Architecture in the Cradle: Early Warning of Architectural Decay with ArchGuard

Haoyu Liu, Dominik Fuchß, Sophie Corallo, Maximilian Hummel, Jan Keim, Tobias Hey
Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany
{haoyu.liu, dominik.fuchss, sophie.corallo, maximilian.hummel, keim, hey}@kit.edu

***Abstract*—Architectural decay can manifest as the evolution of architectural smells, degrading integrity, and increasing maintenance costs. Existing techniques capture smells post hoc or predict on component level, acting too late or on too coarse a granularity. We investigate if the risk of introducing architectural smells can already be predicted when issues are opened. Thus, we propose an issue-level prediction approach that utilizes the semantic representations of Large Language Models (LLMs).**

To enable training and evaluation, we construct a dataset from three GitLab-hosted projects by linking issues to smells via smell-inducing changes. On this dataset, we train classifiers to identify high-risk issues and conduct an empirical study comparing seven different representations and nine classifiers.

Our best-performing classifier (SVM with OpenAI embeddings) achieves F_1-scores of up to 0.506, with a recall of about 0.74. This means that our approach can identify approximately 74% of smell-inducing issues before implementation begins. When design alternatives are still being considered. Our approach provides early warnings of potential architectural risks. This work shifts from reactive remediation to proactive quality assurance, raising awareness of potential architectural risks.

***Index Terms*—Software Architecture, Software Quality, Machine Learning, Natural Language Processing, Classification**

I. Introduction

Software architecture shapes how systems evolve [1]–[6]. Due to insufficient architectural awareness, evolution can introduce architectural violations, which degrade architectural integrity over time [2], [7], [8]. Regarding this phenomenon, Li et al. [9]'s survey suggests that developers have referred to it by different names, including architectural erosion, decay, and degradation. Following them, we use decay to refer to this phenomenon. Following previous empirical studies of decay [7], [10], [11], we use architectural smells to measure decay. Once introduced, smells tend to persist and grow rather than being resolved [11], [12], making early detection and prevention critical for long-term quality. Therefore, warning about architectural decay is an important task.

Existing work for managing architectural decay has mainly pursued two directions. *Post-hoc detection* uses static analysis to identify architectural smells in a system [13], [14], but by then smells are embedded and costly to remediate. *Component-level forecasting* predicts which components may deteriorate [10], but operates at too coarse a granularity to guide developers during planning and design. Both approaches intervene too late or at too high a level to effectively prevent smells.

A proactive approach can warn about changes that cause architectural smells. However, this requires early, fine-grained signals at decision points before implementation. In agile development, issues coordinate work from planning through completion and contain rich information [15]–[18]. However, prior work focuses on issues *impacted by* existing architectural smells: the **results** of smells. We focus on issues that *induce* architectural smells: the **reasons** for smells. Specifically, we define architectural smell-inducing (ASI) issues whose closure introduces new smells or expands existing ones.

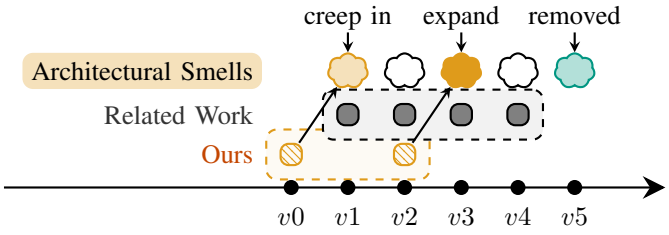

Fig. 1. Simplified life cycle of architectural smells

Understanding ASI issues is a basis for preventing architectural decay at its source. Figure 1 illustrates the life cycle of smells and the distinct opportunities for prevention. A smell enters the system at v_1, expands at v_3, and is eventually removed at v_5. Prior research on smell-impacted issues (●) addresses the period from v_1 to v_4, when smells **already** exist and affect development work. In contrast, ASI issues (▨) occur at v_0 and v_2, **before** the smell manifests or expands. These are the critical intervention points: if developers were warned of the potential architectural damage an issue might introduce during the planning phase (at v_0 or v_2), they could make more informed design decisions and manually prevent the introduction or exacerbation of smells. This shift from reactive remediation to proactive warning represents a fundamental change in how we can manage architectural quality.

To our knowledge, despite their potential value, architectural smell-inducing issues have not been studied in prior work. This gap motivates our research, driven by two main questions:

(i) **Identification**: Can we identify which issues in a project's history induced architectural smells?
(ii) **Prediction**: Can we predict whether an issue will induce architectural smells based solely on information available before implementation?

To address the identification challenge, we develop a traceability recovery approach that links architectural smells to

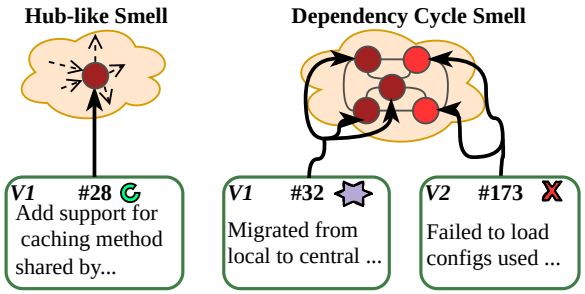

Fig. 2. Evolved architectural smells with annotated inducing issues

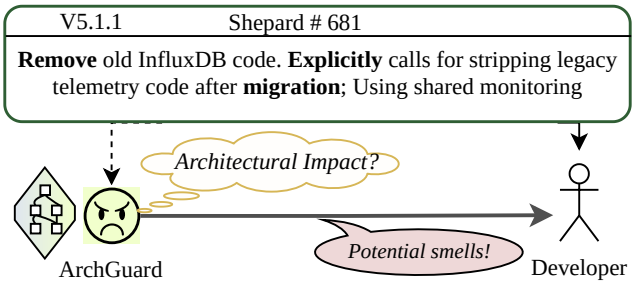

Fig. 3. ArchGuard warns developers to consider avoiding architectural smells

their inducing issues. Our method combines smell evolution analysis with issue-commit links: we track how file-level changes cause smells to evolve across versions, then trace these changes back through the commit history to the issues. This chain enables us to systematically annotate issues that historically induced architectural smells in a project.

For the prediction challenge, we introduce **ArchGuard**, a machine learning-based approach that predicts architectural smell-inducing issues using only their title and description, information already available at issue creation (**before code changes**). ArchGuard leverages Large Language Models (LLMs) to extract semantic representations from text, capturing the architectural implications embedded in natural language descriptions. We explore multiple classifier families to determine which models are most effective at distinguishing architectural smell-inducing issues from regular ones.

We evaluate our approach on three open-source projects. Our identification method successfully traces 41.2% to 100% of smells to smell-inducing issues, demonstrating that architectural decay is meaningfully related to specific development activities. For prediction, we compare nine classifiers across three families: traditional machine learning, fine-tuned transformers, and LLMs, achieving a recall of up to 0.782 with F_1-scores up to 0.506. We also conduct an ablation study to isolate the contributions of different features and compare semantic representations, thereby identifying which signals are most predictive. For cold-start scenarios, such as a new project with limited training data, we conduct a cross-project evaluation to assess the generalizability of our approach.

As such, this paper makes the following three contributions:

1) A traceability approach linking architectural smells to the issues that introduced them (ASI issues).
2) ArchGuard, an architectural-decay warning framework.
3) An empirical evaluation comparing three families of established machine-learning classifiers.

II. Background

We first present concepts of architectural smells and usage scenarios of our approach. We then present the limitations of the issue-based architectural design decisions (ADDs) recovery approach in identifying smell-inducing issues by showing their overlap with ASI issues.

