells.

Nevertheless, we acknowledge that the analyzed warning does not represent all the possible warnings currently available in the state-of-the-art. Our study selected the most popular SATs. Nonetheless, future works will focus on extending the current selection of warnings. Similarly, we analyzed only three AS due to the lack of AS detection tools.

Moreover, the original data is not recent. Therefore, it can be argued that it may face deprecation. Similar arguments can be put towards the SAT. It is worth noticing that deprecation, in this context, is a multifaceted threat. Firstly, according to Esposito and Falessi (2024), deprecation cannot impact generalizability. Deprecation, in this specific context, may affect the quantity of AS detected, given that the concept of microservices has recently reached its peak of interest⁷.

Internal validity Internal validity is the extent to which an experimental design accurately identifies a cause-and-effect relationship between variables (Wohlin et al., 2012). Moreover, according to Wohlin et al. (2012), internal validity not only concerns causal relationships, which our study does not directly claim, but also involves controlling for bias or confounding factors, ensuring consistency in data collection, and avoiding measurement errors. Indeed, our study relies on a large-scale analysis of 103 popular Java projects in the Qualitas Corpus dataset. However, they can potentially be biased by their project selection. We address this threat by designing our inclusion criteria based on past studies (Terra et al., 2013; Tempero et al., 2010). Furthermore, we addressed tool selection bias by including multiple SATs widely adopted in industry and using ARCAN, one of the few validated tools for AS detection. Our findings remain valid within the scope of the selected tools.

External validity External validity concerns how the research elements (subjects, artifacts, etc.) represent actual elements (Wohlin et al., 2012). External validity concerns how the research elements (subjects, artifacts, etc.) represent actual elements (Wohlin et al., 2012). Our study is based on the Qualitas Corpus dataset, which was last updated in 2013. As such, it includes monolithic Java systems and does not reflect recent architectural paradigms, such as microservices or cloud-native systems. This limits the generalizability of our findings to

⁷<https://content.ardoq.com/gartner-magic-quadrant-for-enterprise-architecture-tools>

modern software projects. The dynamics between warnings and smells in contemporary ecosystems may differ due to design practices, modularity, deployment infrastructure, and static analysis tooling changes. Future work should replicate this study on newer datasets reflecting current architectural trends. The analyzed project comes from the GitHub public repository. Mining versioning systems, particularly GitHub, also threaten validity. More specifically, GitHub's user base predominantly comprises developers and contributors to open-source projects, potentially skewing findings towards this specific demographic. In our context, this is not a real threat because it is the study's focus. Moreover, the dynamic nature of GitHub, with frequent updates, forks, and merges, poses challenges in ensuring the stability and consistency of data over time. We addressed this issue by providing the raw data in our replication package. Furthermore, the accessibility of GitHub data is subject to various permissions and restrictions set by project owners, potentially hindering reproducibility and transparency in research. We addressed this research, limiting our data collection to public repositories. Regarding generalizability, the vast diversity of projects analyzed in our study presents a significant diversity in terms of programming languages, project sizes, and development methodologies, thus aiding the generalizability of results.

Conclusion validity focuses on how we draw conclusions based on the design of the case study, methodology, and observed results (Wohlin et al., 2012). The distribution of our data determines the type of statistical tools we can use to test our hypothesis. We tested the data to assess if it departed from normality with the AD test, and we could reject the null hypothesis for each test. Therefore, we selected the WT test instead of the t-test as Spearman's ρ in place of Pearson's. Furthermore, statistical tests threaten the validity of the appropriateness of statistical tests and procedures, such as assumption violation, multiple comparisons, and Type I or Type II errors. We address this issue using WT instead of the t-test due to the rejection of the normal data distributions. Moreover, we applied the HB correction (Kim & Choi, 2021) to balance type 1 and type 2 errors well. We also set our alpha to 0.01, hence reducing it from the standard value of 0.05 due to the many statistical tests we performed.

Spearman's ρ is a non-parametric measure of the strength and direction of a correlation between two variables, but due to multiple factors, this approach may hinder the validity. For instance, the correlation coefficient is unreliable with a small sample size and cannot accurately represent the relationship between variables. Moreover, Spearman assumes a linear monotonic correlation between the variables; if the assumption were to prove invalid or not applicable, Spearman's correlation might not accurately measure the strength and direction of the association. We have mitigated this issue by ensuring a representative sample and checking the data distribution. We could reduce this issue by using other non-parametric measures of correlation. Still, our analysis found that Spearman's ρ was more suitable than other correlation measures, such as Kendall's τ , because it can effectively handle tied observations in the data. On the other hand, Kendall's τ relies on concordant and discordant pairs of observations.

