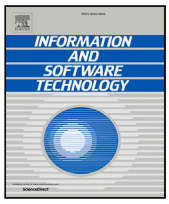
Information and Software Technology 185 (2025) 107783

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



An empirical study on architectural smells through a pipeline for continuous technical debt assessment

Matteo Bochicchio [a,*], Darius Sas [b], Alessandro Gilardi [a], Francesca Arcelli Fontana [a]

[a] University of Milano-Bicocca, Italy
[b] TXT Arcan, Italy

ARTICLE INFO

Dataset link: https://zenodo.org/records/13827946

Keywords:
Architectural debt
Architectural smells
Co-changes
Software maintenance
Software evolvability
Empirical study
Data collection pipeline

ABSTRACT

Context: Architectural smells, are a well-known indicator of architectural technical debt, their presence could have a great impact on the maintainability and evolvability of a project. Hence, it is important to carefully study and monitor them.
Objective: In this paper, we describe an empirical study on the analysis of the correlations existing between architectural smells and co-changes, with the aim of getting further insights into how architectural smells can influence maintenance efforts.
Method: Using the Goal-Question-Metric approach, we compared pairs of files affected by smells with clean ones to determine if smelly pairs co-change more frequently. To collect the data, we exploit a new data collection pipeline based on Apache Airflow to generate large-scale, up-to-date datasets with static analysis tools. For the current study, the pipeline uses Arcan 2, a static analysis tool for architectural smell detection.
Results: The empirical study, conducted on a set of projects analyzed by the pipeline, found that the median Co-change rate in smelly (both files affected) and mixed (one file affected) pairs was higher than in clean pairs. Moreover, the Co-change rate of the smelly pairs is higher than that of the mixed ones. This result became more significant as the lines of code increased.
Conclusion: The empirical study found that architectural smells are linked to higher Co-change rates in affected files, leading to increased maintenance efforts for developers. Moreover, the results highlight the value of the pipeline data and offer useful insights for managing architectural technical debt.

1. Introduction

The technical debt metaphor represents suboptimal design and implementation solutions that yield a benefit in the short term but introduce a technical context that makes future changes more expensive [1]. Researchers have identified different types of technical debt, depending on the software artifact or process it affects [2]. Ernst et al. found that the primary source of technical debt is related to architectural decisions [3]. This form of debt is known as *architectural technical debt*, which can be described more generally as the type of debt that makes software development costlier because of one or more suboptimal architectural solutions [2], which led to the presence of architectural smells [4].

In recent years, an increasing number of studies tried to better understand this phenomenon by employing a plethora of different research methodologies, with most studies adopting a quantitative data collection methodology based on static analysis [5]. In this paper, we aim to investigate the impact of architectural debt on software maintenance and evolution. In particular, we focus our attention on architectural smells [6,7] and study whether the presence of architectural smells correlates with an increase in the logical coupling [8] of the affected elements. An increase in logical coupling between two elements is detrimental to maintainability because it makes the coupled elements more likely to change together (a phenomenon known as a "co-change" [9]).

Similarly to architectural smells, co-changes are considered a sign of poor design as they expose a logical coupling between two files with a possible impact on run-time qualities, such as reliability, since co-changes are useful predictors of faults [8,9]. For our study the detection of architectural smells is done using Arcan 2.9.0 [10,11], an architectural technical debt estimation tool based on *architectural*

* Corresponding author.
E-mail addresses: matteo.bochicchio@unimib.it (M. Bochicchio), darius.sas@txtgroup.com (D. Sas), a.gilardi7@campus.unimib.it (A. Gilardi), francesca.arcelli@unimib.it (F.A. Fontana).

[1] See https://github.com/darius-sas/ccan-rs.

https://doi.org/10.1016/j.infsof.2025.107783
Received 19 December 2024; Received in revised form 8 May 2025; Accepted 12 May 2025
Available online 30 May 2025

smells detection. Whereas to calculate co-changes, we use our own implementation of previous approaches [9,12] which is publicly available online.[1] While, according to the data collected for our empirical study, we exploit a pipeline that we have defined to support also other future empirical studies. A common challenge that empirical quantitative studies face is represented by the threats to the external validity of the conclusions drawn from the data collected. A common solution to this problem is to increase the number of projects included in the dataset.

As a first step towards addressing this issue, our data collection pipeline is able to **continuously** fetch the latest updates from GitHub and compile them into ready-to-use data. The pipeline offers the following advantages over more traditional approaches: (1) it can scale to track and analyze thousands of repositories; (2) it can be configured to run with whatever static analysis tool desired by the researchers (tools that allow exporting the results); (3) the pipeline can automatically upload the resulting dataset to Zenodo for the research community to freely use. The key purpose of the pipeline is to utilize up-to-date data at the time of analysis, ensuring that findings are reflective of current practices, technologies, and frameworks. Therefore, as long as the pipeline is maintained and continues to incorporate contemporary data, the results should remain applicable and valuable for future software development contexts.

Hence, the goal of this paper is two-fold: (1) describe an empirical study on the correlations between architectural smells and logical coupling, and (2) propose a new data collection pipeline. In the paper, we first describe the tool Arcan used to detect the AS and then the pipeline that we used to collect the data for the empirical study. Moreover, through the pipeline,[2] we produced a large-scale dataset of architectural smells,[3] a dataset that, assuming the availability of enough cloud resources, can be automatically and continuously updated as the software systems it contains evolve. The dataset produced with the pipeline contains over 30,000 commits and releases analyzed with Arcan 2.9.0 amounting to over a billion lines of code analyzed in total and almost 7 million of architectural smells instances detected. It is important to note that, while we used Arcan 2.9.0, the pipeline can be adapted to obtain similar results with other tools such as for example Designite [13], SpotBugs, PMD,[4] or other static analysis tools which allow exporting the data.

Our long-term goal is to build a **streamlined approach** to data collection for empirical software engineering studies. More specifically, we want to provide researchers with a comprehensive corpus of *readily available* up-to-date static analysis data and reduce the friction of performing large-scale data collection. For this reason, besides the dataset, the pipeline is publicly available online to allow researchers to replicate the data collection pipeline instance on their machines and even use their own tool.

The remainder of the paper is organized as follows: Section 2 briefly summarizes related work; Section 3 explains how Arcan 2.9.0 operates and the metrics it calculates; Section 4 introduces the design of the pipeline; Section 5 describes the study design with the Research Question and the results of the empirical study; Section 6 describes the limitations and threats to validity of this work; and, finally, in Section 7 we discuss the main results and outline some future developments.

2. Related work

In this section, we briefly present related works on empirical studies on Architectural Smells (AS), works on co-changes and their relationships with other issues and works on pipelines or datasets proposed in the literature for benchmark purposes or other investigations in the software quality assessment area.

2.1. Empirical studies on architectural smells

One of our first interest is to further understand the evolution of AS. Some previous works investigated the evolution of AS [14,15], on systems from the Qualitas Corpus [16], but the insights from such studies are limited to software written in Java.

Sas et al. [17] provide a quantitative study on how individual architectural smell instances evolve over time in 9 C/C++ projects, how long they typically survive within a project and how they overlap with instances of other smell types. The authors interview 12 among the developers and architects working on these projects and provide insights on the effects of AS on the long-term maintainability and evolvability of the projects.

Other studies analyzed the correlations between AS and design patterns [18], the correlations between AS and code smells [19], between design and architectural smells [20], and correlations between AS and degradation manifested as architecture-violating dependencies [21]. No one of these studies considered the correlations between AS and co-changes.

While in a previous work [22] we studied whether the frequency and size of changes are correlated with the presence of a selected set of AS. The findings, based on a case study of 31 open-source Java systems, show that 87% of the analyzed commits present more changes in artifacts with at least one smell, and the likelihood of changing increases with the number of smells. Additionally, there is also evidence to suggest that change frequency increases after the introduction of a smell and that the size of changes is also larger in smelly artifacts.

2.2. Correlation between co-changes and issues in software projects

The essential part of our paper regards an empirical study on the correlation between co-changes and architectural smells. Hence, the impact that co-changes have on software projects as an indicator of potential issues is certainly of great interest and we outline below some of the works proposed in the literature.

Jaafar et al. [23] investigated the impact of design pattern and anti-pattern dependencies on changes and faults. They found that classes having dependencies with anti-patterns are more fault-prone than others, and that structural changes are the most common changes impacting classes having dependencies with anti-patterns. The authors also found that co-change dependencies with anti-pattern and design pattern classes have similar negative impacts on changes and faults, and that classes with dependencies on anti-pattern classes are more subject to logic faults and structural changes. They concluded that the knowledge about the propagation of faults through static and co-change dependencies is useful to improve the prediction of faults in classes and should be studied further in the future.

Kouroshfar et al. [9] studied the impact of co-changes on software quality. The authors found that co-changes that crosscut multiple architectural modules are more likely to introduce bugs than co-changes that are localized within a single module. This suggests that co-changes involving several architectural modules should be given more attention during bug fixing, as they are more likely to be the source of errors.

Palomba et al. [24] proposed a novel approach to detect code smells by leveraging change history data extracted from versioning systems. Their research demonstrated that co-changes between source code artifacts can effectively reveal five common code smells: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Furthermore, the study highlighted the potential of co-change analysis to pinpoint methods that exhibit a stronger affinity with a different class than the one they are currently part of. Ultimately, the analysis presented in the study underscores the value of co-change analysis as a powerful technique for identifying code smells and enhancing software quality.

Sas et al. [12] investigated co-changes and their relation to AS, as proxies of reliability and maintainability. They analyzed 14 open-source projects and used two algorithms to detect the co-changes,

[2] The source code of the pipeline is publicly available at https://github.com/Arcan-Tech/arcan-pipeline.

[3] See https://zenodo.org/records/10149834.

[4] Available at https://spotbugs.github.io/ and https://pmd.github.io/.

which they merged with the architectural smell data to create the data set. They found that 50% of co-changes take part in an architectural smell, and in 90% of cases that co-change tends to appear before the introduction of the smell. In the empirical study presented in this paper, we perform a study on the correlations between AS and co-changes on a much larger dataset and using a different data analysis approach. Moreover, we consider C# projects on top of Java projects in order to strengthen the external validity of our findings. Lastly, to strengthen construct validity, we also take into consideration the possible confounding effect of the lines of code variable.