A. Architectural smells

Instances of smells are sets of components together with a smell type (e.g., cyclic dependencies, hub-like component, microservice bad smells) [13], [19], [20]. They accumulate as architectural technical debt [21], impacting maintainability [19], [22]–[26]. In this study, we focus on two widely studied smells: *cyclic dependencies* and *hub-like components*. Cyclic dependencies violate the acyclicity of the dependency graph by forming cycles among components. Such cycles are detected by analyzing component dependency graphs. The analysis tool Arcade [14] identifies them as sets of components forming strongly connected subgraphs, detected using Kosaraju's algorithm [27]. In contrast, Arcan [13] flags any cycle induced by dependencies and identifies them via depth-first search.

Hub-like components capture an excessive concentration of dependencies around a component, indicating overloaded responsibilities. Both Arcade and Arcan operationalize this smell using a threshold on component dependency counts, but differ in the statistics used to set the threshold. Arcade sets it using mean and standard deviations across components, while Arcan uses median and standard deviation.

B. Scen. 1: Analyzing smell-inducing issues retrospectively

Architectural smells evolve over time as projects evolve. Linking smells to the issues that induce them helps uncover their root causes and is crucial for remediation.

Understanding individual smell origins. Smells result from multiple decisions across versions. For example, a hub-like smell accumulates through numerous architectural decisions that add dependencies. Identifying the issues that ultimately make the component into a hub helps developers understand the rationale and make informed refactoring choices.

Tracing multi-component smell evolution. Complex smells, such as dependency cycles, involve multiple architectural components. As illustrated in Figure 2, components were infected by the smell at the same time, rather than gradually by different decisions. An annotated list of architectural smell-inducing issues for each component provides the context necessary to effectively break the cycle.

Identifying smell-inducing patterns. Annotated ASI issue lists enable empirical studies of what issue types drive smell growth, revealing whether bug-fixing or feature development is more likely to intensify smells.

TABLE I
DIFFERENCE BETWEEN ASI ISSUES AND ADD ISSUES

Project	Total Issues	Issues linked to Commit	ASI issues (% linked)	RecovAr (% linked)	ASI not in RecovAr (% ASI)	RecovAr not in ASI (% RecovAr)
Inkscape	5788	2536	257 (10.13%)	846 (33.36%)	81 (31.51%)	670 (79.20%)
Shepard	674	346	46 (13.29%)	159 (45.95%)	0	(0%) 113 (71.07%)
StackGres	2613	1581	279 (17.65%)	808 (51.11%)	0	(0%) 529 (65.47%)

C. Scen. 2: Predicting smell-inducing issues Just-in-Time

Beyond retrospective analysis, annotated architectural smell-inducing issues enable proactive quality assurance through Just-in-Time (JiT) prediction. With annotated issues, a prediction model can also be trained. As illustrated in Figure 3, this approach can be integrated into the development workflow[1]. When developers create new issues, such a model can predict whether the issue is relevant to the growth of architectural smells (ASs). Receiving warnings from ArchGuard, the developer could be more aware of the potential consequences and make better design decisions.

D. Prestudy: Understanding Smell-Related Issues

While existing approaches such as RecovAr [28] and ADDRA [29] can recover architectural design decisions from architectural change-related issues (ADD issues), they are not well-suited to studying smell evolution specifically.

We identify two key limitations: First, ADD issues might not cover all ASI issues. **Commonality:** ADD issues are captured by detecting architectural changes at the component-entity level, such as files moving between components. Architectural smells primarily concern inter-component dependencies [30]. For both, the core reason is a change in dependencies. **Difference:** Although they co-occur often, it is not guaranteed that dependency changes causing smells can always cause file movements. For example, adding a dependency from file f_1 in component C_1 to f_2 in component C_2 would create a component-level dependency from C_1 to C_2, potentially forming a dependency cycle. However, a single dependency may be sufficiently strong to drive file movements.

Second, ADDs encompasses broad architectural concerns beyond smells, requiring maintainers to perform additional filtering to identify smell-inducing issues.

To validate these limitations, we conducted an empirical study comparing architectural change-related issues (captured by RecovAr) with architectural smells detected by Arcade [14] on three open-source projects (detailed project descriptions in Section V-C). As shown in Table I, for Inkscape, RecovAr only captures 68.49% of ASI issues, missing 31.51% of smell-inducing issues. Furthermore, 79.20% of architectural change-related issues recovered by RecovAr are not smell-inducing, confirming the need for a more targeted approach. Similar patterns are observed across Shepard and StackGres. Although RecovAr does not miss smell-inducing issues, it contains 71.07% and 65.47% non-smell-related issues.

[1]https://gitlab.com/dlr-shepard/shepard/-/issues/681

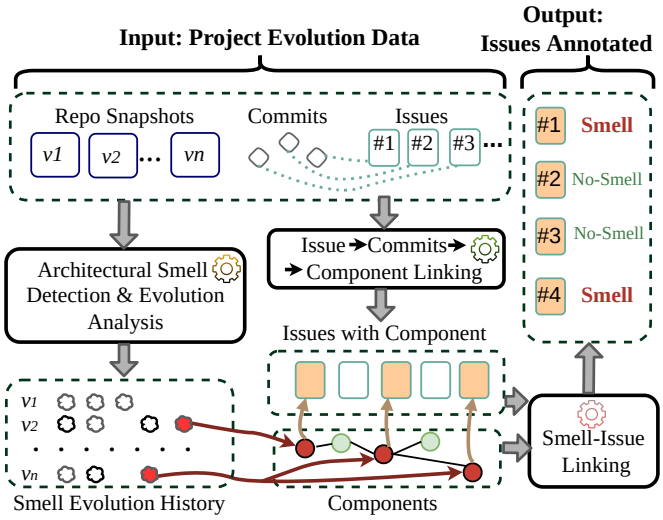

Fig. 4. Architectural smell-inducing issue identification process

These findings motivate us to propose an approach to specifically identify architectural smell-inducing issues rather than relying on general architectural change detection.

III. FINDING ARCHITECTURAL SMELL-INDUCING ISSUES

This part aims to link architectural smells to the issues that induce them. We define ASI issues as those for which closing them entails introducing new architectural smells or adding new architectural components to existing smells. However, due to the nature of issues, they exist in a different domain than architectural smells. To connect them, we identify and use changes that induce architectural smells. Specifically, we use four architecture-based static analysis steps: **smell detection**, **smell evolution analysis**, **issue-component linking**, and **smell-to-issue linking**. Figure 4 provides an overview of the process. Given project repositories across different versions, smell detection will identify architectural smells and annotate each smell's affected components (smelly components). After that, the results of smell evolution are changes in smells across versions. After linking to components, issues affecting the smelly components will be annotated as ASI issues.

A. Smell Detection

This step aims to obtain the evolved AS and the files affected by them throughout the project's evolution. Based on the files, we can link the issues whose resolution was involved.

a) Version graph and evolution paths: Release histories of complex systems often involve branching and merging. We therefore model the evolution of a project P as a directed version graph $G_{Ver} = (V, E)$. V is the set of versions (releases or commit tags). Each version $v \in V$ is associated with a repository snapshot $Repo(v)$. $E \subseteq V \times V$ is the set of directed edges, where (v_i, v_{i+1}) in E denotes that v_{i+1} directly succeeds v_i (consecutive releases or parent-child in the commit graph). We can analyze decay step by step along an *evolution path*: $EvoPath = (v_1, v_2, \ldots, v_\ell)$. Such a path

can represent the main branch or a release line (e.g., branch 1.x vs. 2.x); if a system contains multiple branches, different paths are analyzed separately.

b) Smell detection interface: This module is designed to be tool-agnostic and flexible: Any architectural smell detection tool can be plugged in, as long as it satisfies the input/output specification defined in Equation 1.