7 Related work

This section discusses works related to our study regarding SATs and smells. Table 8 compares our work with prior studies on static analysis tools and architectural smells. Existing research has investigated SAT usage, developer perceptions, or the evolution of smells, yet none examined direct correlations between SAT warnings and architectural smells. According to Table 8, our study distinguishes itself by providing the first large-scale empirical evidence on their relationship and by proposing warning-based prioritization strategies that effectively approximate remediation guided by architectural impact severity. Therefore, we position our contribution as a pragmatic bridge between ubiquitous SAT outputs and higher-level architectural erosion.

7.1 Static analysis tools

In this section, we report the relevant work on SAT focusing on their usage, warnings, and the detected issues (Flanagan et al., 2002; Heckman & Williams, 2011; Beller et al., 2016). In recent years, SATs have strongly increased in popularity (Vassallo et al., 2019; Lenarduzzi et al., 2019c) given their ease-to-use (Zampetti et al., 2017). Along with this attention increase, researchers have focused on analyzing SAT usage (Wagner et al., 2005; Nagappan & Ball, 2005; Zheng et al., 2006; Nanda et al., 2010) and effectiveness (Esposito et al., 2024). For instance, SATs effectively improve bug prediction models (Querel & Rigby, 2018). Developers use SAT to inspect software codebases, identify bugs (Rutar et al., 2004; Tomas et al., 2013), evaluate code quality (Johnson et al., 2013; Schnappinger et al., 2019; Marcilio et al., 2019), and address coding issues (Saarimäki et al., 2019; Lenarduzzi et al., 2019a, 2020c).

Recent studies have explored SAT's application across various programming languages (Beller et al., 2016), their configurations (Beller et al., 2016), and their evolution within software systems (Falessi et al., 2017; Digkas et al., 2018; Saarimäki et al., 2019). However, fewer studies have focused on the impact of SAT on software quality (Falessi et al., 2017; Tollin et al., 2017; Lenarduzzi et al., 2020c, 2021; Trautsch et al., 2020).

Moreover, the research community also focused on prioritization strategies for addressing warnings. For instance, researchers proposed prioritization based on removal times (Kim & Ernst, 2007), or developer engagement in remediating specific warnings (Marcilio et al., 2019). Moreover, studies have compared estimated warning resolution times with actual developer efforts (Digkas et al., 2018; Saarimäki et al., 2019; Baldassarre et al., 2020; Lenarduzzi et al., 2020b).

Researchers have extensively studied SonarQube regarding the impact of warning on software quality. They investigated the fault-proneness of SonarQube rules (Falessi et al., 2017; Lenarduzzi et al., 2020c, a) and their susceptibility to changes (Tollin et al., 2017; Lenarduzzi et al., 2020c). Findings indicate that SonarQube rules increase change-proneness at the class level, while correlations between faults and rules can reduce fault-proneness (Falessi et al., 2017; Lenarduzzi et al., 2020c).

Similarly, researchers have examined PMD warnings concerning their impact on pull request acceptance. Results show quality warnings do not affect pull request acceptance (Lenarduzzi et al., 2021). A comprehensive study (Trautsch et al., 2020) on warning trends in OSS projects using PMD highlighted that large-scale changes in SAT warnings

Table 8 Comparison of our work with related studies on SATs and smells

Study	Focus	Method / Scope	Contributions and Gaps
Falessi et al. (2017); Lenarduzzi et al. (2020c); Tollin et al. (2017)	SAT rules and quality impact	Empirical studies on SonarQube rules, class-level correlations	Showed links between warnings and change/fault-proneness, but did not study architectural smells
Trautsch et al. (2020); Lenarduzzi et al. (2021)	Warning evolution and trends	Longitudinal analyses with PMD, PR acceptance	Studied evolution and acceptance, but no connection with architectural issues
Johnson et al. (2013); Wright et al. (2020)	Developer perceptions of SATs	Interviews, surveys, user studies	Identified adoption barriers (e.g., false positives), but no quantitative correlation with architecture
Yang et al. (2021); Ge et al. (2025); Li and Yang (2024)	Improving SAT utility	Data mining, preference mining, automated tracking	Enhance prioritization of warnings, but focus on code-level issues
Fontana et al. (2019); Macia et al. (2012b); Oizumi et al. (2016); Herold (2020); Sas et al. (2022)	Architectural smells and erosion	Correlation studies, anomaly aggregation, longitudinal analyses	Explored code–architecture links, but did not involve SAT warnings
Gnoyke et al. (2024); Jolak et al. (2025)	Smell evolution and tooling	Novel tools and empirical studies	Extend smell detection, but do not combine SATs with smells
Simões and Venson (2024); Yıldırım et al. (2024); Nizam et al. (2025)	LLMs as static analysis	Benchmarks vs. SATs, embedding optimization	Emerging use of AI for quality checks, no empirical correlation with architectural smells
Our work	SAT warnings ↔ architectural smells	Large-scale study (103 projects, 72M LOC), 3 SATs + ARCAN	First to reveal moderate correlations, propose prioritization strategies bridging code-level warnings and architectural remediation

often result from coding style modifications, such smells braces and naming conventions. The presence of PMD in build configurations positively impacts defect density, suggesting an improvement in external quality.