2.3. Datasets and pipelines for software quality analysis

According to other datasets, Tempero et al. [16] introduced the first curated dataset of software projects in 2010, naming it the Qualitas Corpus. This dataset contained both the sources and compiled versions of 100 systems and 495 of their releases. The projects of the Qualitas Corpus are written in Java, while our dataset allows researchers to study architectural smells in a large number of projects written in 5 different programming languages. On the other hand, we do not include the source code of the projects, but rather we provide the results and analyses of a specific tool (i.e. Arcan) on the projects in our corpus, and do so at a much bigger scale.

Couto et al. [25] presented the COMETS dataset consisting of 17 object-oriented metrics, ranging from traditional size-based metrics (i.e. lines of code) to the well-known CK metrics [26], for 10 different open source Java systems. The metrics were collected every two weeks for each project for a period of 400 weeks. The metrics contained in our dataset partially overlap with the ones calculated by Couto et al. [25]. The main differences are the scale of the analysis, the fact that our dataset also contains the architectural smells detected and that we include multiple programming languages.

Palomba et al. [27] created a dataset of 243 manually validated instances of code smells extracted from 20 open-source Java systems on top of presenting a web-based platform to share and access such a dataset. The main difference from our dataset is that they focus on code smells rather than architectural smells as we do, as they are two distinct concepts and are also uncorrelated with each other [19].

Lenarduzzi et al. [28] produced a curated dataset of SonarQube analyses of 33 Java projects from the Apache Software Foundation. Additionally, they linked this information with code smells data (using the Ptidej tool [29]) and issues and faults reported in the respective issue trackers of each project. A total of 78k commits were analyzed, 1.6M SonarQube issues were found, 68k code smells were detected, 28k faults were isolated and 57k refactorings were identified. Our work differs from Lenarduzzi et al.'s in the fact that we focus on architectural smells (as we use a different tool) and that our dataset contains more projects (13,451) but fewer commits (34,983). Additionally, we also do not link architectural smells to issue trackers, although it could be interesting for future work.

Diamantopoulos et al. [30] presented a dataset with the 3000 most popular Java projects at the time containing metrics and code violations detected by PMD tool. Moreover, they also provide development-related metrics such as the number of contributions by each collaborator, the number of issues, etc. The primary differences from our work are that (1) we focus only on static analysis of source code, (2) we analyze more projects and more programming languages, and (3) we focus on architectural smells rather than the code-level violations calculated by PMD.

Thakur et al. [31] introduced a data collection platform that analyses GitHub repositories using the Designite tool. They analyzed over 12,000 repositories and 200 million lines of code of Java and C# projects. Additionally, their platform provides an easy-to-use web interface to navigate and visualize the results of the analyses for each project. Our work differs from Thakur et al.'s in the different tools selected for analysis (Arcan and Designite) which makes the two datasets complementary. Additionally, our dataset also includes C, C++, and Python projects on top of making available the dependency graphs of the analyzed projects.

Sharma et al. [32] further extended the work of Thakur et al. [31] and compiled a dataset with results of Designite for 86,000 C# and Java projects. Since Sharma et al.'s work is a continuation of Thakur et al.'s, the same differences from our work that are mentioned above hold.

Moreover, many tools can be used for mining data from repositories on GitHub. Gousios et al. [33] introduced GHTorrent as a tool for mining and analyzing data from GitHub, providing researchers and developers access to a comprehensive, queryable dataset of repositories, commits, issues, pull requests, enabling empirical studies in software engineering and collaborative development. Spadini et al. [34] developed PyDriller, a Python framework for mining software repositories, offering a simple and efficient interface to extract and analyze data from version control systems, enabling researchers and practitioners to conduct empirical studies in software engineering. Zhong et al. [35] presented MAPO (Mining API Usages from Open Source Repositories), a tool designed to automatically extract and analyze API usage patterns from open-source projects, helping developers better understand common practices and improve API utilization in software development.

The novelty of this work compared to the previous ones is that we designed and implemented an open-source pipeline that can produce technical debt datasets that are readily available and always up to date. Additionally, the pipeline can be easily adapted to use other tools than the one adopted in this paper, thus providing value also for researchers that do not study architectural smells specifically.

3. Data collected through the arcan 2 tool

3.1. Features

Arcan 2.9.0 is a publicly-available[5] tool for automatic technical debt detection through static source code analysis initially published by Arcelli et al. [10]. Different types of technical debt exist [2], Arcan focuses specifically on architectural technical debt by detecting architectural smells [6,7], a symptom of the presence of technical debt and one of the best available approach for identifying architectural weaknesses. As highlighted in Table 1, four AS types are detected by Arcan 2.9.0:

- **Hublike Dependency (HL)**: this smell arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions [36,37].
- **Cyclic Dependency (CD)**: refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem's dependency structure.
- **Unstable Dependency (UD)**: describes a subsystem (component) that depends on other subsystems that are less stable than itself, according to the Instability metric value [38], with a ripple effect of changes in the system [37].
- **God Component (GC)**: This smell represents a component (or package, in Java) that is considerably larger in size (i.e. lines of code) than other components in the system [17].

Our study focuses on these four specific AS, since both God Component and the other three AS based on dependency issues, have a great impact on maintainability and architectural debt, as also outlined in previous studies by Le et al. [39] and Mo et al. [40]. Moreover, we focused our attention on these AS since the tool we used could detect only these four AS. In future work, we could also consider other types of AS.

[5] See https://github.com/Arcan-Tech/arcan-2-cli-trial.

For each AS type, Arcan 2.9.0 can detect multiple instances of the smell, and for each of these instances, it calculates a number of different properties called *smell characteristics* [14] (see Table 1). Arcan 2.9.0 also computes a set of software metrics and *estimates* the amount of *technical debt principal* in the system as an index [11]. Technical debt (TD) is estimated as the (weighted) sum of the technical debt index of each architectural smell detected in the system. The system's index is compared with the technical debt index of all the systems in a reference benchmark [11] to compute a *rating*, allowing users to have a comparison of how a system relates to the other systems in the benchmark. The rating is simply calculated as follows:

$$rating(P) = \begin{cases} A & .8 \leq score(P) \leq 1 \\ B & .6 \leq score(P) < .8 \\ C & .4 \leq score(P) < .6 \\ D & .2 \leq score(P) < .4 \\ F & 0 \leq score(P) < .2 \end{cases} \quad (1)$$

where the $score(P)$ is the percentile of projects in the benchmark that has a *higher* technical debt density [11] than the given project P, with A being the best score and F the worst.

3.2. Output graph files

To better understand the internal workings of the pipeline described in the next section and how the output dataset is created, we now describe the output generated by Arcan 2.9.0. The tool uses a graph data structure to represent source code artifacts and store the results of processing those artifacts. Such graph is called *dependency graph* and follows the schema depicted in Fig. 1 and in Fig. 2, where nodes represent source code artifacts like packages, classes, and functions; and edges represent the relationships among nodes, such as membership between classes and packages and dependencies between classes. In particular, nodes are of four types:

1. **units**, which represent compilation units, such as classes or files, depending on the programming language;
2. **containers**, which represent elements that contain similar elements or units. Examples are packages, namespaces, and folders, depending on the programming language;
3. **functions**, which represent programming constructs that group individual instructions. These are primarily functions and methods.
4. **smells**, which represent architectural smells. The smells dependency graph follows the schema depicted in Fig. 2.

Containers, units, and functions are all **components**, and share some common properties, such as the qualified `name`, the non-qualified name `simpleName`, and the path on the file system `filePathReal`. Other properties stored within the nodes are all the metrics listed in Table 1 that have a granularity different than "Project".

Concerning edges, there are several types of edges, but the two most important ones are: **membership edge**, which represents the membership between components, for example, to what package a class belongs to; and **dependency edge**, that represents the dependencies between components, for example, a class that has a field of a particular type has a dependency towards the class of that type. Pruijt et al. define what is a dependency in Java [45], and Arcan 2.9.0 adheres to such standard.

The output dataset is the flattened version of the graphs outputted by Arcan 2.9.0, namely each node or edge is a row in the database, while properties are the columns. Further details are presented in Section 4.2 and in Appendix.

Table 1
The TD principal metrics, list of AS, smell characteristics, and software metrics calculated by Arcan 2.9.0. *Note*: we only use Java-specific terminology (e.g. Package) for brevity's sake, the metric applies to the respective counterparts of other programming languages. All the details of the metrics are available in the referenced literature cited in the table.

Name	Granularity	Domain	Ref.
Technical debt principal			
Arch. tech. debt index (ATDI)	Smell, Project	$\mathbb{N}$	[11]
Arch. tech. debt hours	Smell, Project	$\mathbb{N}$	[11]
Arch. tech. debt score	Project	$[0, 1]$	[a]
Arch. tech. debt rating	Project	A, B, C, D, F	[a]
Architectural smells			
Hublike Dependency (HL)	Package, Class	–	[37]
Cyclic Dependency (CD)	Package, Class	–	[37]
Unstable Dependency (UD)	Package	–	[37]
God Component (GC)	Package	–	[17]
Smell characteristics[b]			
Affected component type	All smells	[c]	[14]
Mean depth of containment tree	All smells	$\mathbb{R}$	[11]
Mean distance of containment tree	All smells	$\mathbb{R}$	[11]
Mean PageRank affected components	All smells	$[0, 1]$	[41]
Severity	All smells	$(0, 10)$	[11]
Size	All smells	$\mathbb{N}$	[14]
Number of edges	HL, CD, UD	$\mathbb{N}$	[14]
Affected classes ratio	HL smell	$[0, 1]$	[14]
Afferent affected ratio	HL smell	$[0, 1]$	[14]
Efferente affected ratio	HL smell	$[0, 1]$	[14]
Instability gap	UD smell	$[0, 1]$	[14]
Strength	UD Smell	$[0, 1]$	[37]
Central component	GC smell	*Package name*	[14]
Lines of code density	GC smell	$\mathbb{R}$	[11]
Shape	CD smell	[d]	[37]
Software metrics			
Abstractness	Package	$[0, 1]$	[42]
Instability	Package	$[0, 1]$	[42]
Distance from main sequence	Package	$[0, 1]$	[42]
Fan In	Package, Class	$\mathbb{N}$	[42]
Fan Out	Package, Class	$\mathbb{N}$	[42]
Lines of code	Package, Class	$\mathbb{N}$	–
Change has occurred (CHO)	Package, Class	$\{F, T\}$	[43]
Code churn (TACH)	Package, Class	$\mathbb{N}$	[43]
PageRank	Package, Class	$[0, 1]$	[41]
Coupling between objects	Class	$\mathbb{N}$	[26]
Depth of inheritance	Class	$\mathbb{N}$	[26]
Number of children	Class	$\mathbb{N}$	[26]
Number of architectural smells	Project	$\mathbb{N}$	–
Number of components	Project	$\mathbb{N}$	–

[a] This study.
[b] Also known as smell properties.
[c] $\{Package, Class\}$.
[d] $\{Tiny, Circle, Chain, Star, Clique, Unknown\}$ [44].