For each version v along an evolution path, we apply an architectural smell detection tool to the snapshot $Repo(v)$. The tool returns a set of smell instances:

$$S^v = \{s_1^v, s_2^v, \ldots, s_{k_v}^v\}. \tag{1}$$

Each smell instance $s \in S^v$ is characterized by: (i) a smell type $type(s)$ (e.g., dependency cycle, link overload) and (ii) a non-empty set of components $Comp(s)$ in the architecture of version v that participate in the smell. This specification aligns with popular architectural smell-detection tools such as Arcade [14] and Arcan [22].

B. Smell Evolution Analysis

After applying architectural smell detection to all versions in an evolution path, each version v is annotated with a smell set S^v, and, for each smell $s \in S^v$, the corresponding component set $Comp(s)$. The next step is to analyze how smells evolve between consecutive versions.

a) Matching smells across versions: The smell-matching process follows the workflow of the smell evolution tool AStracker [11]. Consider two consecutive versions v_1 and v_2 with smell sets S^{v_1} and S^{v_2}. We first group smells by type and compare only smells of the same type (e.g., comparing all dependency-cycle smells with each other, all Hub-like smells with each other). Within one type, we align individual smell instances based on the overlap of their component sets.

Given two smells $s^1 \in S^{v_1}$ and $s^2 \in S^{v_2}$ of the same type, we measure their similarity using the Jaccard index over their component sets $Comp(\cdot)$:

$$J(s^1, s^2) = \frac{|Comp(s^1) \cap Comp(s^2)|}{|Comp(s^1) \cup Comp(s^2)|} \tag{2}$$

Two smells are considered a match if (i) they have the same type and (ii) their Jaccard similarity is at least above a threshold θ (hyperparameter).

b) Evolution categories: Using the matching result from an edge (v_1, v_2), we classify smell changes involved in this step into four categories:

- **Remaining**: matched pairs (s^1, s^2) where $Comp(s^1)$ and $Comp(s^2)$ are identical (same components).
- **Modified**: matched pairs (s^1, s^2) where $Comp(s^1)$ and $Comp(s^2)$ differ (some components were added or removed from the smell, but some are not changed).
- **Removed**: smells in S^{v_1} that have no match in S^{v_2}.
- **Added**: smells in S^{v_2} that have no match in S^{v_1}.

Since we aim to identify issues that induce smells, we target where smells **deteriorate**: added smells or modified smells with added component(s). We define these two cases as growing smells and use them for linking.

C. Issue-Component Linking

The remaining steps aim to link growing smells back to the implementation issues whose resolution contributed to their growth. Our key idea is to use components as the bridge between issues and smells. This step answers the question: for an issue, which component(s) did its implementation affect? Issue implementations are realized as commit(s) at the file level, whereas architectural smells are defined at the component level. We therefore first identify structurally relevant commits and then lift file-level changes to components.

Not all file changes are architecturally relevant. Developers can modify code without affecting structural dependencies. Since smells arise from inappropriate dependencies, changes to file-level dependencies are necessary. We therefore restrict our analysis to commits that modify dependency declarations, such as *import* or *include* statements. This coarse filter may still retain non-smell-inducing commits, which we remove in smell-analysis steps. For an issue i, let $Cmt(i)$ denote the set of linked commits that modify structural dependencies.

From each commit $c \in Cmt(i)$, we extract the set of files whose dependencies changed; let $F(c)$ denote this set. The union over all such commits is the structurally affected files of issue i: $F(i) = \{f \mid c \in Cmt(i),\ f \in F(c)\}$

Based on component-file mapping $F(c)$ (the set of files belonging to component c), we derive the affected components:

$$CompTrace(i) = \{c \mid F(c) \cap F(i) \neq \varnothing\} \tag{3}$$

Thus, for each issue i, we obtain the set of components whose structural dependencies were touched.

D. Growing Smells to Issue Linking

This step links growing smells to issues via components. Given a growing smells s with a step (v_i, v_{i+1}), we determine which components were changed in this smell. We distinguish by evolution category:

- For **modified** smells, the relevant components are the components newly added to the smell,
- For **added** smells, the relevant components are all components in the new smell instance in v_{i+1}.

We denote this set by $RelComp(s)$.

For an issue i, we quantify how strongly it is involved in the evolution of s by the ratio of relevant components that were touched by the issue (as given by Equation 3):

$$overlap(s, i) = \frac{|RelComp(s) \cap CompTrace(i)|}{|RelComp(s)|} \tag{4}$$

An $overlap$ of 1.0 means that all relevant components of the smell were modified in the issue; any positive value indicates partial component-level involvement in the issue.

We link an issue i to an evolved smell s on edge (v_i, v_{i+1}) if both of the following hold:

1) The commits implementing i fall within the time interval corresponding to the evolution from v_i to v_{i+1}, and
2) $overlap(s, i) > \gamma$, where γ is a threshold hyperparameter.

The final output is a set of annotated ASI issues:

$$\textit{ASI-Issues} = \{(i, s, \textit{overlap}(s, i)) \mid s \text{ is a growing smell}, \; i \text{ is linked to } s\}. \quad (5)$$

Each tuple $(i, s, \textit{overlap}(s, i))$ captures how strongly issue i contributed to the growth of smell s. This enables downstream analyses such as training classifiers to predict architectural smell-inducing issues from issue texts and studying characteristics of issues that tend to cause architectural decay.

IV. ArchGuard: Predicting Smell-Inducing Issues

Given the annotated set of architecturally smelly issues, we define the following prediction task: determining whether an issue will induce an architectural smell *during the design stage*, before any implementation occurs. This task relies solely on information available at design time: the issue title, description, and optionally early-stage discussions.

Our prediction pipeline extracts and leverages semantic signals from these textual artifacts. We investigate three established categories of predictive approaches: (i) encoding-based vector classification models, (ii) prompting decoder-based LLMs, and (iii) fine-tuning encoder-based LLMs. Each category offers distinct advantages. Encoding-based models are scalable and inexpensive to deploy, prompting-based approaches leverage the rich prior knowledge of pre-trained LLMs, and fine-tuned LLMs can exploit domain-specific patterns by adapting model parameters to the task.

A. Encoding-Based Classification

For encoding-based classification, we map each issue to a fixed-dimensional vector with an **encoder** and apply lightweight supervised **classifiers**. These classifiers offer low cost, high interpretability, and extensibility.

a) Encoders: Encoders transform texts into vectors suitable for classification. To cover the main categories of text representations, we use the following encoders

TF–IDF is a distribution-based vector representation for capturing token importance across all project texts. It computes the token frequency and then normalizes it by document frequency. As a statistical model, it is cheap to compute and requires no pre-training to obtain vectors.

Embeddings using small language models (SLMs), i.e., **Sentence-BERT (SBERT) embeddings**, fine-tuned on text similarity tasks, distinguish semantic meaning. For issues exceeding the 512 token context window of SBERT, we split the texts into chunks, encode each chunk independently, and aggregate chunk vectors using pooling (e.g., average, max).

Lastly, we use **LLM-based embeddings** (OpenAI's text-embedding-3-small) with an 8k-token context window.

b) Classifier: After constructing issue-level vectors, we apply vector-based classification models. We consider common classifiers such as **feed-forward neural network (ffNN)**, **naïve bayes (NB)**, **logistic regression (LR)**, **random forest (RF)**, **support vector machine (SVM)**, **XGBoost (XGB)** models, and **k-nearest neighbors (kNN)**. Their low inference cost and ease of tuning make these models suitable for large-scale deployment scenarios.

B. Fine-tuning Language Models

To evaluate the benefits of end-to-end learning, we fine-tune a RoBERTa model for binary classification of ASI versus non–ASI issues. Fine-tuning allows the model to learn project-specific terminology and cues. As with SentenceBert, issue texts often exceed RoBERTa's context window (512 tokens). Therefore, we need a design that ensures complete coverage of long issues, while enabling RoBERTa to analyze localized semantic signals within each chunk. Thus, we apply **Issue Chunking Aggregation**, where issues got splitted into chunks, classified separately, and aggregated. We use the mean of each chunk's result to obtain the result on issue-level. The impact of the design choices will be evaluated in RQ2.