Recently, Wright et al. (2020) investigated why developers use SAT, revealing new tool requirements such smells systems for recommending warnings based on developer expertise and collaborative interfaces for warning analysis. Conversely, Johnson et al. (2013) examined why developers do not widely use SAT tools, identifying false positives and the presentation of warnings smells significant barriers. Furthermore, Yang et al. (2021) explored enhancing the utility of static code warning tools through data mining algorithms that filter out commonly ignored warnings, using FindBugs smells in a case study. Similarly, Ge et al. (2025) proposed preference mining to recommend the most suitable SAT results for each project accordingly. Li and Yang (2024) studied the benefits of automating the tracking of the evolution of static code warnings with StaticTracker, a tool that combines results from PMD and SpotBugs, successor of FindBugs, for static bug detection.

Along with the rise of new techniques such as Large Language Models (LLMs), Simões and Venson (2024) evaluated the overall quality of code with LLMs as SATs, and compared their results with those from SonarQube. Yıldırım et al. (2024) scaled up this research trend

by comparing 10 SATs against four LLMs on assessing their ability to detect a compilation of 40 API vulnerabilities in the source code. Furthermore, Nizam et al. (2025) explored different embedding techniques to improve the quality of code embeddings when leveraging LLMs for source code analysis.

7.2 Architectural smells

Smells have been less thoroughly investigated than code smells. While numerous studies have explored correlations between code smells, research into the relationship between code smells and architectural smells has been limited. One study found a minimal correlation between smells and code smells, suggesting that smells cannot be reliably inferred from code smells (Fontana et al., 2019).

The connection between code anomalies, similar to code smells, and smells has also been studied with mixed results. Some findings indicated that many detected code anomalies did not correspond to architectural issues (Macia et al., 2012b), while other results showed that over 80% of architectural problems were linked to code smells (Macia et al., 2012a). Specific code smells like Long Method, and God Class were consistently associated with architectural problems.

Oizumi et al. (2016) investigate the aggregation of code anomalies and their relationship to smells and show that a single anomaly can signal an architectural problem. Specifically, 50% of syntactic aggregations and 80% of semantic aggregations are linked to design issues (Oizumi et al., 2014, 2016). A study does not find a strong correlation between smells and architectural erosion, outlining that smells cannot explain architectural erosion alone (Herold, 2020), while Sas et al. (2022) highlights the effects of smells on long-term system maintainability and evolvability, supported by interview excerpts. Practitioners highlight which parts of the smells analysis offer actionable insights for planning refactoring activities.

Moreover, Mumtaz et al. (2022) shows that community smells, particularly the Missing Links smell, are related to design smells. They discuss specific refactoring techniques that concurrently address community and design smells, managing social and technical issues together.

Recently, Gnoyke et al. (2024) studied the evolution patterns of smells to describe symptoms of architectural erosion. They present extended techniques to measure either intra-version as well as inter-version smells. Following the latest works on smells, Jolak et al. (2025) introduced a novel open-source smell analysis tool, which gathers and analyzes data related to smells, modularity and testability. Additionally, they studied the statistical relationship between different types of smells with modularity and testability.

Finally, Fontana et al. (2019) on the correlation between technical debt indices based on code-level issues from SonarQube revealed no correlation, suggesting differing impacts based on the features considered (Roveda et al., 2018). Although several tools have been developed for detecting architectural debt, including DV8, Designite, Jarchitect, and others, which focus on smells in various programming languages (Samarthyam et al., 2016; Ganesh et al., 2013; Bergstrom et al., 2003; Jolak et al., 2025), no prior studies have examined correlations between smells and warning smells we have addressed in this paper.

8 Conclusion

Our results show a statistically significant positive correlation between warnings and smells, showing weak to moderate correlations across different tools and warnings and smells in 661 out of 663 tests. In terms of co-occurrence, we found a statistically significant difference in the pairs of warnings and smells, with 88% of the Wilcoxon All-Pair Ranking Test cases rejecting the null hypothesis. Specifically, we showed that warnings can influence the presence of smells. When prioritizing architectural smells, our findings indicate that using a prioritization method based on the probability of warning occurrences (\mathcal{P}) can effectively guide remediation efforts, revealing a statistically significant difference between the three ranking methods tested. Our proposed approach to warning prioritization highlights that, in the absence of smell indications, warning severity and our empirically computed \mathcal{P} together with the possibility of dropping a third of the data to inspect, proved to be an effective smells remediation effort prioritization.