Table 2
The programming languages supported by Arcan 2.9.0 and the implementation technologies of the respective parsers.

Programming language	Supported versions	Parsing technology
Java	Java 18 and earlier	Java library (Spoon[a])
C	C17 and earlier	Java library (Eclipse CDT[b])
C++	C++ 11 and earlier	Java library (Eclipse CDT[b])
C#	.NET 6 (via Mono)	LSP (OmniSharp[c])
Python	Python 3.11 or earlier	LSP (Pyright[d])

[a] https://github.com/INRIA/spoon.
[b] https://projects.eclipse.org/projects/tools.cdt.
[c] https://github.com/OmniSharp/omnisharp-roslyn.
[d] https://github.com/microsoft/pyright.

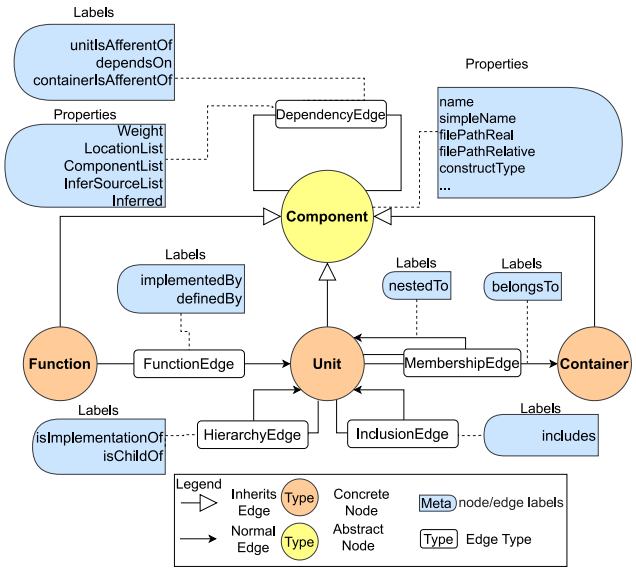


Fig. 1. Components dependency graph schema.

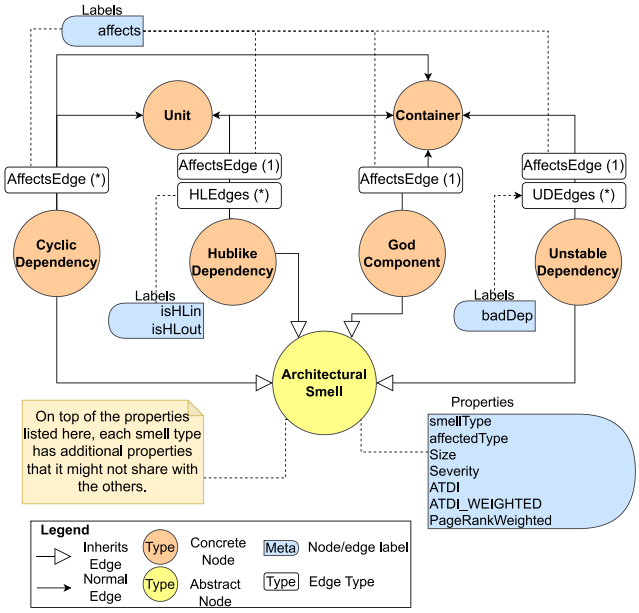


Fig. 2. Smells dependency graph schema.

4. The data pipeline

4.1. Pipeline design overview

Our proposed solution is a cloud-based data pipeline able to check a set of GitHub repositories for new releases (or commits, if no releases are available) and **automatically schedules** an analysis with Arcan 2.9.0 before saving the results in a database.

The high-level design of the pipeline is depicted in Fig. 3. We implemented the pipeline using Apache Airflow[6] to orchestrate the execution of the necessary tasks. Tasks are organized into three workflows: **Ingestion** (tasks 1 through 3), **Execution** (tasks 4 through 7), and **Generation** (tasks 8 through 10).

[6] See https://airflow.apache.org/.

At a high level, the workflows function as follows. *Ingestion* is responsible for acquiring the latest *version* of a project daily; a version could either be a release or a commit if no release is available. *Execution* is responsible for continuously executing Arcan on each non-analyzed version twice: first, Arcan *parses* source code to build the dependency graph of the given version, then Arcan detects architectural smells, computes the software metrics, and estimates technical debt. *Generation* is responsible for generating a unified dataset starting from the results of the previous phase before publishing such dataset on Zenodo[3].

The upcoming three sections explain the three workflows in detail.

4.1.1. Ingestion workflow

At its core, the *Ingestion* workflow monitors a list of GitHub projects and saves newly acquired releases in the pipeline's analysis queue. This initial list of projects monitored by the pipeline was created using previous work from the literature [46,47], which contained over 130,000 projects. However, not all of them were suitable for our purpose, so we *filtered out* the projects:

1. with less than 5,000 lines of code, to include small projects too, some of which may evolve and grow in the future;
2. with less than 10 stargasers[7];
3. not written in languages supported by Arcan 2.9.0 (i.e. Java, C, C++, Python, or C#. See Table 2).

For the calculation of LOC (filter 1), we only consider code files written in the programming language under analysis. Files written in languages irrelevant to the analysis, such as HTML or JSON, are not included.

After applying filters (1) and (2) we were left with 32,299 projects; applying filter (3) reduced this number to **13,451 projects**.[8] Note that, these selection criteria are intentionally broad to include as many projects as possible. Researchers who desire to use the dataset can always opt to apply stricter criteria when using the data.

After acquiring the projects, *Ingestion* queries GitHub's APIs to retrieve the latest metadata of each project in order to select the next versions to analyze. A particular problem faced during this stage is how to sample the releases of a project given that release schedules greatly differ from project to project. Therefore, we analyzed the 32,299 projects in our dataset and found that (1) of the 151,453 GitHub releases 45% of the projects have no GitHub release; (2) releases are published, on average, once a month; (3) some projects release several times a day. Therefore, to mitigate the differences between the two extreme cases (i.e. occasional releases and daily releases), given n newly detected releases, we sample one release for each repository every:

$$sampleStep(n) = \left\lceil \frac{n-1}{maxRelLimit - 1} \right\rceil \tag{2}$$

$$maxRel(n) = \lfloor 6 \cdot log_{10}(n+1) \rfloor \tag{3}$$

The function $sampleStep(n)$ determines the step size for sampling project releases, ensuring a uniform selection even in projects with a high number of releases. It calculates how frequently a release should be analyzed based on the total number of available releases and a predefined upper limit ($maxRelLimit$). The function $maxRel(n)$ sets the maximum number of releases to process per project in a single run. Its logarithmic growth prevents overloading the analysis pipeline while ensuring that representative data from frequently updated projects is still captured. Note that GitHub's APIs enforce $n \leq 1000$ (most recent only), therefore, we only ingest into the pipeline at most $maxRel(1000) = 18$ releases for a project in a single run, 1 release every $sampleStep(1000) = 59$. Fig. 4 depicts a plot of the number of releases ingested given the number of newly detected GitHub releases n. Finally, for projects that exhibit no releases, we ingest one commit a month.

[7] GitHub repositories can be *starred* by GitHub users. The number of stars is a proxy metric for estimating the popularity of a repository.

[8] The full list is available in the replication package [48].

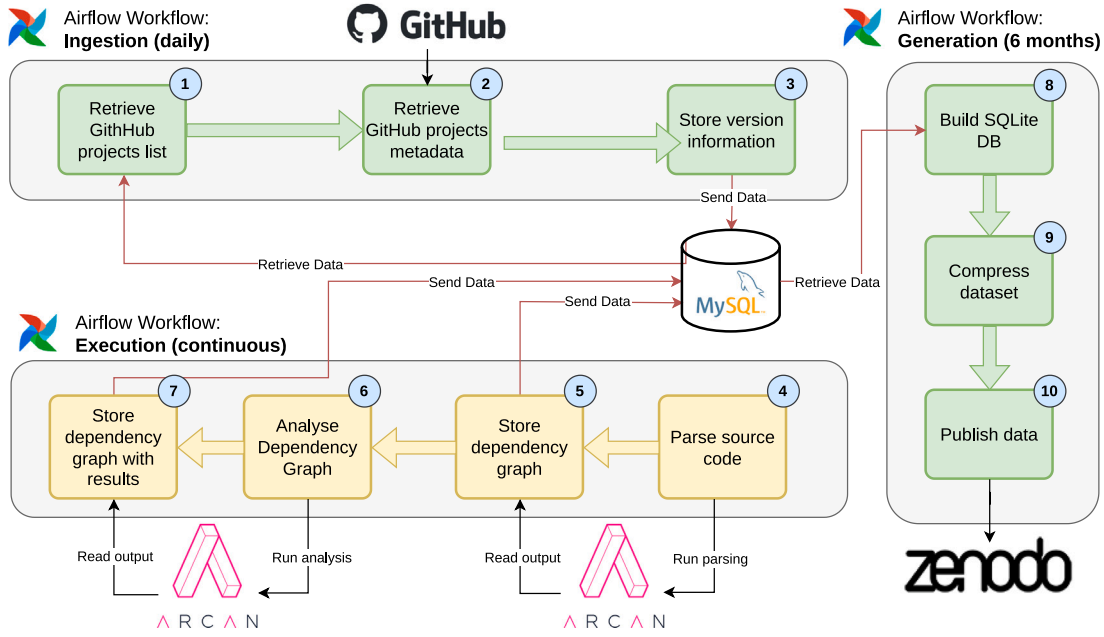


Fig. 3. The design of the data pipeline implemented by the two Apache Airflow Workflows: *Ingestion* and *Execution*.

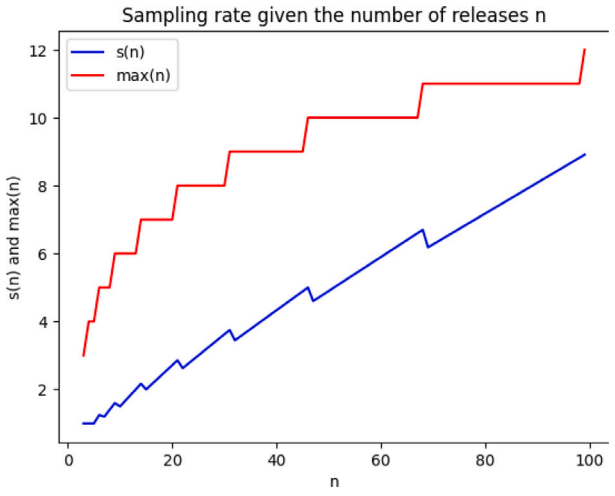


Fig. 4. The number of ingested new GitHub releases n for a given project.