C. Prompting LLMs

We complement supervised approaches with prompting LLMs to measure the intrinsic architectural reasoning capabilities of pretrained models. We set the system prompt to the smell definition and provide the LLM with the issue text and ask to classify whether the issue is likely to trigger architectural smell evolution. The prompt template can be found in replication package [31].

V. Research Design

In this paper, we investigate whether we can find and predict architectural smell-inducing issues. Our research design focuses on three aspects of ASI issues: their prevalence (RQ1), predictability (RQ2 & RQ3), and transferability (RQ4).

A. Research Questions

RQ1: *What proportion of evolved architectural smells can be traced to issues?*

Motivation: Before attempting prediction, it is important to understand the empirical evidence of smell evolution Quantifying this clarifies the feasibility of issue-based analysis of architectural decay.

RQ2: *How do different predictors **perform** in finding architectural smelly issues?*

Motivation: Even if architectural smell changes can be linked to issues, it remains unclear whether such issues contain enough distinctive linguistic signals for detection. If they do, different learning algorithms may capture different aspects of the characteristics of smell-inducing issues. Understanding which predictor performs well provides actionable insights for future algorithm enhancement and tool development.

RQ3: *How does the temporal availability of information influence the prediction performance of ASI issues?*

Motivation: Issue discussions evolve over time as more comments and details accumulate. The later we look in the issue's timeline, the more information becomes available. The earlier we look, the more useful the prediction can be for triggering timely architectural alarms. Ideally, we want to predict smell-inducing issues at issue creation time. However, if it improves performance, incorporating information from early pre-implementation discussions may also be beneficial.

TABLE II
DATASET OVERVIEW

Project	Ver.	Issues	MRs	Commits @MR	Commits @Repo	First issue	Last issue
Inkscape	5	5788	2828	6244	48571	2018-11-19	2025-07-06
StackGres	24	1581	1389	5891	9499	2019-07-02	2025-09-12
Shepard	8	346	447	3803	3351	2021-06-30	2025-10-07

RQ4: *How does **cross-project** prediction performance differ from within-project setting?*

Motivation: When new projects start, they may lack sufficient historical data for training supervised models. A useful model should generalize beyond the project on which it was trained. Cross-project evaluation can also tell us to what degree smell-related linguistic patterns are general or project-specific.

B. Experiment Design

To answer the RQs, we have four families of experiments.

1) Experiments for RQ1: To answer RQ1, we run smell detection, linking, and quantification on our datasets:

a) Smell evolution extraction: For each project, we apply the architectural smell detection tools Arcade [14] and Arcan [13]. We detect smell evolution between consecutive release versions and record the affected architectural components and files. For Arcan results, we use its ASTracker [32] plugin to obtain smell evolution results. For Arcade results, we implement an evolution analyzer, available in our replication package [31]. We set the Jaccard similarity threshold (Equation 2) to 0.3. This value is lower than ASTracker's default, as Arcade recovers high-level components from files, but Arcan operates at package/class level. Therefore, with the default threshold, Arcade components will result in fewer matches.

b) Architectural smell-inducing issue annotation: From our collected data, we use explicit trace links provided by the GitLab platform to associate commits with issues. Therefore, issues under consideration are those linked with at least one commit. We set the smell-issue overlap threshold γ to 0, meaning we include any non-zero smell-issue involvement as defined in Equation 4. Based on our approach, commits/issues linked with at least one smell will be marked as smell-inducing, while remaining issues are marked as non-smelly.

c) Quantification: We compute: (i) the number and proportion of smell changes that are linked to issues, (ii) how many issues are smell-inducing vs. how many aren't.

The outcome of RQ1 is a characterization of the prevalence and nature of issue-linked smell evolution. It provides the target class distribution for the prediction tasks in RQ2 & RQ3.

2) Experiments for RQ2: For RQ2, we regard predicting architectural smell-inducing issues as a binary classification task. We consider the issues labeled as smell-inducing by the two smell detectors (Arcade & Arcan) separately, as they can identify different smells and are applicable to different projects. We evaluate multiple embedding models for each supervised classifier: For TF-IDF encoding, we conduct standard data cleaning such as stemming and set the max-vocabulary size to 50,000. For SBERT embeddings, we use *sentence-transformers/paraphrase-mpnet-base-v2*. For OpenAI embeddings, we use the *text-embedding-3-small* model. Our replication package contains a table listing all hyperparameters for the supervised classifiers. We include a random baseline that predicts labels based on the smell frequency in the data.

We conduct two runs of a five-fold cross-validation with train, test splits of $80\%, 20\%$. We use average precision, recall, and F_1-scores across the runs and folds to measure performance for this and the following experiments. Additionally, we conduct a Wilcoxon signed-rank test to compare against a random baseline, which predicts based on frequency.

Unlike typical classification setups, we define a successful identification only as correctly identifying positive samples, i.e., finding an ASI issue. This focus is both most relevant to developers and also the most challenging setting. As Table I shows, only a small percentage of issues are actually ASI issues. Because non-ASI issues dominate the dataset, predicting the negative class is comparatively easy, and high performance on non-ASI issues can be achieved even with simple baselines. Including such metrics and averaging over both classes would yield good-looking but misleading results, as the overall scores would be dominated by the non-ASI class. Thus, throughout RQ2-RQ4, we report metrics only for the positive (ASI) class.

3) Experiments for RQ3: This research question investigates how varying information availability affects the predictive performance of architectural smell-inducing issues. We evaluate the same task as in RQ2 but under two different temporal conditions:

1) *Condition C_0: No discussion (planning-only).* The model sees only the issue title and description (used in RQ2). This approximates the situation at issue creation.
2) *Condition C_{jit}: Just-in-time information.* The model also sees comments that were posted *before* the creation of the first related merge request. This simulates a decision point where implementation is about to start. Our prediction here can give hints to developers about architectural aspects to consider during implementation.

4) Experiments for RQ4: We examine ArchGuard's generalizability across projects: We use the same supervised model settings as RQ2. Instead of within-project cross-validation, we perform leave-one-project-out evaluation. The idea is that for data from the same smell detector, we keep one project out for test, and do train and validation completely on the remaining projects. For example, for ASI issues detected with Arcade, we use all issues of Shepard as a test, and train on Inkscape and StackGres issues.

C. Dataset Construction

We build our dataset from GitLab-hosted projects for three reasons: 1) Arcade tool is tightly coupled with Gitlab mining workflow; 2) GitLab provides explicit linking between issues, merge requests (MRs), and commits; while on GitHub issue—commit links are often implicit, making trace reconstruction

TABLE III
RESULTS FOR RQ1: SMELL EVOLUTION LINKAGE STATISTICS

		Growing Smells			Linked Smells					Linked Issues				
Project	**Steps**	**n**	**mean**	**std**	**total**	**n**	**total%**	**mean%**	**std**	**total**	**n**	**total%**	**mean**	**std**
Inkscape (Arcade)	4	20	5.0	± 2.3	20	19	95.0%	96.4%	± 7.1	2536	257	10.13%	64.3	± 43.7
StackGres (Arcade)	23	19	0.8	± 1.4	19	17	89.5%	88.9%	± 33.3	1581	279	17.65%	12.1	± 32.9
StackGres (Arcan)	19	211	11.1	± 17.0	211	87	41.2%	56.0%	± 36.0	1581	237	15.0%	13.1	± 13.2
Shepard (Arcade)	7	4	0.6	± 1.1	4	4	100.0%	100.0%	± 0.0	346	46	13.3%	6.7	± 17.3
Shepard (Arcan)	5	143	28.6	± 49.2	143	142	99.3%	99.8%	± 0.4	346	120	34.7%	24.0	± 40.3

harder; 3) Gitlab provides higher API rate limits than Github, accelerating dataset construction.