Future research efforts should enhance SAT capabilities to detect the most smell-prone warnings more accurately, improving smell remediation prioritization, and expanding the smells detectable by ARCAN. Additionally, research should aim to deepen the analysis of warnings' smell-proneness to better trace smells back to warnings. This will help understand the motivations or root causes linking specific warnings to specific smells.

Author Contributions We specify our contributions according to the CRediT taxonomy: **Matteo Esposito:** Methodology, Writing – Original Draft Preparation. Matteo developed the primary research methodology and contributed significantly to writing the initial manuscript draft. **Mikel Robredo:** Writing – Original Draft Preparation, Data Curation. Mikel provided essential data preparation and contributed to drafting specific sections. **Francesca Arcelli Fontana:** Validation, Writing – Review & Editing, Supervision. Francesca critically validated the study's results, supervised the research's methodological approach, and reviewed the manuscript for academic rigor. **Valentina Lenarduzzi:** Conceptualization, Methodology, Supervision, Validation, Writing – Review & Editing. Valentina led the conceptualization and methodological framing of the research, supervised the entire project, validated findings, and contributed to reviewing and refining the manuscript.

Funding Open Access funding provided by University of Oulu (including Oulu University Hospital). This research received no external funding and was supported solely by the authors' institutions.

Data Availability We provide a replication package on Zenodo(<https://zenodo.org/doi/10.5281/zenodo.11366846>), which includes the raw data, comprehensive instructions for executing the SAT, a complete list of projects identified as containing warnings and smells, as well as the full results of the statistical analyses, covering both the normality assessments and the hypothesis tests related to our research questions.

Declarations

Competing Interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Al-Mutawa, H. A., Dietrich, J., Marsland, S., et al. (2014). On the shape of circular dependencies in java programs. *ASWEC, 2014*, 48–57.
- Arcelli Fontana, F., & Zanoni, M. (2011). On investigating code smells correlations. *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 474–475). Berlin, Germany: RefTest Workshop.
- Arcelli Fontana, F., Pigazzini, I., Roveda, R., et al. (2017). Arcan: A tool for architectural smells detection. In *International Conference on Software Architecture Workshops, ICSA*, (pp. 282–285)
- Arisholm, E., Briand, L.C., & Fuglerud, M. (2007). Data mining techniques for building fault-proneness models in telecom java software. In: *ISSRE 2007, The 18th IEEE International Symposium on Software Reliability*, Trollhättan, Sweden, 5-9 November 2007. IEEE Computer Society, (pp. 215–224). <https://doi.org/10.1109/ISSRE.2007.22>.
- Avgeriou, P., Taibi, D., Ampatzoglou, A., et al. (2021). An overview and comparison of technical debt measurement tools. *IEEE Software*.
- Baldassarre, M. T., Lenarduzzi, V., Romano, S., et al. (2020). On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. *Information and Software Technology*, 128, 106377.
- Basili, V.R., Caldiera, G., & Rombach, H.D. (1994). The goal question metric approach. *Encyclopedia of Software Engineering*
- Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice*. Addison-Wesley Professional.
- Beller, M., Bholanath, R., McIntosh, S., et al. (2016). Analyzing the state of static analysis: A large-scale evaluation in open source software. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, (pp. 470–481).
- Benjamini, Y., & Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)*, 57(1), 289–300.
- Bergstrom, S., Boskovic, J., & Mehra, R. (2003). *Development of the adaptive reconfigurable control analysis, design, and evaluation (arcade) toolbox*
- Çarka, J., Esposito, M., & Falessi, D. (2022). On effort-aware metrics for defect prediction. *Empirical Software Engineering*, 27(6), 152.
- Chen, H., Liu, W., Gao, D., et al. (2017). Personalized defect prediction for individual source files. *JSJKX*, 44(4), 90–95. <https://doi.org/10.11896/j.issn.1002-137X.2017.04.020>
- Conover, W. (1999). *Practical nonparametric statistics*, (3rd edn). New York, NY [u.a.].
- Dancey, C.P., & Reidy, J. (2007). *Statistics without maths for psychology*. Pearson education
- Deeb, S., Benlhdri, M., Ammar, H., et al. (2021). Refactoring cost estimation for architectural technical debt. *International Journal of Software Engineering and Knowledge Engineering*, 31(02), 269–28. <https://doi.org/10.1142/S021819402150008X>
- Desai, A.B., & Parmar, J.K. (2016). Refactoring cost estimation (rce) model for object oriented system. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, (pp. 214–218). <https://doi.org/10.1109/IACC.2016.48>
- Digkas, G., Lungu, M., Avgeriou, P., et al. (2018). How do developers fix issues and pay back technical debt in the apache ecosystem? In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (pp. 153–163).
- Emmert-Streib, F., & Dehmer, M. (2019). Large-scale simultaneous inference with hypothesis testing: multiple testing procedures in practice. *Machine Learning and Knowledge Extraction*, 1(2), 653–683.
- Ernst, N.A., Bellomo, S., Ozkaya, I., et al. (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. *Symposium on the Foundations of Software Engineering*, 50–60.
- Esposito, M., & Falessi, D. (2023). Uncovering the hidden risks: The importance of predicting bugginess in untouched methods. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (pp. 277–282). <https://doi.org/10.1109/SCAM59687.2023.00039>
- Esposito, M., & Falessi, D. (2024). Validate: a deep dive into vulnerability prediction datasets. *Information and Software Technology*, 107448.
- Esposito, M., Moreschini, S., Lenarduzzi, V., et al. (2023). Can we trust the default vulnerabilities severity? In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, (pp. 265–270)
- Esposito, M., Falaschi, V., & Falessi, D. (2024). *An extensive comparison of static application security testing tools*. [arXiv:2403.09219](https://arxiv.org/abs/2403.09219)
- Falessi, D., Russo, B., & Mullen, K. (2017). What if i had no smells? In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, (pp. 78–84).