4.1.2. Execution workflow

The *Execution* workflow begins by dequeuing the versions imported by the *Ingestion* workflow. These versions are analyzed in parallel using multiple Docker containers of Arcan 2.9.0.

As mentioned before, the analysis splits into two phases: (1) **parsing** of the source code to create the dependency graph (tasks 4 and 5 in Fig. 3) and (2) **analysis** of the dependency graph to detect smells and calculate metrics (tasks 6 and 7 in Fig. 3). Note that the analysis phase stores its results within the dependency graph itself. Graphs can later be accessed and transformed in any desired format. The output of both phases is a dependency graph abiding the schema described in Section 3.2 and both graphs are encoded in the GraphML [49] format.

The workflow has been executed continuously and analyses run in batches of $N = 6$ versions in parallel, with each instance of Arcan 2.9.0 analyzing one version. On average, 659.3 versions have been analyzed each day, with an average failure rate of 11.64%, which highly differs based on the programming language of the project, as depicted in Table 3. Projects written in C# exhibit the highest failure rate because they depend on Windows-exclusive libraries, thus the language server parser cannot properly load them, resulting in errors. This is because our pipeline deployment and Arcan 2.9.0 run on Linux-based Docker containers and a Linux host. Ultimately, 88% of versions are successfully analyzed and we deem this an acceptable rate.

Table 3
Failure rates of the Arcan 2.9.0 analyses by programming language.

Language	Total analyzed	Success	Failed	Success rate (%)	Fail rate (%)
C	3459	3113	346	90.0	10.0
C++	7398	6385	1013	86.3	13.7
C#	6149	4125	2024	67.1	32.9
Java	11,210	10 385	825	92.6	7.36
Python	11,377	10 975	402	96.5	3.53
Total	**39,593**	**34,983**	**4610**	**88.35**	**11.64**

4.1.3. Generation workflow

The *Generation* workflow is pretty straightforward. It begins by creating the SQLite database file (task 8 in Fig. 3) that contains the output of Arcan in a table-like format (as opposed to plain graph files). Then, it starts compressing this file (task 9 in Fig. 3) so as to save as much disk space as possible. Finally, a compressed version of the dataset can be uploaded to Zenodo (task 10 in Fig. 3) so that it can be used by other researchers.

4.1.4. Deployment and implementation

The pipeline is deployed as a dockerised Airflow application.[9] The main rationale behind using Apache Airflow to implement the pipeline is that Aifrlow workers can be distributed on multiple machines, assuming that all tasks operate without a local state (e.g. local files). Given that our pipeline follows this rule, we can eventually scale it to analyze even more versions by distributing the workers on multiple machines. We outline that the pipeline can be easily extended to automatically add new projects.

[9] The application runs on a single machine with 8 vCores and 30 GB of RAM (model B2-30 OVH cloud) running Ubuntu 22.04, and connects to a managed MySQL 8 database.

Table 4
Total number of architectural smells detected in all versions analyzed.

Programming language	Cyclic dependency	God component	Hublike dependency	Unstable dependency
C	1,265,281	5387	14,456	2198
C++	1,335,863	7000	20,787	4860
C#	1,786,759	2323	13,686	3895
Java	1,862,343	2763	39,110	18,915
Python	559,156	2213	21,423	0
Total	**6,809,402**	**19,686**	**109,462**	**29,868**

4.2. A dataset for benchmark studies

Here we provide some details on the first dataset created by the execution of the pipeline. Other details on accessing and using the dataset are given in Appendix. Moreover, the pipeline has been exploited to create another dataset used in the empirical study described in the next section. The dataset contains 34,983 unique versions (either releases or commits) belonging to 10,224 unique projects. The total number of lines of code (measured by counting all the non-commented and non-empty lines in a source code file) analyzed is 1.2 billion, with an average of 35,007 per version. The dataset contains 6,968,418 AS *non-unique* instances. They are non-unique because some smells may be the same instance but detected separately in two different versions. Table 4 dissects the total number of instances by type and different programming language. Cyclic Dependencies are the most prominent type of AS in terms of number of instances. On the other hand, God Components are the least common smell, mostly because they are a size-based smell and large components are rare. It is also interesting to note that no Python project has an Unstable Dependency smell. The result stems from the fact that many Python projects are written in a procedural style rather than an object-oriented one. As such, the metrics used to calculate Unstable Dependency smells (defined by Martin et al. [38]) are less meaningful in these cases.

The first version of the dataset is freely available in the replication package [48]. Future updated versions will be publicly available on Zenodo[3].

5. Empirical study on the relationship between architectural smells and co-changes

The data derived from the implementation of the data collection pipeline, and the pipeline itself, open the door to a many research opportunities and studies, as the one described in this section.

5.1. Goal and research questions

This study aims to understand the impact of architectural smells on source code changes. Using the Goal-Question-Metric approach [50]. The goal can be formulated as follows:

> *"Study the relationships between architectural smells and co-changes to investigate whether pairs of files affected by architectural smells tend to co-change more frequently than clean files in software projects".*

Which can be summarized into the following research question:

***Research Question**: Do pairs of files affected by architectural smells exhibit a higher chance of co-changing than clean files?*

Answering this research question allows us to understand whether the presence of an architectural smell is linked with an increased likelihood of a pair of files to co-change, which suggests that the presence of smells increases the effort required to maintain the two files with respect to a pair of files that are not affected by a smell (i.e. changing a file is likely to need a change in another file too).

5.2. Analyzed projects

We used the pipeline described in the previous section to collect all the AS detection and co-changes data.

Through the pipeline, we collected data on 5,000 projects. To conduct our study, we had to filter the projects analyzed by the pipeline to perform our analyses on a reduced number of projects. This is because the computation required for calculating co-changes is highly expensive.

We used the following criteria to select the projects to be analyzed for our study:

- Only Java and C# projects. We only used these two languages because they are both fully object-oriented languages[10] where logical coupling is more likely to emerge, thus mitigating construct validity [51].
- Projects with a number of stargazers greater than or equal to 100 on GitHub. This was done to ensure there are sufficient changes for our analysis and that the project is actively being used.
- Total number of Lines of Code (LOC) greater than or equal to 10,000, and more than 10 files. This was done to ensure there were enough architectural smells.

We chose these quantitative thresholds (e.g., for LOC, number of files) as heuristics. Similar values have been used in previous studies found in the literature [22]. After performing this initial filtering, we obtained a population of 944 projects. From these projects, in order to reduce the co-change computation time (computationally expensive), we first filtered out toy, demo, and student projects by parsing the repository names based on a set of keywords: "demo", "exam", "example", "test", "sample", and eliminated projects containing these keywords. Then, we manually checked the remaining projects to ensure that demo or example projects, which could introduce noise into the analysis, were excluded. As a final step, we randomly sampled **118 projects** (12.5%).

It is important to emphasize that this empirical study is also an example of using the dataset derived from the pipeline for research purposes. The empirical study is based on the dataset published on Zenodo after the last execution of the pipeline. Performing the statistical analysis and calculating the co-changes are extremely resource-intensive tasks, hence, project selection was necessary. The researchers interested in using our dataset can focus on a portion of the dataset according to their research aims.

Table 5 shows the demographics of the analyzed projects, including the median values of LOC, the number of file pairs per project, and the number of versions per project. The LOC values range from a minimum of 10,105 to a maximum of 929,886. The number of pairs per project ranges from a minimum of 12 pairs to a maximum of 5,457,822 pairs. The number of versions per project ranges from a minimum of 21 to a maximum of 3,458. The projects have an average stargazer count of 2,749.8 for Java and 1,366.5 for C#. Therefore, our dataset includes projects of various sizes for comparison.

5.3. Collected variables

The data on AS was collected as described in Section 4.2, then we filtered out all smells with an Architectural Technical Debt Index (ATDI) smaller than 10, so as to ignore smaller AS. Here too, we used

[10] In Java and C#, programming is not procedural, whereas in C++, it can be. Although C++ supports object-oriented programming through classes, developers can write code without using classes, allowing for a more procedural approach if desired. In contrast, Java and C# enforce an object-oriented paradigm, meaning that every piece of code must be encapsulated within a class. This makes it impossible to avoid using classes in Java and C#, as they are fundamental to the structure of the language.

Table 5
Demographics of the projects analyzed in this study.

	Java	C#
Projects count	60	58
LOC median	37,787	44,283
Pairs number median	11,631	9090
Median # of versions	361	446

Table 6
Lines of code (LOC) quartiles and corresponding categories.

LOC values	Category
$10 < LOC \leq 32$	XS
$32 < LOC \leq 56$	S
$56 < LOC \leq 112$	M
$112 < LOC \leq inf$	L

this threshold for ATDI as a heuristic. Similarly, we removed files with less than 20 lines of code, thereby eliminating all trivial classes.

To address the RQ, we focus on whether there is a statistical difference between files with AS that co-change often as opposed to non-affected files. In summary, for *every pair of files* in a project we collect the following variables:

- **Number of architectural smells**: number of AS in each pair file (as two separate variables).
- **Smell-Affected**: this variable only has three possible values, depending on whether *one*, *two*, or *no files* in the pair are affected by at least one smell with an ATD index higher than 10. This is our *independent variable*.
- **Co-change**: number of times the two files were modified simultaneously [9]. In this study, we grouped commits by development day to ensure that we count as a co-change also files that, while not necessarily change in the same commit, they do change together in a close time-span. This is the proxy variable we use for measuring logical coupling [8]. Logical coupling refers to the relationship between software components that often change together, despite not being structurally dependent. In fact, two entities are close to each other (logically coupled) when they are frequently changed together.
- **Co-change rate**: number of times a pair co-changes divided by the number of versions. This represents the measured probability of a co-change happening in the project. We consider this our *dependent variable*.

We calculated the number of co-changes for each pair of files in the projects selected in the previous section, extracting the information available from their GitHub repositories. The co-change counting process was performed automatically using R scripts, which are included in the replication package [48]. For the calculation of co-changes, we considered only the main branch (e.g., main or master). Furthermore, in our study, we extract data from all commits within a single branch, including the initial commit. The presence of the initial commit and potential large merges do not affect our analysis because the number of commits considered per project is substantial.