We target projects in Java, because Arcade tool supports C/C++ and Java language, and the open-source version of Arcan supports only Java. We additionally include a C-based program, Inkscape, as it is the recommended example in the ARCADE documentation. *Inkscape* [33] is an open-source vector graphics editor.

For Java-based projects, we applied the following inclusion/exclusion criteria:

Inclusion criteria. A project had to (i) be hosted on GitLab, (ii) declare *Java* as a project topic, (iii) rank among the top 50 projects by number of stars (visibility), (iv) have at least 500 merge requests (enough activities), and (v) be an actual software application.

Exclusion criteria. We excluded repositories that are not production systems (e.g., datasets, benchmarks, or task collections). Applying these criteria yielded two Java projects: *StackGres* [34], a Kubernetes operator and *Shepard* [35] a heterogeneous research data management system [36].

Table II summarizes the collected artifacts per project. We collected data in October 2025; final data includes 37 releases, totaling 7,715 issues, and 4,664 MRs.

The analyzed issues span from 2018-11-19 to 2025-10-7. For each project, the table shows the number of releases (Ver.), issues, merge requests (MR), commits in the MRs (Commits@MR), commits within the git history (Commits@Repo), dates of the first issues and the last issues. Commits@MR and Commits@Repo overlaps depending on the workflow of developers. Commits directly pushed to or cherry-picked from other branches are not tracked by MRs. Multiple Commits@MR can also be squashed into a single commit inside Repo.

VI. RESULTS

A. RQ1: Smell Traceability

RQ1 examines the proportion of evolved smells able to be traced to issues. Table III presents traceability rates. Inkscape is analyzed by Arcade alone because the open source version of Arcan supports Java only. We were not able to analyze all versions with Arcan. It extracts dependencies from compiled artifacts (JARs/class files) and therefore requires successful builds. Due to a missing dependency, we were unable to build two versions of Shepard (”2024.08.05 Prepare quarkus migration”, and ”2024.07.04 Monorepo”) and four versions of StackGres(0.5, 0.6, 0.7, 1.18.0). Arcade operates at the Java and C/C++ source-code level, enabling analysis of all versions.

In general, Arcade yields fewer evolved smells and higher traceability (roughly 90–100%). For Arcan, results are mixed: *Shepard* still exhibits near-complete traceability, whereas *StackGres* is substantially worse, likely due to changes out of issue management. This reflects the project characteristics shown in Table II, Inkscape and StackGres contain far more commits inside the repository than in MR, showing that many commits were outside the issue management. The number of detected smells also differs, because Arcan operates at the class/package level, whereas Arcade recovers higher-level components via clustering.

Our traceability approach linked many smells to their inducing issues, though not always 100%.

Answer to RQ1. Across our dataset, **41.2%–100%** of *growing smells* are traceable to issues. Traceability is lowest for *StackGres* under Arcan (87/211), and near-complete for *Shepard* (Arcade: 4/4; Arcan: 142/143).

B. RQ2: Prediction Performance

RQ2 examines the performance of classifiers in predicting architectural smell-inducing issues. We evaluated **supervised machine learning models** (seven classifiers combined with three embeddings), **finetuned RoBERTa models** and **Zero-shot LLMs based classifiers**.

For the smell-inducing issues based on Arcade (Table IV), SVM with OpenAI embeddings achieves the best average performance with an F_1-score of .360 (P=.243, R=.710). It exhibits strong recall (average .710) while maintaining the second-highest precision across all three systems, particularly excelling on the StackGres project with an F_1-score of .415. Only three configurations achieve higher average recall (ffNN, kNN, and LR, each with OpenAI embeddings as well). However, they are not as precise as SVM, resulting in lower average F_1-scores. Finetuned RoBERTa models perform comparably to supervised ML models. Zero-shot classifier using GPT-5-mini performs worse, achieving slightly higher precision than random guessing. This demonstrates that the intrinsic architectural reasoning capabilities of LLMs alone are insufficient without learning from project-specific patterns. Our significance tests show that, except for the zero-shot classifier, all perform significantly better than the baseline. Their p-values are available in the replication package [31].

TABLE IV
RESULTS OF THE TOP-2 CONFIGURATION OF EACH CLASSIFIER ON ISSUES INDUCING ARCADE DETECTED SMELLS. CLASSIFIERS ARE DESCRIBED IN SECTION V-B.

Class.	Emb.	Inkscape			StackGres			Shepard			Avg.		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
ffNN	OpenAI	.158	.652	.251	.290	.704	.409	.180	.800	.286	.210	.719	.315
	SB_{max}	.150	.591	.229	.275	.704	.392	.169	**.823**	.279	.198	.706	.300
kNN	OpenAI	.141	**.694**	.235	.274	.751	.401	.178	.816	.292	.198	**.753**	.309
	SB_{max}	.134	.589	.218	.260	.692	.378	.206	.724	.319	.200	.668	.305
LR	OpenAI	.160	.615	.253	.271	.776	.402	.215	.772	.334	.215	.721	.330
	TF-IDF	.160	.523	.244	**.307**	.717	**.429**	.221	.661	.330	.229	.634	.334
NB	OpenAI	.164	.592	.257	.272	.763	.401	.252	.627	.357	.230	.661	.338
	SB_{max}	.138	.549	.220	.270	.699	.389	.254	.683	.366	.221	.644	.325
RF	OpenAI	.165	.588	.257	.280	.722	.403	.234	.746	.354	.226	.685	.338
	SB_{max}	.138	.581	.222	.284	.708	.405	.212	.683	.322	.211	.658	.316
SVM	OpenAI	**.195**	.601	**.294**	.280	**.803**	.415	.253	.726	.371	**.243**	.710	**.360**
	SB_{max}	.150	.591	.238	.284	.733	.410	**.257**	.727	**.376**	.230	.684	.341
XGB	OpenAI	.163	.603	.257	.291	.708	.412	.223	.640	.329	.226	.650	.333
	SB_{max}	.139	.580	.225	.279	.683	.396	.222	.697	.335	.213	.653	.319
$RoBERTa_{fine-t.}$		.133	.593	.210	.296	.656	.401	.134	.822	.228	.187	.690	.334
GPT-5-mini		.167	.012	.023	.176	.011	.020	.222	.043	.073	.188	.022	.066
Random		.078	.080	.078	.196	.202	.199	.183	.207	.191	.152	.163	.156

TABLE V
RESULTS OF THE TOP-2 CONFIGURATION OF EACH CLASSIFIER ON THE ISSUES INDUCING ARCAN DETECTED SMELLS. CLASSIFIERS ARE DESCRIBED IN SECTION V-B.