- Flanagan, C., Leino, K.R.M., Lillibridge, M., et al. (2002). Extended static checking for java. In *Conference on Programming Language Design and Implementation*, (pp. 234–245).
- Fontana, F.A., Pigazzini, I., Roveda, R., et al. (2016). Automatic detection of instability architectural smells. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, Raleigh, NC, USA, October 2-7, 2016, (pp. 433–437). <https://doi.org/10.1109/ICSME.2016.33>.
- Fontana, F. A., Lenarduzzi, V., Roveda, R., et al. (2019). Are architectural smells independent from code smells? an empirical study. *Journal of Systems and Software*, *154*, 139–156.
- Fontana, F. A., Locatelli, F., Pigazzini, I., et al. (2020). An architectural smell evaluation in an industrial context. *ICSEA, 2020*, 68–74.
- Fowler, M., & Back, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc.
- Ganesh, S., Sharma, T., & Suryanarayana, G. (2013). Towards a principle-based classification of structural design smells. *Journal of Object Technology*, *12*(2), 1:1-29. <https://doi.org/10.5381/jot.2013.12.2.a1>
- Ge, X., Fang, C., Li, X., et al. (2025). Static code analyzer recommendation via preference mining. *Expert Systems with Applications*, *285*, 127861.
- Gnoyke, P., Schulze, S., & Krüger, J. (2024). Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells. *Journal of Systems and Software*, *217*, 112170.
- Heckman, S., & Williams, L. (2011). A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, *53*(4), 363–387. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing
- Herold, S. (2020). An initial study on the association between architectural smells and degradation. In *European Conference on Software Architecture (ECSA)*, (pp. 193–201).
- Hochberg, Y. (1988). A sharper bonferroni procedure for multiple tests of significance. *Biometrika*, *75*(4), 800–802.
- Hojo, T., & Pearson, K. (1931). Distribution of the median, quartiles and interquartile distance in samples from a normal population. *Biometrika*, 315–363
- Johnson, B., Song, Y., Murphy-Hill, E., et al. (2013). Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, (pp. 672–681)
- Jolak, R., Karlsson, S., & Dobsław, F. (2025). An empirical investigation of the impact of architectural smells on software maintainability. *Journal of Systems and Software*, *225*, 112382.
- Jorgensen, M., & Shepperd, M. (2006). A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, *33*(1), 33–53.
- Kim, J. H., & Choi, I. (2021). Choosing the level of significance: A decision-theoretic approach. *Abacus*, *57*(1), 27–7. <https://doi.org/10.1111/abac.12172><https://onlinelibrary.wiley.com/doi/abs/10.1111/abac.12172>
- Kim, S., & Ernst, M. D. (2007). Which warnings should i fix first? *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (pp. 45–54). New York, NY, USA: Association for Computing Machinery.
- Lenarduzzi, V., Martini, A., Taibi, D., et al. (2019). Towards surgically-precise technical debt estimation: Early results and research roadmap. *International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE, 2019*, 37–42.
- Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2019b). The technical debt dataset. In: *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. Association for Computing Machinery, New York, NY, USA, PROMISE'19, (pp. 2–11). <https://doi.org/10.1145/3345629.3345630>.
- Lenarduzzi, V., Sillitti, A., & Taibi, D. (2019c). A survey on code analysis tools for software maintenance prediction. In *Software Engineering for Defence Applications - SEDA 2018, Advances in Intelligent Systems and Computing (AISC)*, (vol. 925). Springer-Verlag.
- Lenarduzzi, V., Lomio, F., Huttunen, H., et al. (2020a). Are sonarqube rules inducing bugs? In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (pp. 501–511).
- Lenarduzzi, V., Mandić, V., Katin, A., et al. (2020b). How long do junior developers take to remove technical debt items? In *14th International Symposium on Empirical Software Engineering and Measurement (ESEM)*.
- Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2020c). Some sonarqube issues have a significant but small effect on faults and changes: a large-scale empirical study. *Journal of Systems and Software*, *170*.
- Lenarduzzi, V., Nikkola, V., Saarimäki, N., et al. (2021). Does code quality affect pull request acceptance? an empirical study. *Journal of Systems and Software*, *171*, 110806.
- Li, J., & Yang, J. (2024). Tracking the evolution of static code warnings: The state-of-the-art and a better approach. *IEEE Transactions on Software Engineering*, *50*(3), 534–550.