To control for the confounding effect of the lines of code variable, we classified each file based on the LOC value. The thresholds used for the categories correspond to the quartiles of the LOC distribution. By doing so, we divided the files into four distinct populations, as described by Table 6, and repeated the analyses for each group of files.

To show whether LOC is indeed a confounding variable, we divided the population of file pairs based on the LOC category. For each pair, we compared the LOC category of the two files, then the LOC category of the pair corresponds to the higher LOC category of the two files (same categories at Table 6). For example, if the first file is in LOC category XS and the second file is in LOC category M, then the pair is classified as LOC category M.

The data operations were automated using Python scripts. We include these scripts in the replication package [48].

5.4. Data analysis

In order to answer the RQ, we use the *Smell-Affected* variable to divide the pair of files into three different groups: **smelly pairs** (when both files are affected by a smell), **mixed pairs** (when only one file is affected by a smell), and **clean pairs** (when no file is affected by a smell).

It is worth noting that smelly pairs refer to pairs of files affected by architectural smells, where both files may be impacted either by instances of the same smell type or by different smell types. Architectural smells introduce structural or dependency weaknesses that propagate maintenance challenges across the affected files, even when the smells originate from different instances, thereby amplifying the effort required to evolve or modify both files together.

5.4.1. Statistical tests

To answer the RQ we statistically compare the *Co-change rate* pairs belonging to these three groups. More specifically, we want to understand whether the Co-change rate of *smelly* and *mixed* pairs is higher than that of *clean* pairs. We use the non-parametric Mann–Whitney U statistical test [52] to perform the comparison. The Mann–Whitney U test is a non-parametric statistical test used to determine if there is a significant difference between two independent groups. The test allows us to determine whether two independent samples represent two populations with differing median values. It is also possible to determine the direction of the inequality between populations because the test compares the ranks of the values from two independent samples. By ranking all values from both populations together, we can observe whether one population generally has higher or lower ranks compared to the other. Suppose most values from one population have higher ranks than those from the other population. In that case, it suggests that the population tends to have higher values overall, indicating the direction of the inequality. This allows us not only to test whether there is a difference but also to infer which population has higher or lower median values. The computation of the effect size helps us determine the direction of the inequality.

The input for the statistical test consists of the Co-change rate values of the two populations referring to a single project.

Test A: Smelly vs. clean. For each project, we formulate the following hypotheses:

- Null hypothesis H_0^A: Smelly and clean pairs of files are *equally* likely to co-change. Formally $\Theta_{smelly} = \Theta_{clean}$
- Alternative hypothesis H_1^A: Smelly pairs of files are more likely to co-change than clean pairs. Formally $\Theta_{smelly} > \Theta_{clean}$

where Θ_{smelly} and Θ_{clean} represent the median of the Co-change rate in the smelly and clean groups, respectively. It is important to note that the null hypothesis (H_0^A) states that the groups "smelly" and "clean" belong to the same population, while the alternative hypothesis (H_1^A) is its negation. The same applies to the groups compared in the two following tests. The confidence level used for this test and all the following tests is $\alpha = 0.05$.

Test B: Mixed vs. clean. Then, we formulate:

- Null hypothesis H_0^B: Mixed and clean pairs of files are *equally* likely to co-change. Formally $\Theta_{mixed} = \Theta_{clean}$
- Alternative hypothesis H_1^B: Mixed pairs of files are more likely to co-change than clean pairs. Formally $\Theta_{mixed} > \Theta_{clean}$

where Θ_{mixed} and Θ_{clean} represent the median of the Co-change rate in the mixed and clean groups, respectively.

Test C: Smelly vs. mixed. Finally, we formulate:

- Null hypothesis H_0^C: Smelly and mixed pairs of files are *equally* likely to co-change. Formally $\Theta_{smelly} = \Theta_{mixed}$
- Alternative hypothesis H_1^C: Smelly pairs of files are more likely to co-change than mixed pairs. Formally $\Theta_{smelly} > \Theta_{mixed}$

where Θ_{smelly} and Θ_{mixed} represent the median of the Co-change rate in the smelly and mixed groups, respectively.

5.4.2. Effect size

To determine which group has the higher median values and measure the extent of the differences between the compared groups, we calculated the effect size. We used two complementary measures for the effect size: the Common Language Effect Size and the Rank-Biserial Correlation.

As a sample statistic, the Common Language Effect Size (CLES) is computed by forming all possible pairs between the two groups, then finding the proportion of pairs that support a direction (say, that items from group 1 are larger than items from group 2) [53]. This measure is calculated as follows:

$$f = \frac{U_1}{n_1 n_2} \tag{4}$$

where U_1 is the statistic calculated from the test (referring to the first group), and n_1 and n_2 are the number of elements of the two compared populations. The common language effect size represents the probability that a randomly selected value from one population will be higher than a randomly selected value from the other population.

- A CLES > 0.5 means that it is more likely for a randomly selected value from population 1 to be higher than from population 2.
- A CLES < 0.5 indicates that it is more likely for a value from population 2 to be higher.

By comparing this probability to 0.5 (the point of no difference), we can infer the direction of inequality. If the value is higher than 0.5, population 1 tends to have larger values, and if it is lower, population 2 tends to have larger values.

The second method is the Rank-Biserial Correlation [54]. Like other correlational measures, the Rank-Biserial Correlation can range from minus one to plus one, with a value of zero indicating no relationship. The correlation is the difference between the proportion of pairs favorable to the hypothesis f minus its complement (i.e. the proportion that is unfavorable u). This simple difference formula is just the difference of the Common Language Effect Size of each sample, and is as follows [55]:

$$r = f - u \tag{5}$$

The sign of the rank-biserial correlation tells us the direction of the inequality between the medians of the two populations, while its magnitude indicates the strength of that difference.

- Positive Rank-Biserial Correlation: Indicates that values from the first sample (group 1) are generally larger than those from the second sample (group 2).
- Negative Rank-Biserial Correlation: Indicates that values from the first sample are generally smaller than those from the second sample.

5.4.3. Correction

Given that multiple tests are conducted concurrently, there is an increased probability of committing Type I errors. A Type I error, also known as a false positive, occurs when a null hypothesis that is actually true is incorrectly rejected [56]. To mitigate this issue, following the execution of the statistical tests, we applied the Benjamini–Hochberg correction, also known as False Discovery Rate (FDR), to the obtained p-values.

The FDR correction is a statistical method used to adjust P values when multiple statistical tests are performed, in order to control the Type 1 error rate, or the false positive rate. It is applied in situations where several outcomes are being assessed simultaneously. The procedure involves arranging the P values resulting from the statistical tests in descending order. Each P value is then compared to a progressively more stringent significance level [57]. The results presented in the following section refer to the outcomes of the tests after applying the FDR correction.

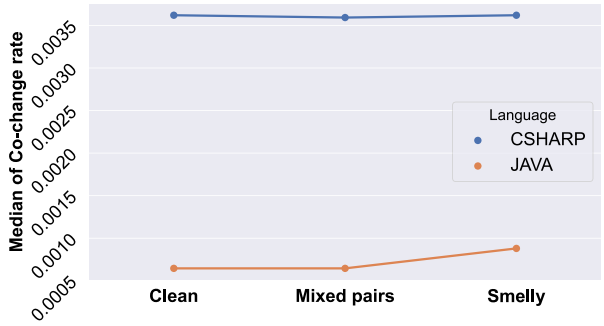


Fig. 5. The median values of Co-change rate by Smell-affected class.

5.5. Results

In this section, we present the results of the analyses and tests explained in the previous section.

Before presenting the tests' results, we outline that it was not possible to perform the tests on 6 projects (5% of the total) because we could not partition the file pairs of these projects into three non-empty populations. For instance, it is meaningless to conduct the test if the Clean group is empty. Consequently, the projects for which we obtained complete results are **112**.

Table 7 compares the percentage of projects in which we rejected the null hypothesis (with p-value < .05) against the percentage of projects in which the null hypothesis was accepted (after the FDR correction). The same table also reports the medians of the p-values and the effect size values. For a detailed view of the values for each individual test, refer to the replication package [48].

Note that for each test, we rejected the null hypothesis in most projects (for test A: 78.57%, test B: 63.39%, test C: 71.43%). The results show that the three groups do not belong to the same population. Considering also the effect size values, this implies that the median value of the Co-change rate in the smelly and mixed pairs populations is *higher* than that in the clean pairs population. Furthermore, the median Co-change rate in smelly is higher than in mixed. Therefore, for most projects, pairs of files affected by AS exhibit a higher chance of co-changing.

The test result is corroborated by the distribution of the Co-change rate value. Fig. 5 shows the median values of the Co-change rate by Smell-affected class. The difference in median values is particularly pronounced in Java projects. Note that the distribution underlying smelly pairs is stochastically greater than the other two. Another point to note is that the values of the Co-change rates are globally low because most system pairs of files do not co-change. Therefore, it is normal for the values to be low, especially when the number of versions is high.

As mentioned before, we repeated the tests for each LOC category to assess the impact of LOC on the analyses. Table 8 shows the total number of pairs per LOC category. Table 7 also contains the results of all the tests performed for each LOC category.

Table 7
Percentage of projects where the null hypothesis is rejected after the FDR correct, along with **median** value of the p-values after the FDR correction, Common Language Effect Size (CLES), and Rank-Biserial Correlation (RBC).

	% H_0 rejection	p-value	CLES	RBC
	Overall			
Test A (smelly vs. clean)	78.57%	0.0028	0.666	0.333
Test B (mixed vs. clean)	63.39%	0.0018	0.436	−0.129
Test C (smelly vs. mixed)	71.43%	0.0010	0.620	0.240
	LOC category XS			
Test A (smelly vs. clean)	47.44%	0.0037	0.711	0.422
Test B (mixed vs. clean)	35.90%	0.0084	0.447	−0.106
Test C (smelly vs. mixed)	41.03%	0.0054	0.663	0.327
	LOC category S			
Test A (smelly vs. clean)	56.12%	0.0030	0.654	0.309
Test B (mixed vs. clean)	44.90%	0.0035	0.461	−0.078
Test C (smelly vs. mixed)	50%	0.0027	0.612	0.223
	LOC category M			
Test A (smelly vs. clean)	65.05%	0.0007	0.671	0.343
Test B (mixed vs. clean)	53.4%	0.0031	0.456	−0.088
Test C (smelly vs. mixed)	66.99%	0.0044	0.610	0.220
	LOC category L			
Test A (smelly vs. clean)	76.47%	0.0013	0.680	0.360
Test B (mixed vs. clean)	63.73%	0.0018	0.415	−0.170
Test C (smelly vs. mixed)	75.49%	0.0023	0.621	0.241

Table 8
Total number of pairs by LOC category.