Class.	Emb.	StackGres			Shepard			Avg.		
		P	R	F_1	P	R	F_1	P	R	F_1
ffNN	OpenAI	.264	.692	.381	.482	.713	.572	.373	.702	.477
	SB_{TF-IDF}	.247	.726	.367	.452	.658	.532	.349	.692	.449
kNN	OpenAI	.229	**.805**	.357	.490	**.758**	.594	.360	**.782**	.475
	SB_{max}	.219	.745	.338	.470	.725	.570	.345	.735	.454
LR	OpenAI	.255	.768	.383	.501	.658	.567	.378	.713	.475
	TF-IDF	.272	.698	.390	.511	.667	.576	.391	.683	.483
NB	OpenAI	.267	.724	.389	.518	.688	.589	.393	.706	.489
	TF-IDF	.252	.798	.382	.466	.654	.542	.359	.726	.462
RF	OpenAI	.273	.730	.396	.533	.683	.595	.403	.707	.496
	SB_{TF-IDF}	.265	.690	.383	.456	.650	.535	.360	.670	.459
SVM	OpenAI	.271	.766	**.400**	.538	.717	**.612**	.405	.741	**.506**
	SB_{max}	.252	.736	.375	.523	.638	.570	.387	.687	.472
XGB	OpenAI	**.275**	.698	.394	.506	.696	.583	.391	.697	.488
	SB_{max}	.254	.703	.372	.501	.667	.570	.377	.685	.471
$RoBERTa_{fine-t.}$		.243	.639	.350	.451	.525	.480	.347	.582	.415
GPT-5-mini		.273	.013	.024	**.833**	.042	.080	**.553**	.027	.052
Random		.116	.122	.118	..357	.346	.349	.236	.234	.234

Results for smell-inducing issues based on Arcan (Table V) show better overall performance compared to Arcade. SVM with OpenAI embeddings remains the best configuration, with an average F_1-score of .506 (P=.405, R=.741). This configuration demonstrates strong performance across systems, with the highest individual F_1-scores on both projects (.400 on StackGres and .612 on Shepard). kNN with OpenAI embeddings again achieves the highest recall (.782 average) but low precision (.360 average), resulting in an F_1-score of .475.

Across both datasets, OpenAI embeddings combined with an SVM consistently outperform, suggesting that advanced pre-trained embeddings, paired with project-specific training, are most effective for predicting ASI issues.

Answer to RQ2. SVM with OpenAI embeddings achieves the best overall performance, with a balance of precision and recall and the highest average F_1-scores. Zero-shot LLMs without project-specific training performs poorly.

C. RQ3: Temporal Information Availability

RQ3 investigates the impact of information availability on prediction performance by comparing two temporal conditions: C_0 (title and description only, available at issue creation, as used in RQ2) versus C_{jit} (including pre-implementation comments). Table VI presents results for the best-performing configuration (SVM with OpenAI embeddings).

The results show minimal differences between the two conditions. For smell-inducing issues based on Arcade detection, additionally using pre-implementation discussions (C_{jit}) achieves nearly identical performance to using only title and description (C_0): precision changes by +.002 to .241, recall increases by +.011 to .721, and F_1-score remains at .360. For Arcan, the added information yields a marginal decrease in average F_1-score to .505 (compared to .506), with precision at .406 (+.001) and recall at .737 (-.004).

TABLE VI
AVG. RESULTS WITH ADDIT. DISCUSSION INPUT (C_{JIT}) COMPARED TO PERFORMANCE ON C_0. RESULTS FOR THE BEST CONFIGURATION FROM RQ2 (T&D) AND THE BEST PERFORMING ON THE ADDIT. DATA (FULL).

Smell-detector	Config. top on	Class.	Emb.	Average					
				P	Δ	R	Δ	F_1	Δ
Arcade	T&D	SVM	OpenAI	.241	(–.002)	.721	(+.011)	.360	(±.000)
	Full								
Arcan	T&D	SVM	OpenAI	.406	(+.001)	.737	(–.004)	.505	(–.001)
	Full								

Answer to RQ3. Information available at issue creation (issue titles and descriptions) alone contains sufficient information for predicting ASI issues; later signals (pre-implementation discussions) only contribute negligibly.

D. RQ4: Cross-Project Generalizability

RQ4 investigates the transferability of trained models across different projects. Table VII presents results for the best-performing configuration from RQ2 (SVM with OpenAI embeddings) in cross-project settings. The results show substantial performance degradation in cross-project scenarios compared to within-project evaluation, although still better than random baseline. For Arcade-based smells, all three projects show reduced performance. Notably, Shepard's recall increases by 2% to .739, maintaining the ability to identify

TABLE VII
GENERALIZATION PERFORMANCE IN CROSS-PROJECT SETTING (RELATIVE CHANGE TO RQ2)

Detector	Dataset	Precision (Δ)	Recall (Δ)	F_1 (Δ)
Arcade	Inkscape	.115 (−.080)	.070 (−.531)	.087 (−.207)
	StackGres	.209 (−.071)	.297 (−.506)	.246 (−.169)
	Shepard	.181 (−.072)	.739 (+.013)	.291 (−.080)
	Average	.168 (−.074)	.369 (−.341)	.208 (−.152)
Arcan	StackGres	.210 (−.061)	.561 (−.205)	.305 (−.095)
	Shepard	.439 (−.099)	.150 (−.567)	.224 (−.388)
	Average	.324 (−.080)	.356 (−.386)	.265 (−.242)

ASI issues, but suffers a 40% precision loss (to .181) and a 27% decline in F_1-score. For Arcan-based smells, both projects show performance losses.

Answer to RQ4: Cross-project models show performance degradation (F_1 drop ranging from 0.095 to 0.388) compared to within-project evaluation, indicating that smell-inducing textual patterns are largely project-specific.

VII. DISCUSSION

A. Implications

For practitioners, our approach enables proactive architectural quality assurance. By predicting smell-inducing issues at creation time using only issue titles and descriptions, development teams can receive early warnings about potential architectural risks during planning, without waiting for discussions or implementation. This allows teams to take preventive actions, such as exploring design alternatives while they are still cheap to evaluate: (i) pay attention to the planned dependency structure, (ii) conduct an architecture-focused review of the intended changes, and (iii) write constraints as ArchUnit rules [37] to prevent architectural risks. High recall rates of our models suggest that major risks can be flagged. Teams can use predictions as a lightweight filter rather than a definitive verdict. Based on our results, projects with strong issue-commit linking (Shepard) yield better smell traceability and prediction, while projects with significant off-issue development (StackGres) are harder to analyze. Teams adopting ArchGuard should improve issue-commit linking.

For researchers, our work establishes a new direction for studying architectural decay. By focusing on smell-inducing issues, we shift attention from the consequences of decay to its root causes. This opens opportunities to understand which types of development activities tend to introduce smells and to better integrate architectural knowledge into development workflows. Our traceability approach provides a methodology for constructing datasets that link implementation decisions to architectural outcomes, thus supporting future studies on architecture evolution and technical debt management. The pipeline can be extended to study drivers of smell evolution on other platforms (e.g., GitHub), where projects are more diverse. It can also be extended to other smell types.

B. Threats to Validity

Following the guidelines of Runeson and Höst [38], we outline and address potential threats to the validity.

External Validity. The generalizability of our findings is limited by the characteristics of the analyzed systems. We conducted our study on three open-source projects that, while actively maintained and representing different domains, may not represent the full spectrum of software development contexts. Our results may not fully transfer to projects with substantially different characteristics: (i) very large-scale systems with complex organizational structures, (ii) proprietary or industrial systems with different development workflows and architectural governance practices, or (iii) projects using different issue-tracking conventions or development processes. To mitigate this threat, we selected projects across diverse domains and sizes, applied two different smell-detection tools, and documented our process to facilitate replication.

Internal Validity. A primary threat to internal validity concerns the accuracy of the temporal information used to associate issues with architectural smells. We rely on platform timestamps (e.g., merge request creation and merge dates) to approximate when changes were implemented. However, these timestamps may not precisely reflect actual dates: developers may implement changes locally before opening an MR, or commit history may be squashed. Our traceability approach also depends on explicit issue-commit links, which may be incomplete due to non-standard workflows (e.g., cherry-picking).

Construct Validity. We use architectural smells detected by Arcade and Arcan as proxies for instances of architectural decay. This assumes that the smells identified by these tools are valid indicators of architecture-level quality issues. While prior studies have evaluated the precision and usefulness of these detectors [11], [15], no automated tool is perfect. To mitigate this threat, we used two validated smell detectors and manually inspected a sample of detections to reduce the impact of false positives or negatives.