- Li, R., Liang, P., Soliman, M., et al. (2021). Understanding architecture erosion: The practitioners' perceptive. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, (pp. 311–322). <https://doi.org/10.1109/ICPC52881.2021.00037>
- Li, R., Liang, P., Soliman, M., et al. (2022). Understanding software architecture erosion: A systematic mapping study. *Journal of Software: Evolution and Process*, *34*(3), e2423.
- Macia, I., Arcoverde, R., Garcia, A., et al. (2012a). On the relevance of code anomalies for identifying architecture degradation symptoms. In *Conference on Software Maintenance and Reengineering (CSMR 2012)*, (pp. 277–286).
- Macia, I., Garcia, J., Popescu, D., et al. (2012b). Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In *International Conference on Aspect-oriented Software Development (AOSD '12)*, (pp. 167–178)
- Marcilio, D., Bonifácio, R., Monteiro, E., et al. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *27th International Conference on Program Comprehension (ICPC)*, (pp. 209–219).
- Martin, R.C. (1995). Object oriented design quality metrics: An analysis of dependencies. *ROAD*, *2*(3)
- Martin, R. C. (2007). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
- Martini, A., Arcelli Fontana, F., Biaggi, A., et al. (2018). *Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company: 12th European Conference on Software Architecture*, (pp. 320–335).
- Mende, T., & Koschke, R. (2009). Revisiting the evaluation of defect prediction models. In T. J. Ostrand, (ed.), *Proceedings of the 5th International Workshop on Predictive Models in Software Engineering, PROMISE 2009*, Vancouver, BC, Canada, May 18-19, 2009. ACM, (p. 7). <https://doi.org/10.1145/1540438.1540448>,
- Mumtaz, H., Singh, P., & Blincoe, K. (2021). A systematic mapping study on architectural smells detection. *Journal of Systems and Software*, *173*, 110885.
- Mumtaz, H., Singh, P., & Blincoe, K. (2021). A systematic mapping study on architectural smells detection. *Journal of Systems and Software*, *173*, 110885. <https://doi.org/10.1016/j.jss.2020.110885><https://www.sciencedirect.com/science/article/pii/S0164121220302752>
- Mumtaz, H., Singh, P., & Blincoe, K. (2022). Analyzing the relationship between community and design smells in open-source software projects: An empirical study. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, (pp. 23–33)
- Nagappan, N., & Ball, T. (2005.) Static analysis tools as early indicators of pre-release defect density. In *27th International Conference on Software Engineering (ICSE)*, (pp. 580–586).
- Nanda, M.G., Gupta, M., Sinha, S., et al. (2010). Making defect-finding tools work for you. In *32nd ACM/IEEE International Conference on Software Engineering - (Vol. 2, pp. 99–108)*.
- Nizam, A., Islamoğlu, E., Adali, Ö. K., et al. (2025). Optimizing pre-trained code embeddings with triplet loss for code smell detection. *IEEE Access*.
- Nord, R., Ozkaya, I., Kruchten, P., et al. (2012). In search of a metric for managing architectural technical debt. In *European Conference on Software Architecture (ECSA)*, (pp. 91–100).
- Oizumi, W., Garcia, A., Ferreira, M., et al. (2014). When code-anomaly agglomerations represent architectural problems? an exploratory study. In *Brazilian Symposium on Software Engineering (SBES)*, (pp. 91–100).
- Oizumi, W. N., Garcia, A. F., da Silva Sousa, L., et al. (2016). Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In *38th International Conference on Software Engineering, ICSE 2016*, Austin, TX, USA, May 14-22, 2016, (pp. 440–451). <https://doi.org/10.1145/2884781.2884868>
- Palomba, F., Zanoni, M., Fontana, F. A., et al. (2016). Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (pp. 244–255). <https://doi.org/10.1109/ICSME.2016.27>
- Pearson, K. (1895). Notes on regression and inheritance in the case of two parents proceedings of the royal society of London, *58*, (pp. 240-242). K Pearson
- Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52.
- Pigazzini, I., Foppiani, D., & Fontana, F. A. (2021). Two different facets of architectural smells criticality: An empirical study. In *ECSA (Companion)*
- Querel, L. P., & Rigby, P. C. (2018). Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (pp. 892–895).
- Rachow, P., & Riebisch, M. (2022). An architecture smell knowledge base for managing architecture technical debt. In *Proceedings of the International Conference on Technical Debt*, (pp. 1–10)