	Java	C#
XS	864,058	288,846
S	2,614,480	974,320
M	4,047,380	1,831,574
L	7,129,620	4,517,726

Therefore, we repeated the same statistical tests, filtering for file pairs that belong to LOC category XS. It is worth noting that using this filter, we could not perform tests on all the projects analyzed previously (112), but only on 78 projects (69.6%). This is because, in the projects not analyzed, it was not possible to divide the dataset into the 3 non-empty populations smelly, mixed, and clean using only pairs of LOC category XS. This is also motivated by the data in Table 8, where lower LOC categories have fewer pairs. The percentages of null hypothesis rejection are lower than before (test A: 47.44%; test B: 35.90%; test C: 41.03%), but still, in a significant portion of projects, pairs of files affected by AS are more likely to co-change.

Then we proceeded by performing the tests considering only pairs with LOC category S. For the same reasons as before, it was possible to analyze 98 projects (87.5%). The percentages of null hypothesis rejection (test A: 56.12%; test B: 44.90%; test C: 50%) are lower than the overall analyses ignoring LOC but improve compared to the tests with LOC category XS.

Finally, we performed the tests on the remaining LOC categories, M and L. For the same reasons as before, the projects analyzed were 103 (92%) for LOC category M and 102 (90.3%) for LOC category L. It can be observed that there is an increase in the percentages compared to the tests with lower LOC categories (LOC M: test A 65.05%; test B 53.4%; test C 66.99%. LOC L: test A 76.47%; test B 63.73%; test C 75.49%), with LOC category L showing statistics very similar to the results obtained with the initial tests on the entire population of pairs. Additionally, LOC category L has an even higher percentage in test C. The results obtained show that, considering only files with high LOC values (in our case, greater than 56), pairs of files affected by smells co-change more often than pairs of no affected files. It is worth noting that in all the tests performed, the difference between the distributions is more pronounced between smelly and clean compared to mixed and clean. Thus, the increasing number of files affected by smells (from 1 to 2) also leads to an increased rejection of the null hypothesis.

Now let us focus on the results of the effect sizes. As shown in Table 7, the effect size measures indicate that the difference between the smelly vs. clean and smelly vs. mixed samples (Tests A and C) is significant, suggesting that they definitely do not belong to the same population. Furthermore, the data show that the median value of smelly is higher than that of mixed and clean. In contrast, there is less difference between the mixed and clean samples.

Additionally, after applying the FDR correction, we observed a slight decrease in the number of cases where the null hypothesis was rejected (typically a small reduction of 1%–2% in the rejection rate for each test). This outcome is due to the mitigation of Type I error risk associated with conducting multiple statistical tests.

All test results are saved in CSV format in the replication package [48], along with the measures of the Common Language Effect Size and the Rank-Biserial Correlation.

5.6. Discussion

In the following section, we discuss and elaborate on the results presented above.

The test results show that, for a significant percentage of the tested projects, it is indeed true that pairs of files affected by AS tend to co-change more frequently than pairs of clean files. Co-changes are a symptom of tight logical coupling between two files, that facilitates a change in one file to propagate to the other. Therefore, a ***higher maintenance effort*** is required to maintain smelly files than clean files. Consequently, developers working on files affected by AS could pay a *higher* technical debt *interest* than when they work on clean files of similar size. This phenomenon becomes *more common* as the size of the file in the pairs affected by smells increases.

It is interesting to consider that the XS LOC category has a lower rejection rate of H_0. In our opinion, on top of the fact that a small piece of code is less likely to be smelly or change, another factor could be that smaller files are usually abstractions (i.e. interfaces or abstract classes) that contain less code and are intended to change less and, by their nature, have lower coupling. This conjecture is based on Martin's [42] observations, which states that good design is characterized by *stable* abstractions and *unstable* implementation (i.e. concrete classes). Therefore, even if an interface is affected by a smell, if the overall design of the application is solid, developers do not have to change it often and, thus, they do not pay technical debt interest on it.

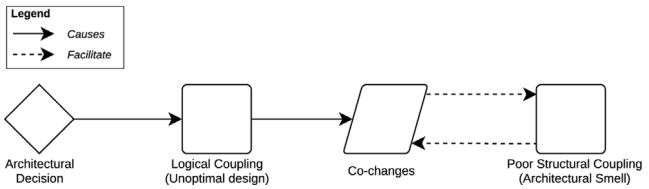


Fig. 6. Architectural smells' introduction in a project induced by logical coupling [12].

These two considerations consolidate into an important **takeaway** for practitioners: *prioritize refactoring large, smelly files that co-change* over clean files, regardless of whether they co-change or not, as breaking logical coupling between two *smelly files* is likely to largely reduce the amount of maintenance required.

In addition to this primary takeaway, to effectively manage architectural technical debt, developers should focus on regularly refactoring highly coupled files or modules. When files exhibit frequent co-changes, this indicates tight coupling, which leads to higher maintenance costs. Refactoring such files to reduce coupling will decrease the need for coordinated changes and reduce long-term technical debt. In line with Martin's observations on good design [42], leveraging abstractions like interfaces or abstract classes can help decouple code, reducing the frequency of changes in related files and thus minimizing technical debt. Additionally, proactively detecting AS through automated tools is crucial. Tools like Arcan can be integrated into CI/CD pipelines to automatically identify architectural issues and smells, enabling developers to address potential problems early in the development cycle, further mitigating technical debt.

Finally, it is worth mentioning how our results compare with related work. Our findings corroborate and strengthen the findings of Sas et al. [12], who analyzed the correlation between smells and co-changes on a smaller set of projects. Their findings show that only in 50% of projects co-changes are significantly correlated with AS. Our findings show that co-changes have a stronger relationship with AS than was initially found by Sas et al. with up to 79% of projects where smelly files have a higher Co-change rate. This is because we focus our analyses on the *Co-change rate* (a continuous variable), rather than on *detecting* co-changes (a categorical variable), thus we can perform a more fine-grained analysis and show that AS are correlated with a *higher* Co-change rate.

In the same paper, the author provides a detailed explanation of the correlation between co-changes and architectural smells. According to the study, poor architectural decisions lead to logical coupling, which in turn results in co-changes, as the concerns are not properly separated among the entities involved in the decision (see Fig. 6). This logical coupling creates the conditions for architectural smells to manifest as actual structural dependencies within the system's dependency network, creating architectural technical debt. Consequently, the affected components become both logically and structurally coupled, increasing the likelihood of changes propagating in a vicious cycle. This insight deepens our understanding of architectural smells, showing that they not only indicate poor architectural decisions but also represent their visible consequences, such as cycles in the dependency network, along with other invisible structural repercussions like logical coupling and co-changes.

It is important to emphasize that with the results obtained, we cannot make any generalizations about the impact of all types of AS. Our conclusions are valid for the four AS considered in the study.

6. Limitations and threats to validity

The work presented in this paper is subject to limitations and threats to its validity that are highlighted in the upcoming subsections.

6.1. Threats to validity

Threats to validity mainly concern construct validity, namely, the degree to which we measure exactly what we aim to measure (i.e. software metrics, architectural smells, smell characteristics). Therefore, the main threat to construct validity arises from our choice to use Arcan as a tool to collect this data. In particular, the detection of the smells is the major threat to validity, as it depends on the implementation offered by Arcan. Lefever et al. [58] have shown that different tools for technical debt measurement (including DV8, CAST, and SonarQube) have divergent, if not conflicting, results regarding which files are problematic in a system. This is because different tools make different assumptions, use different definitions of a smell, and have different implementations of how to detect a smell [58]. Therefore, we can only state that the quantitative data contained in this dataset may not be fully comparable with the results obtained by other tools. However, this would be the case even if we used any other tool, as shown by Lefever et al. [58]. Hence, this threat can be considered partially mitigated, as the definitions of each AS used by Arcan are based on independent, previous works [6,42,59]. Moreover, to guarantee that the results obtained by Arcan are indeed in line with the definitions provided by previous work, the tool was used and evaluated in several studies [14,17,60] on multiple industrial projects written in different programming languages.

Another threat is related to some of the limitations mentioned in the previous section (i.e. broad inclusion criteria), which results in unexpected GitHub projects being included in our dataset. To mitigate this threat as much as possible, we used a combination of statistical and manual analysis to identify and remove undesired projects (e.g. popular educational repositories). Additionally, should potential users require stricter criteria, it is possible to apply more stringent rules to select the projects and further mitigate this threat.

Concerning the threats of our empirical study, we share the same threats to construct validity related to Arcan's implementation as described above as well as the ones concerning the selection of the projects. To mitigate the latter threat, we performed a manual inspection of the projects selected to be analyzed for the empirical study to ensure they were indeed suitable for our case.

Finally, another concern pertains to the fact that while the statistical tests do indicate a difference between the two samples, in practice, this difference may be negligible. To address this, we calculated the effect size, which confirms a significant difference between the two samples.

6.2. Limitations of the pipeline

The dataset created through the pipeline is based on a set of intentionally broad selection criteria outlined in Section 4.1.1. This broad approach, while inclusive, might not fully meet the requirements of all researchers. A limitation arises from the availability of historical data for the projects in our dataset. We opted to skew the data towards the most recent versions, with the rationale that, since this is a continuous data pipeline, it has to always include the latest versions and, eventually, the distribution will normalize. Additionally, a secondary purpose of the dataset is to serve as a benchmark for Arcan's metrics, particularly the technical debt index. Furthermore, we favored expanding the dataset horizontally (i.e., adding more projects) while accepting a slight trade-off in data quality, rather than expanding it vertically (i.e., adding more versions of the same projects).

6.3. Addressing privacy concerns

In curating our dataset of open-source GitHub projects, we place paramount importance on addressing privacy concerns and ensuring fair use. It is worth noting that we refrain from collecting and storing sensitive user data, such as names or email addresses. Our data

collection is limited to saving the repository owner's username. Additionally, we exclusively analyze public repositories, which are promptly removed from our records once the analysis is complete. Furthermore, we respect the rights of GitHub users who wish to have their projects excluded from our dataset. To facilitate this process, users can make removal requests by emailing the authors of the study.