C. Limitation and Future Work

Improving traceability coverage. We were unable to trace all growing smells back to issues due to changes outside of issues. Git workflows, such as cherry-picking or direct pushes, can introduce such changes [39]. Future work can combine the traceability recovery approach [39] (e.g., link inference from commits) to reconstruct links and improve coverage.

Increasing cross-project variety. Our cross-project evaluation assesses generalizability in a simple setting. Future work could explore more complex yet realistic settings, such as the relationship between training/test projects. Potential hypotheses include that "similar projects," such as those from the same team or in the same domain, are more transferable.

Deeper analysis of smell-inducing issues. While we identified which issues induce smells, we have a limited understanding of their characteristics. Future work could investigate correlations with issue metadata (e.g., labels, assignees,

complexity metrics) to understand what distinguishes smell-inducing issues from others.

Ensemble model predictions. We evaluated individual classifiers and embeddings independently. Ensemble approaches that combine multiple models or leverage both semantic embeddings and traditional features could improve predictions.

Integrating intentional decisions. Some smells may be introduced intentionally as pragmatic trade-offs [12]. Our approach does not distinguish between deliberate and accidental architectural degradation, which could lead to false positives where developers consciously accept technical debt. Further work can consider this to build a cleaner dataset.

Better prompt engineering. We use LLMs as zero-shot classifiers, without prompt engineering or other LLM-specific optimization techniques (e.g., retrieval augmentation, chain-of-thought), which might underestimate their performance. However, LLM-based tools have been widely and successfully adopted for coding tasks in zero-shot settings. This difference suggests that further investigation is needed to understand why software-architecture-related tasks are particularly challenging for LLMs compared with code-centric tasks.

VIII. Related Work

Architectural decay can manifest as architectural smells [30] or as inconsistency between prescribed and implemented architecture models [40], [41]. Most tools for detecting architectural smell instances in systems are based on static analysis on graphs [10], [13], [42]–[45], or patterns [20]. Existence of smells can also be predicted by smell evolution history [10], [46]. Beyond mere detection, architectural smells and architectural technical debt can be ranked by their potential risk using indexing [22], [47], [48] or estimated maintenance costs [16]. A transformer-based approach has also been proposed to recommend refactorings for smells [49]. Unlike these works, we aim to determine which implementation will cause architectural smells, enabling earlier, fine-grained predictions.

The consequences of architectural smells include increased defect [50], [51] and change proneness [52]–[54], higher memory consumption and response time [55] and more issues [15]. The negative impact of architectural smells is also reported by industry practitioners [11], [56]. In a grounded-theory study of Stack Overflow posts, Tian et al. [57] report that developers describe architectural smells using general, high-level terms, and further analyze how developers discuss smells in online forums [58]. These findings collectively indicate that semantic and textual cues in developer communication carry information about architectural problems.

Several studies explicitly exploit such semantic information to identify architecture-related problems. Shahbazian et al. [59] propose an approach to predict the architectural significance of issues. Similarly, Li et al. automatically identify code review comments related to architecture erosion [60] and characterize symptoms of architecture violations in these comments [61]. In a complementary study, Li et al. [62] investigate practitioners' perceptions of architecture erosion, its causes and consequences, and the supporting tools and practices used to identify and control erosion.

To preserve architecture consistency [63] and ensure conformance, a related but distinct research study investigates how to warn developers about architectural violations. Buckley et al. proposed JITTAC for identifying architectural violations based on Reflexion Modelling [64], Their approach's benefits are shown through multi-case studies [65], [66]. Our work addresses a different problem. Their focus is on architectural drift, where the implemented architecture differs from the designed one. In contrast, we target architectural decay by warning developers about potential smell-inducing issues. Both directions are necessary and complementary

Traceability in software architecture has been studied with respect to architecture models, documentation, and code [67]–[71]. These approaches focus on recovering links between architectural descriptions and implementation artifacts in a static setting. In contrast, we shift the focus of traceability for architectural smells from code and models to issues: we link architectural smells to their inducing issues and study how far the textual and semantic characteristics of issues can be used to predict whether an issue is smell-inducing.

IX. Conclusion

This paper addresses the challenge of architectural decay by identifying and predicting architectural smell-inducing issues, i.e., issues arising from implementation that intensify architectural smells. We developed and evaluated a traceability approach that links architectural smells to their inducing issues by combining smell evolution analysis with issue-commit traceability. Building on this foundation, we introduced ArchGuard. It classifies whether an issue will induce architectural smells using only information available at issue creation time: the issue title and description.

By enabling prediction at issue creation, ArchGuard provides developers with early warnings of potential architectural risks. This proactive approach shifts developers' focus from reactive remediation to active awareness of architectural risks. With the awareness, developers can make a more careful architectural plan, potentially reducing the accumulation of architectural technical debt.

Data Availability Statement

Code, data, and results are in the replication package [31].

Acknowledgements

This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – project number 501930651, NFDIxCS and under - SFB 1608 - 501798263. It was also supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, Jun. 2021.

[2] E. Woods, M. Erder, and P. Pureur, *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*. Addison-Wesley Professional, May 2021.

[3] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 546–556, iSSN: 2576-3148.

[4] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, "Evolution styles: Foundations and tool support for software architecture evolution," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, Sep. 2009, pp. 131–140.

[5] H. Muccini and K. Vaidhyanathan, "Software Architecture for ML-based Systems: What Exists and What Lies Ahead," in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. Madrid, Spain: IEEE Press, May 2021, pp. 121–128.

[6] H. P. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, Jan. 2012.

[7] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[8] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts*. "O'Reilly Media, Inc.", Sep. 2021.

[9] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding architecture erosion: The practitioners' perceptive," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 311–322.

[10] J. Garcia, E. Kouroshfar, N. Ghorbani, and S. Malek, "Forecasting architectural decay from evolutionary history," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2439–2454, 2022.

[11] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells—an industrial case study," *Empirical Software Engineering*, vol. 27, no. 4, p. 86, 2022.

[12] D. Sas, P. Avgeriou, and F. Arcelli Fontana, "Investigating Instability Architectural Smells Evolution: An Exploratory Case Study," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 557–567, iSSN: 2576-3148.

[13] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 282–285.

[14] M. Schmitt Laser, N. Medvidovic, D. M. Le, and J. Garcia, "ARCADE: An extensible workbench for architecture recovery, change, and decay evaluation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1546–1550.

[15] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[16] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Detecting the Locations and Predicting the Maintenance Costs of Compound Architectural Debts," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3686–3715, Sep. 2022.

[17] J. Maarleveld, A. Dekker, S. Druyts, and M. Soliman, "Maestro: A Deep Learning Based Tool to Find and Explore Architectural Design Decisions in Issue Tracking Systems," in *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium*, B. Tekinerdoğan, R. Spalazzese, H. Sözer, S. Bonfanti, and D. Weyns, Eds. Cham: Springer Nature Switzerland, 2024, pp. 390–405.

[18] M. Soliman, M. Galster, and P. Avgeriou, "An Exploratory Study on Architectural Knowledge in Issue Tracking Systems," in *Software Architecture*, S. Biffl, E. Navarro, W. Löwe, M. Sirjani, R. Mirandola, and D. Weyns, Eds. Cham: Springer International Publishing, 2021, pp. 117–133.

[19] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, May 2018.

[20] E. Ntentos, U. Zdun, G. Falazi, U. Breitenbücher, and F. Leymann, "Detecting and resolving coupling-related infrastructure as code based architecture smells in microservice deployments," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 201–211.

[21] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, Mar. 2015.

[22] D. Sas and P. Avgeriou, "An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4169–4195, Aug. 2023.

[23] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52–73, Aug. 2015.

[24] A. Martini, J. Bosch, and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2014, pp. 85–92.

[25] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 179–188, iSSN: 1558-1225.

[26] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 488–498.

[27] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[28] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering Architectural Design Decisions," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 95–9509.