- Roveda, R., Fontana, F.A., Pigazzini, I., et al. (2018). Towards an architectural debt index. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*, (pp. 408–416).
- Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.
- Rutar, N., Almazan, C. B., & Foster, J. S. (2004). A comparison of bug finding tools for java. In *Symposium on Software Reliability Engineering*, (pp. 245–256).
- Saarimäki, N., Baldassarre, M. T., Lenarduzzi, V., et al. (2019). On the accuracy of sonarqube technical debt remediation time. In *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, (pp. 317–324).
- Saarimäki, N., Lenarduzzi, V., & Taibi, D. (2019). On the diffuseness of code technical debt in open source projects. In *International Conference on Technical Debt (TechDebt 2019)*.
- Samarthyam, G., Suryanarayana, G., & Sharma, T. (2016). Refactoring for software architecture smells. In *1st International Workshop on Software Refactoring, IWor@ASE 2016*, Singapore, Singapore, September 4, 2016, (pp. 1–4). <https://doi.org/10.1145/2975945.2975946>.
- Sas, D., & Avgeriou, P. (2023). An architectural technical debt index based on machine learning and architectural smells. *IEEE Transactions on Software Engineering*.
- Sas, D., Avgeriou, P., & Uyumaz, U. (2022). On the evolution and impact of architectural smells-an industrial case study. *Empirical Software Engineering*, *27*(4), 86.
- Schnappinger, M., Osman, M.H., Pretschner, A., et al. (2019). Learning a classifier for prediction of maintainability based on static analysis tools. In *27th International Conference on Program Comprehension*, (pp. 243–248).
- Simões, I. R. d. S., & Venson, E. (2024). Evaluating source code quality with large language models: a comparative study. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*, (pp. 103–113).
- Spearman, C. (1904). The proof and measurement of association between two things. *The American Journal of Psychology*, *15*(1), 72–101.
- Stephens, M. A. (1974). Edf statistics for goodness of fit and some comparisons. *Journal of the American statistical Association*, *69*(347), 730–737. <https://doi.org/10.1080/01621459.1974.10480196><https://www.tandfonline.com/doi/abs/10.1080/01621459.1974.10480196>
- Stephens, M. A. (2017). Tests based on edf statistics. In *Goodness-of-fit-techniques*. Routledge, (pp. 97–194).
- Suryanarayana, G., Samarthyam, G., & Sharma, T. (2015). *Refactoring for Software Design Smells* (1st ed.). Morgan Kaufmann.
- Taibi, D., Janes, A., & Lenarduzzi, V. (2017). How developers perceive smells in source code: A replicated study. *Information and Software Technology*, *92*(Supplement C), 223–235. <https://doi.org/10.1016/j.infsof.2017.08.008>, <http://www.sciencedirect.com/science/article/pii/S0950584916304128>
- Tempero, E., Anslow, C., Dietrich, J., et al. (2010). The qualitas corpus: A curated collection of java code for empirical studies. *APSEC, 2010*, 336–345. <https://doi.org/10.1109/APSEC.2010.46>
- Terra, R. M., Miranda, L. F., Valente, M. T., et al. (2013). Qualitas.class corpus: a compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, *38*, 1–4.
- Tollin, I., Fontana, F. A., Zannoni, M., et al. (2017). Change prediction through coding rules violations. *EASE'17*, (pp. 61–64).
- Tomas, P., Escalona, M. J., & Mejias, M. (2013). Open source tools for measuring the Internal Quality of Java software products. A survey. *Computer Standards and Interfaces*, *36*(1), 244–255.
- Trautsch, A., Herbold, S., & Grabowski, J. (2020). A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects. *Empir Software Engineering*, *25*, 5137–5192.
- Tu, H., Yu, Z., & Menzies, T. (2020). Better data labelling with emblem (and how that impacts defect prediction). *IEEE Transactions on Software Engineering*, 1–1. <https://doi.org/10.1109/TSE.2020.2986415>.
- Vassallo, C., Panichella, S., Palomba, F., et al. (2019). How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 1–39.
- Wagner, S., Jürjens, J., Koller, C., et al. (2005). Comparing bug finding tools with reviews and tests. In *International Conference on Testing of Communicating Systems*, (pp. 40–55).
- Wan, Z., Zhang, Y., Xia, X., et al. (2023). Software architecture in practice: Challenges and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (pp. 1457–1469).
- Wang, S., Liu, T., Nam, J., et al. (2020). Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, *46*(12), 1267–1293. <https://doi.org/10.1109/TSE.2018.2877612>
- Wohlin, C., Runeson, P., Höst, M., et al. (2012). *Experimentation in Software Engineering*. Springer.
- Wright, J. R., Alim K., & Do, L. N. Q. (2020). Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering (TSE)*.