7. Conclusions and future work

In this work, we introduced a data collection pipeline that can continuously analyze open-source projects to detect architectural smells using Arcan 2.9.0. For the first compiled dataset, the pipeline analyzed over 30,000 commits and releases mined from over 13,000 GitHub repositories. The analyses are compiled into a dataset containing all the successful analyses as an SQLite database. The first dataset is publicly available[3] as well as the source code of the Airflow pipeline[2] and the tool Arcan 2.9.0[5].

Our long-term goal is to create a platform that streamlines data collection for software engineering empirical studies, and we believe, that this work is a step towards achieving this goal. This platform will help software engineering researchers increase the external validity of their studies while also reducing the effort required to build a new dataset from scratch.

Through the empirical study described in the paper we observed that the presence of AS is coincident with a higher Co-change rate of the affected files, thus increasing maintenance effort for developers. These results not only demonstrate the applicability of the data collected by the pipeline, but also provide useful insights for practitioners willing to manage architectural technical debt.

Future work on the data pipeline can focus on (1) further refine the list of projects analyzed by the pipeline and exclude undesired projects; (2) develop API access to the data so that users can retrieve only the portion of the dataset they are most interest in; (3) improve the pipeline's performance by reducing the failure rate of Arcan.

As for the empirical study, a potential direction for future research is to investigate the correlation between co-changes and AS in additional programming languages such as Python, C, and C++. However, incorporating other languages is not as straightforward as it may seem. The definition of AS may need to be re-evaluated in certain cases, given that languages like C and C++ are not inherently object-oriented. This fundamental difference complicates direct comparisons with languages such as Java or C#, where AS definitions often rely on object-oriented principles. Therefore, any extension of this research to additional languages may be accompanied by a rigorous redefinition of AS tailored to each language's characteristics.

An intriguing future research path involves investigating the potential correlation between GitHub issues and the presence of AS in the tracked projects. For instance, researchers can explore whether software projects burdened with less technical debt exhibit faster issue resolution, or if there exists a quantitative relationship between the volume of AS and the frequency of reported issues.

Another interesting research direction is to study AS refactorings and better understand how they are removed from a system. In particular, it would be interesting to see whether a large language model can learn from historical data (collected by the pipeline) of architectural smell removal and test the assumption that incidental removals can be used to teach artificial intelligence how to refactor architectural smells. Such an approach has obvious implications for practitioners, as it would help them more easily repaying the technical debt accrued by the smells.

CRediT authorship contribution statement

Matteo Bochicchio: Writing – review & editing, Writing – original draft, Software, Formal analysis, Data curation, Conceptualization. **Darius Sas:** Writing – review & editing, Writing – original draft, Supervision, Conceptualization. **Alessandro Gilardi:** Data curation. **Francesca Arcelli Fontana:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Conceptualization.

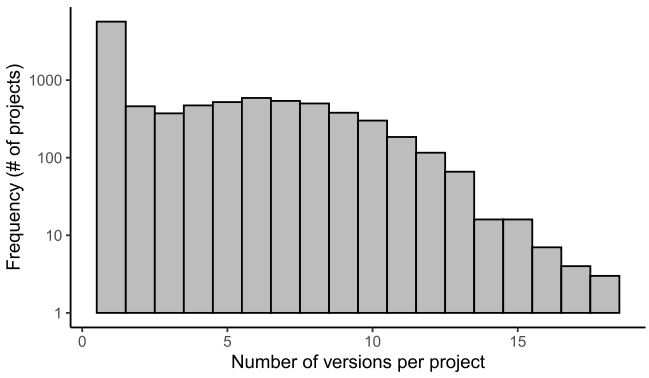


Fig. 7. The frequency of the number of versions per project (*y*-axis uses logarithmic scale).

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to assist with writing. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

Declaration of competing interest

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

Appendix. Appendix on the dataset for benchmark studies

This Appendix describes the dataset created by the execution of the data pipeline by providing a high-level overview of the projects analyzed and a description of how the data is structured. This dataset can be used as a new reference benchmark by Arcan to calculate the technical debt score and rating.

A.1. Demographiscs

The distribution of the number of versions analyzed per project is depicted in Fig. 7. About 55% of the projects have only one version analyzed, while 40% have between 2 and 10 versions, while the remaining 5% have up to 18 versions. This distribution is expected as only a few projects have a development driven by a company or a big community that can sustain a regular release schedule. Indeed, this is the case of the only three projects of which we analyzed 18 versions: M66B/FairEmail (Java, 2000+ GitHub releases), electron/electron (C++, 1200+ GitHub releases), and xamarin/xamarin-forms-samples (C#, 1900+ GitHub releases). The average number of days in between versions analyzed is 68, 150, and 18, respectively.

According to the line of code of the projects, further details are provided in Table 9, which breaks down the totals per programming language, and Fig. 8, which depicts the distribution of the total lines of code in each version analyzed. As it can be noted, C and C++ have the most analyzed number of lines of code and, on average, the versions for these two languages have a higher count of lines of code analyzed.

As we outlined, Arcan 2.9.0 can estimate the architectural technical debt of a project through the architectural smells detected by the tool and indicated in Table 1. Fig. 9 depicts the density Architectural Technical Debt Index (ATDI) per thousand of lines of code. As it can be noted, the median density is similar across all programming languages included in the dataset, with the only exception being Python, which has a lower density than the others. Another similarity, is that there are several outliers for all languages, which correspond to the big projects in the dataset.

Table 9
The total number of lines of code analyzed divided by programming language.

Language	C	C++	C#	Java	Python	Total
Lines of code	344,615,925	370,776,940	64,434,382	293,096,608	151,755,117	1,224,678,972

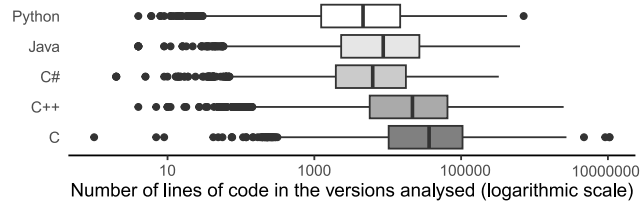


Fig. 8. The distribution of the number of lines of code in the versions analyzed grouped by programming language.

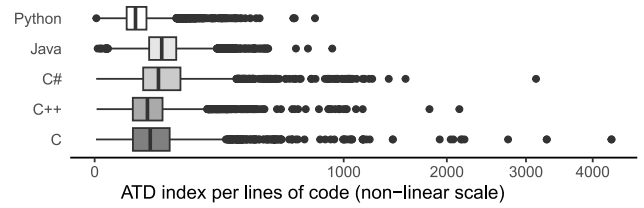


Fig. 9. The distribution the ATDI per KLOC in the versions analyzed grouped by programming language.

A.2. Accessing and using the dataset

The dataset is an SQLite database that represents the nodes and edges of the GraphML files produced by Arcan 2.9.0 as two tables. The nodes table's records are either a component or a smell in a specific version, with each field representing the software metrics and smell characteristics reported in Table 1, Figs. 1 and 2. Similarly, each record in the edges table represents a single edge in the graph with its properties as fields, including which nodes they connect and their weight. To explore and inspect the dataset we used the free tool SQLiteStudio.[11] Additionally, we included the R scripts we used to analyze the data in the replication package. Both tables provide the means to uniquely identify the projects (via the `project_name` column) and versions analyzed (via the pair of columns `project_name` and `project_version`). The replication package contains a detailed description of each column of the two tables [48].

A.3. Arcan2

The database has been created by using for AS detection the tool Arcan 2, specifically version 2.9.0. The main features of Arcan with respect to the first version of the tool [37], are related to the supported programming languages (the first version supported only Java), improvements in the detection rules of the smells and in the Architectural Debt Index computation, the integration of the pipeline for the data analysis, the possibility of doing evolution analyses, and the calculation of several other software metrics.

Data availability

The dataset is publicly available on Zenodo: https://zenodo.org/records/13827946.

[11] See https://sqlitestudio.pl/.

References

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), Dagstuhl Rep. 6 (4) (2016) 110–138, http://dx.doi.org/10.4230/DagRep.6.4.110, http://drops.dagstuhl.de/opus/volltexte/2016/6693.

[2] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220, http://dx.doi.org/10.1016/j.jss.2014.12.027, http://www.sciencedirect.com/science/article/pii/S0164121214002854.

[3] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, I. Gorton, Measure it? Manage it? Ignore it? software practitioners and technical debt, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015, ACM Press, New York, New York, USA, 2015, pp. 50–60, http://dx.doi.org/10.1145/2786805.2786848, URL http://dl.acm.org/citation.cfm?id=2786805.2786848.

[4] U. Azadi, F.A. Fontana, D. Taibi, Architectural smells detected by tools : a catalogue proposal, Int. Conf. Tech. Debt (TechDebt 2019) (March) (2019) http://dx.doi.org/10.1109/TechDebt.2019.00027.

[5] R. Verdecchia, I. Malavolta, P. Lago, Architectural technical debt identification: the research landscape, in: 2018 ACM/IEEE International Conference on Technical Debt, 2018, http://dx.doi.org/10.1145/3194164.3194176.

[6] M. Lippert, S. Roock, Refactorings in large software projects - how to successfully execute complex restructurings, Wiley, 2006, p. 286.

[7] J. Garcia, P. Daniel, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2009, pp. 255–258, http://dx.doi.org/10.1109/CSMR.2009.59.

[8] S. Kim, T. Zimmermann, E.J. Whitehead, A. Zeller, Predicting faults from cached history, 2007, pp. 489–498, http://dx.doi.org/10.1109/ICSE.2007.66,

[9] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, Y. Cai, A study on the role of software architecture in the evolution and quality of software, Vol. 2015-Augus, IEEE Computer Society IEEE International Working Conference on Mining Software Repositories, 2015, pp. 246–257, http://dx.doi.org/10.1109/MSR.2015.30.

[10] F. Arcelli Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, Proc. - 2016 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2016 (2016) 433–437, http://dx.doi.org/10.1109/ICSME.2016.33.

[11] D. Sas, P. Avgeriou, An architectural technical debt index based on machine learning and architectural smells, IEEE Trans. Softw. Eng. 49 (8) (2023) 4169–4195, http://dx.doi.org/10.1109/TSE.2023.3286179.

[12] D. Sas, P. Avgeriou, R. Kruizinga, R. Scheedler, Exploring the relation between co-changes and architectural smells, SN Comput. Sci. 2 (1) (2021) 13, http://dx.doi.org/10.1007/S42979-020-00407-5.

[13] T. Sharma, P. Mishra, R. Tiwari, Designite - a software design quality assessment tool, in: 2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, BRIDGE, 2016, pp. 1–4.