[29] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536–557, Apr. 2008.

[30] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a Catalogue of Architectural Bad Smells," in *Architectures for Adaptive Software Systems*, R. Mirandola, I. Gorton, and C. Hofmeister, Eds. Berlin, Heidelberg: Springer, 2009, pp. 146–162.

[31] H. Liu, D. Fuchß, S. Corallo, M. Hummel, J. Keim, and T. Hey, "Replication Package: Architecture in the Cradle: Early Warning of Architectural Decay with ArchGuard," 2026. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.18891082

[32] "ASTracker," GitHub repository, accessed: 2026-03-01. [Online]. Available: https://github.com/darius-sas/astracker

[33] "Inkscape," GitLab repository, accessed: 2026-03-01. [Online]. Available: https://gitlab.com/inkscape/inkscape

[34] "StackGres," GitLab repository, accessed: 2026-03-01. [Online]. Available: https://gitlab.com/ongresinc/stackgres

[35] "shepard," GitLab repository, accessed: 2026-03-01. [Online]. Available: https://gitlab.com/dlr-shepard/shepard

[36] T. Haase, R. Dr. Glück, P. Kaufmann, and M. Willmeroth, "shepard - storage for heterogeneous product and research data," Dec. 2025, language: eng. [Online]. Available: https://zenodo.org/records/17897485

[37] TNG Technology Consulting GmbH, "Archunit: A java architecture test library," latest release: 1.4.1 (May 7, 2025). Accessed: 2026-02-26. [Online]. Available: https://github.com/TNG/ArchUnit

[38] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2008.

[39] J. Businge, M. Openja, S. Nadi, and T. Berger, "Reuse and maintenance practices among divergent forks in three software ecosystems," *Empirical Software Engineering*, vol. 27, no. 2, p. 54, Mar. 2022.

[40] N. Ghorbani, T. Singh, J. Garcia, and S. Malek, "Darcy: Automatic architectural inconsistency resolution in java," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1639–1657, 2024.

[41] N. Ghorbani, J. Garcia, and S. Malek, "Detection and repair of architectural inconsistencies in java," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 560–571.

[42] A. Tommasel and J. A. Diaz-Pace, "Identifying emerging smells in software designs based on predicting package dependencies," *Engineering Applications of Artificial Intelligence*, vol. 115, p. 105209, 2022.

[43] A. Martini and J. Bosch, "Architectural Technical Debt in Embedded Systems," in *Systems Engineering in the Fourth Industrial Revolution: Big Data, Novel Technologies, and Modern Systems Engineering*. Wiley, 2020, pp. 77–103.

[44] A. Bakhtin, M. Esposito, V. Lenarduzzi, and D. Taibi, "Leveraging Network Methods for Hub-like Microservice Detection," Jun. 2025, arXiv:2506.07683 [cs].

[45] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1008–1028, 2021.

[46] F. Arcelli Fontana, P. Avgeriou, I. Pigazzini, and R. Roveda, "A Study on Architectural Smells Prediction," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2019, pp. 333–337. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8906714

[47] J. A. Diaz-Pace, A. Tommasel, I. Pigazzini, and F. A. Fontana, "Sen4smells: A tool for ranking sensitive smells for an architecture debt index," in *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*, 2020, pp. 1–7.

[48] R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, "Atdx: Building an architectural technical debt index," in *ENASE 2020*, R. Ali, H. Kaindl, and L. Maciaszek, Eds. SciTePress, May 2020, pp. 531–539, 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020 ; Conference date: 05-05-2020 Through 06-05-2020.

[49] S. Nursapa, A. Samuilova, A. Bucaioni, and P. T. Nguyen, "ROSE: Transformer-Based Refactoring Recommendation for Architectural Smells," Jul. 2025, arXiv:2507.12561 [cs].

[50] D. A. Tamburri, F. A. Fontana, R. Roveda, and V. Lenarduzzi, "Architecture Smells vs. Concurrency Bugs: an Exploratory Study and Negative Results," Mar. 2023, arXiv:2303.17862 [cs].

[51] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, May 2015, pp. 51–60.

[52] L. Aversano, U. Carpenito, and M. Iammarino, "An empirical study on the evolution of design smells," *Information*, vol. 11, no. 7, 2020.

[53] D. Sas, P. Avgeriou, I. Pigazzini, and F. Arcelli Fontana, "On the relation between architectural smells and source code changes," *Journal of Software: Evolution and Process*, vol. 34, no. 1, p. e2398, 2022.

[54] D. Sas, P. Avgeriou, R. Kruizinga, and R. Scheedler, "Exploring the relation between co-changes and architectural smells," *SN Computer Science*, vol. 2, no. 1, p. 13, Dec 2020.

[55] F. Arcelli Fontana, M. Camilli, D. Rendina, A. G. Taraboi, and C. Trubiani, "Impact of Architectural Smells on Software Performance: an Exploratory Study," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 22–31.

[56] D. Sas, I. Pigazzini, P. Avgeriou, and F. A. Fontana, "The Perception of Architectural Smells in Industrial Practice," *IEEE Software*, vol. 38, no. 6, pp. 35–41, Nov. 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9509940

[57] F. Tian, F. Lu, P. Liang, and M. A. Babar, "Automatic identification of architecture smell discussions from stack overflow," in *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, USA, July 9-19, 2020*, R. García-Castro, Ed. KSI Research Inc., 2020, pp. 451–456.

[58] F. Tian, P. Liang, and M. A. Babar, "How developers discuss architecture smells? an exploratory study on stack overflow," in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 91–100.

[59] A. Shahbazian, D. Nam, and N. Medvidovic, "Toward predicting architectural significance of implementation issues," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 215–219.

[60] R. Li, M. Soliman, P. Liang, and P. Avgeriou, "Symptoms of architecture erosion in code reviews: A study of two openstack projects," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 2022, pp. 24–35.

[61] R. Li, P. Liang, and P. Avgeriou, "Warnings: Violation symptoms indicating architecture erosion," *Information and Software Technology*, vol. 164, p. 107319, 2023.

[62] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding architecture erosion: The practitioners' perceptive," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 311–322.

[63] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley, "Architecture consistency: State of the practice, challenges and requirements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 224–258, 2018.

[64] J. Buckley, S. Mooney, J. Rosik, and N. Ali, "Jittac: A just-in-time tool for architectural consistency," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1291–1294.

[65] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold, "Real-time reflexion modelling in architecture reconciliation: A multi case study," *Information and Software Technology*, vol. 61, pp. 107–123, 2015.

[66] N. Ali, J. Rosik, and J. Buckley, "Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study," in *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 23–32. [Online]. Available: https://doi.org/10.1145/2304696.2304702

[67] D. Fuchß, H. Liu, T. Hey, J. Keim, and A. Koziolek, "Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction," in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, Mar. 2025, pp. 1–12, iSSN: 2835-7043.

[68] D. Fuchß, H. Liu, S. Corallo, T. Hey, J. Keim, J. von Geisau, and A. Koziolek, "Who's Who? LLM-assisted Software Traceability with Architecture Entity Recognition," Nov. 2025.

[69] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Recovering Trace Links Between Software Documentation And Code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13.

[70] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolek, "Trace Link Recovery for Software Architecture Documentation," in *Software Architecture*, S. Biffl, E. Navarro, W. Löwe, M. Sirjani, R. Mirandola, and D. Weyns, Eds. Cham: Springer International Publishing, 2021, pp. 101–116.

[71] D. Fuchß, T. Hey, J. Keim, H. Liu, N. Ewald, T. Thirolf, and A. Koziolek, "LiSSA: Toward Generic Traceability Link Recovery Through Retrieval-Augmented Generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Apr. 2025, pp. 1396–1408, iSSN: 1558-1225.