- Xia, X., Lo, D., Pan, S. J., et al. (2016). HYDRA: massively compositional model for cross-project defect prediction. *IEEE Transactions on Software Engineering*, 42(10), 977–998. <https://doi.org/10.1109/TSE.2016.2543218>
- Yang, X., Chen, J., Yedida, R., et al. (2021). *Learning to recognize actionable static code warnings (is intrinsically easy)*. [arXiv:2006.00444](https://arxiv.org/abs/2006.00444).
- Yıldırım, R., Aydın, K., & Çetin, O. (2024). Evaluating the impact of conventional code analysis against large language models in api vulnerability detection. In *Proceedings of the 2024 European Interdisciplinary Cybersecurity Conference*, (pp. 57–64).
- Yu, T., Wen, W., Han, X., et al. (2019). Conpredictor: Concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering*, 45(6), 558–575.
- Zampetti, F., Scalabrino, S., Oliveto, R., et al. (2017). How open source projects use static code analysis tools in continuous integration pipelines. In *International Conference on Mining Software Repositories*, (pp. 334–344).
- Zheng, J., Williams, L., Nagappan, N., et al. (2006). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4), 240–253.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Matteo Esposito is a Postdoctoral Researcher at University of Oulu. His research focuses on the intersection of Large Language Models, Software Quality, Software Maintenance, and Software Architecture. Before transitioning to academia, he was R&D Vice Director at an Italian cybersecurity firm, leading innovation initiatives in secure systems development. He received his PhD, MSc, and BSc in Computer Science Engineering from the University of Rome “Tor Vergata”, Italy. He served as a program committee member, and organizing committee of various international conferences (e.g. ESEM, ECSA, and ESOC) and as a reviewer for various international journals (e.g., TSE, TOSEM, JSS, and IST) in the field of software engineering.



Mikel Robredo is a Doctoral Researcher at University of Oulu. His research focuses on the study of time dependent techniques and factors applied to Software Quality and Maintenance subfields such as Technical Debt and Refactoring Analysis. He focuses his research efforts into Data Science, Statistics as well as Data Mining applied to the field of Empirical Software Engineering. He received his MSc in Statistical Data Analytics from Tampere University, Finland, and his BSc in Economics with a Major in Econometrics from the University of the Basque Country, Spain.



Francesca Arcelli Fontana is currently a Full Professor with the University of Milano-Bicocca, Italy, where she is also the Head of the Software Evolution and Reverse Engineering Laboratory. Her research interests include software engineering, in particular software evolution and reverse engineering, architectural smell detection, program comprehension, software quality assessment, and managing technical debt.



Valentina Lenarduzzi is an Associate Professor (tenure track) at the University of Oulu (Finland). Her research activities are related to contemporary software development practices and methodologies, including data analysis in software engineering, software quality, software maintenance, and evolution, focusing on Technical Debt as well as code and architectural smells. She got the Ph.D. in Computer Science in 2015 and was a postdoctoral researcher at the Free University of Bozen-Bolzano, (Italy), at Tampere University (Finland), and LUT University (Finland). Moreover, she was visiting researcher at the University of Kaiserslautern (TUK) and the Fraunhofer Institute for Experimental Software Engineering IESE (Germany). She served as a program committee member of various international conferences (e.g., ICPC, ICSME, ESEM), and for various international journals (e.g., TSE, EMSE, JSS, IST) in software engineering. She has been the program co-chair of several conferences. Dr. Lenarduzzi is recognized by the Journal of Systems and Software (JSS) as one of the most active

SE researchers in top-quality journals from 2013 to 2020.

Authors and Affiliations

Matteo Esposito¹ · Mikel Robredo¹ · Francesca Arcelli Fontana² ·
Valentina Lenarduzzi¹

✉ Matteo Esposito
matteo.esposito@oulu.fi

Mikel Robredo
mikel.robredomanero@oulu.fi

Francesca Arcelli Fontana
francesca.arcelli@unimib.it

Valentina Lenarduzzi
valentina.lenarduzzi@oulu.fi

¹ University of Oulu, Oulu, Finland

² University of Milano-Bicocca, Milano, Italy