[14] D. Sas, P. Avgeriou, F. Arcelli Fontana, Investigating instability architectural smells evolution: an exploratory case study, in: 35th International Conference on Software Maintenance and Evolution, IEEE, 2019, pp. 557–567, http://dx.doi.org/10.1109/ICSME.2019.00090, URL https://ieeexplore.ieee.org/document/8919109/.

[15] P. Gnoyke, S. Schulze, J. Krüger, An evolutionary analysis of software-architecture smells, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 413–424.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, in: 2010 Asia Pacific Software Engineering Conference, IEEE, 2010, pp. 336–345.

[17] D. Sas, P. Avgeriou, U. Uyumaz, On the evolution and impact of architectural smells - an industrial case study, Empir. Softw. Eng. 27 (4) (2022) 86, http://dx.doi.org/10.1007/S10664-022-10132-7.

[18] I. Pigazzini, F.A. Fontana, B. Walter, A study on correlations between architectural smells and design patterns, J. Syst. Softw. 178 (2021) 110984, http://dx.doi.org/10.1016/J.JSS.2021.110984.

[19] F. Arcelli Fontana, V. Lenarduzzi, R. Roveda, D. Taibi, Are architectural smells independent from code smells? An empirical study, J. Syst. Softw. 154 (2019) 139–156, http://dx.doi.org/10.1016/j.jss.2019.04.066, arXiv:1904.11755.

[20] T. Sharma, P. Singh, D. Spinellis, An empirical investigation on the relationship between design and architecture smells, Empir. Softw. Eng. 25 (5) (2020) 4020–4068, http://dx.doi.org/10.1007/s10664-020-09847-2.

[21] S. Herold, An initial study on the association between architectural smells and degradation, in: Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14, Springer, 2020, pp. 193–201.

[22] D. Sas, P. Avgeriou, I. Pigazzini, F. Arcelli Fontana, On the relation between architectural smells and source code changes, J. Softw.: Evol. Process. 34 (1) (2021) http://dx.doi.org/10.1002/smr.2398.

[23] F. Jaafar, Y.G. Guéhéneuc, S. Hamel, F. Khomh, M. Zulkernine, Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults, Empir. Softw. Eng. 21 (3) (2016) 896–931, http://dx.doi.org/10.1007/s10664-015-9361-0.

[24] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Trans. Softw. Eng. 41 (5) (2014) 462–489.

[25] C. Couto, C. Maffort, R. Garcia, M.T. Valente, COMETS: A dataset for empirical research on software evolution using source code metrics and time series analysis, SIGSOFT Softw. Eng. Notes 38 (1) (2013) 1–3, http://dx.doi.org/10.1145/2413038.2413047.

[26] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented designa metrics suite for object oriented design, PhD Propos. 1 (1992) 476–493.

[27] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. De Lucia, Landfill: An open dataset of code smells with public evaluation, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 482–485, http://dx.doi.org/10.1109/MSR.2015.69.

[28] V. Lenarduzzi, N. Saarimäki, D. Taibi, The technical debt dataset, in: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, 2019, pp. 2–11.

[29] Y.-G. Gueheneuc, Ptidej: A flexible reverse engineering tool suite, in: 2007 IEEE International Conference on Software Maintenance, 2007, pp. 529–530, http://dx.doi.org/10.1109/ICSM.2007.4362684.

[30] T. Diamantopoulos, M.D. Papamichail, T. Karanikiotis, K.C. Chatzidimitriou, A.L. Symeonidis, Employing contribution and quality metrics for quantifying the software development process, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 558–562, http://dx.doi.org/10.1145/3379597.3387490.

[31] V. Thakur, M. Kessentini, T. Sharma, Qscored: An open platform for code quality ranking and visualization, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 818–821.

[32] T. Sharma, M. Kessentini, Qscored: A large dataset of code smells and quality metrics, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, IEEE, 2021, pp. 590–594.

[33] G. Gousios, D. Spinellis, Ghtorrent: GitHub's data from a firehose, in: 2012 9th IEEE Working Conference on Mining Software Repositories, MSR, IEEE, 2012, pp. 12–21.

[34] D. Spadini, M. Aniche, A. Bacchelli, Pydriller: Python framework for mining software repositories, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 908–911.

[35] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: Mining and recommending API usage patterns, in: ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23, Springer, 2009, pp. 318–343.

[36] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for software design smells: Managing technical debt, Refactoring for Software Design Smells: Managing Technical Debt, 2014, pp. 1–237, http://dx.doi.org/10.1016/C2013-0-23413-9, arXiv:arXiv:1011.1669v3 http://www.designsmells.com/resources/.

[37] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, E.D. Nitto, Arcan: A tool for architectural smells detection, Proc. - 2017 IEEE Int. Conf. Softw. Archit. Work. ICSAW 2017: Side Track Proc. (2017) 282–285, http://dx.doi.org/10.1109/ICSAW.2017.16.

[38] R.C. Martin, Object oriented design quality metrics: An analysis of dependencies, 1995.

[39] D.M. Le, C. Carrillo, R. Capilla, N. Medvidovic, Relating architectural decay and sustainability of software systems, in: Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Institute of Electrical and Electronics Engineers Inc., 2016, pp. 178–181, http://dx.doi.org/10.1109/WICSA.2016.15, URL http://ieeexplore.ieee.org/document/7516827/.

[40] R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot patterns: The formal definition and automatic detection of architecture smells, Proc. - 12th Work. IEEE/ IFIP Conf. Softw. Archit. WICSA 2015 (2015) 51–60, http://dx.doi.org/10.1109/WICSA.2015.12.

[41] R. Roveda, F.A. Fontana, I. Pigazzini, M. Zanoni, Towards an architectural debt index, in: Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018, IEEE, 2018, pp. 408–416, http://dx.doi.org/10.1109/SEAA.2018.00073, URL https://ieeexplore.ieee.org/document/8498240/.

[42] R.C. Martin, J. Grenning, S. Brown, K. Henney, J. Gorman, Clean Architecture: a Craftsman's Guide to Software Structure and Design, Prentice Hall, 2018.

[43] M.O. Elish, M.A.R. Al-Khiaty, A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software, J. Softw.: Evol. Process. 25 (5) (2013) 407–437, http://dx.doi.org/10.1002/smr.1549, URL http://doi.wiley.com/10.1002/smr.1549.

[44] H.A. Al-Mutawa, J. Dietrich, S. Marsland, C. McCartin, On the shape of circular dependencies in java programs, in: Proceedings of the Australian Software Engineering Conference, ASWEC, IEEE, 2014, pp. 48–57, http://dx.doi.org/10.1109/ASWEC.2014.15, URL http://ieeexplore.ieee.org/document/6824106/.

[45] L. Pruijt, C. Köppe, J.M. van der Werf, S. Brinkkemper, The accuracy of dependency analysis in static architecture compliance checking, in: Software - Practice and Experience, 47, (2) John Wiley and Sons Ltd, 2017, pp. 273–309, http://dx.doi.org/10.1002/spe.2421, URL http://doi.wiley.com/10.1002/spe.2421.

[46] C. Sas, A. Capiluppi, C.D. Sipio, J.D. Rocco, D. Di Ruscio, Gitranking: A ranking of github topics for software classification using active sampling, Softw.: Pr. Exp. (2022).

[47] M. Izadi, A. Heydarnoori, G. Gousios, Topic recommendation for software repositories using multi-label classification algorithms, Empir. Softw. Eng. 26 (2021) 1–33.

[48] M. Bochicchio, D. Sas, A. Gilardi, F. Arcelli Fontana, Dataset and replication package, 2025, http://dx.doi.org/10.5281/zenodo.15051564.

[49] U. Brandes, M. Eiglsperger, J. Lerner, C. Pich, Graph markup language (graphml), in: Handbook of Graph Drawing and Visualization, CRC Press, 2013, pp. 517–541.

[50] R. van Solingen (Revision), V. Basili (Original article, G. Caldiera (Original article, H.D. Rombach (Original article, Goal question metric (GQM) approach, in: Encyclopedia of Software Engineering, John Wiley & Sons, Ltd, 2002, http://dx.doi.org/10.1002/0471028959.sof142, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471028959.sof142 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof142.

[51] P. Runeson, M. Höst, A. Rainer, B. Regnell, Case Study Research in Software Engineering - Guidelines and examples, 1st ed., John Wiley & Sons, Inc., 2012, p. 241, http://dx.doi.org/10.1002/9781118181034.

[52] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1) (1947) 50–60, http://dx.doi.org/10.1214/aoms/1177730491.

[53] J.A. Kennedy, L.J. Kray, G. Ku, A social-cognitive approach to understanding gender differences in negotiator ethics: the role of moral identity, Organ. Behav. Hum. Decis. Process. 138 (2017) 28–44, http://dx.doi.org/10.1016/j.obhdp.2016.11.003.

[54] E.E. Cureton, Rank-biserial correlation, Psychometrika 21 (3) (1956) 287–290.

[55] D.S. Kerby, The simple difference formula: An approach to teaching nonparametric correlation, Compr. Psychol. 3 (2014) 11.IT.3.1, http://dx.doi.org/10.2466/11.IT.3.1, arXiv:https://doi.org/10.2466/11.IT.3.1.

[56] S. Dudoit, M.J. Van der Laan, K.S. Pollard, Multiple testing. Part I. Single-step procedures for control of general type I error rates, Stat. Appl. Genet. Mol. Biol. 3 (1) (2004).

[57] C. Andrade, Multiple testing and protection against a type 1 (false positive) error using the Bonferroni and hochberg corrections, Indian J. Psychol. Med. 41 (1) (2019) 99–100.

[58] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, On the lack of consensus among technical debt detection tools, in: Proceedings - International Conference on Software Engineering, IEEE Computer Society, 2021, pp. 121–130, http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00021, arXiv:2103.04506.

[59] G. Samarthyam, G. Suryanarayana, T. Sharma, Refactoring for software architecture smells, in: Proceedings of the 1st International Workshop on Software Refactoring - IWoR 2016, ACM Press, New York, New York, USA, 2016, pp. 1–4, http://dx.doi.org/10.1145/2975945.2975946, URL http://dl.acm.org/citation.cfm?doid=2975945.2975946.

[60] D. Sas, I. Pigazzini, P. Avgeriou, F.A. Fontana, The perception of architectural smells in industrial practice, IEEE Softw. 38 (6) (2021) 35–41, http://dx.doi.org/10.1109/MS.2021.3103664.