Architectural Decay as Predictor of Issue- and Change-Proneness

Duc Minh Le*, Suhrid Karthik†, Marcelo Schmitt Laser†, and Nenad Medvidovic†
Software Infrastructure*
Bloomberg L.P.
London, EC4N 4TQ, UK
dle50@bloomberg.net

Computer Science Department†
University of Southern California
Los Angeles, CA 90089, USA
{skarthik,schmittl,neno}@usc.edu

***Abstract*—Architectural decay imposes real costs in terms of developer effort, system correctness, and performance. Over time, those problems are likely to be revealed as explicit implementation issues (defects, feature changes, etc.). Recent empirical studies have demonstrated that there is a significant correlation between architectural "smells"—manifestations of architectural decay—and implementation issues. In this paper, we take a step further in exploring this phenomenon. We analyze the available development data from 10 open-source software systems and show that information regarding *current* architectural decay in these systems can be used to build models that accurately predict *future* issue-proneness and change-proneness of the systems' implementations. As a less intuitive result, we also show that, in cases where historical data for a system is unavailable, such data from other, unrelated systems can provide reasonably accurate issue- and change-proneness prediction capabilities.**

***Index Terms*—Architectural Decay, Issue Proneness, Change Proneness, Architectural Smell, Decay Prediction**

I. Introduction

Software systems change regularly, as do their architectures. Over time, a system's architecture is increasingly affected by decay, caused by careless or unintended design decisions [53]. Decay results in systems whose implemented architectures differ in important ways from their designed architectures. Both researchers and practitioners have recognized the negative impact of architectural decay and its role in causing technical debt. Despite this, when developers modify a system during maintenance, they often focus on code and neglect the architecture.

Researchers have proposed a number of techniques to analyze a system at the code level and to predict issues that are likely to appear in the system's future versions. A common approach has been to use historical artifacts, such as data from issue trackers and version control systems, to build prediction models. Early approaches [29], [47], [24], [13] built models to predict implementation issues based on code metrics. Later studies made use of other properties that were reckoned to be potential causes of issues, such as code dependencies [73] and code smells [25].

In contrast to code-level techniques, analogous techniques at the architecture level have not received nearly as much attention, even though recent work has demonstrated that even simple code updates can cause system-wide architectural changes [36]. Frequently, such updates introduce *architectural smells* in a system (e.g., dependency cycle, ambiguous interface [35]). These smells may have no immediately visible effect, but they are symptoms of architectural decay and accumulated technical debt [64], [18], [36], [35]. As decay compounds in long-lived systems, the number of architectural smells grows, creating unforeseen issues when engineers try to modify a system.

In such cases, engineers are eventually likely to realize the negative effects of the incurred technical debt and the need to refactor their system. However, they usually spot deeper architectural problems only when related implementation-level issues surface. For example, issue #1178 reported for Apache Pig indicates that developers recognize the problem of having a large number of functions in a component: "[The component] has been an area of numerous bugs, many of which have been difficult to fix" [2]. Similarly, issue #223 in Apache CXF acknowledges the need to refactor CXF's architecture to reduce the amount of code changes and improve extensibility [1].

Recent studies have established strong correlations between architectural smells and both (1) a system's proneness to change and (2) the emergence of certain implementation issues [35], [37]. Furthermore, many bugs reported for a system have been shown to have architectural roots [70], [44]. Prior work has also demonstrated that identifying *code* smells using existing approaches will not help to uncover the underlying architectural issues, and modifications to address thus identified problems run the risk of being inadequate, short-term patches [51]. Despite this, predictive models that leverage *architectural characteristics* to anticipate the implementation issues or the amount of change a system may experience have been scarce.

In this paper, we propose and empirically evaluate an approach to predict a system's (1) future implementation issues and (2) proneness to change based on the system's current and past architectural characteristics. Our work is inspired in part by the recent finding [35] that architectural smells and implementation issues are strongly correlated. Specifically, we analyze 466 versions of 10 open-source software systems. For each system version, we use 3 different methods to recover its architectures from source code. We analyze thus obtained 1,398 architectural models to detect 11 distinct types of architectural smells. The detected smells are subsequently used as features in our prediction models. We make use of different machine learning techniques to predict a given system's issue- and change-proneness based on the collected architectural features.

Our study has resulted in two principal findings regarding the predictive power of the models obtained in this manner:

1) The architectural smells detected in a system can help to accurately predict both the issue-proneness and change-

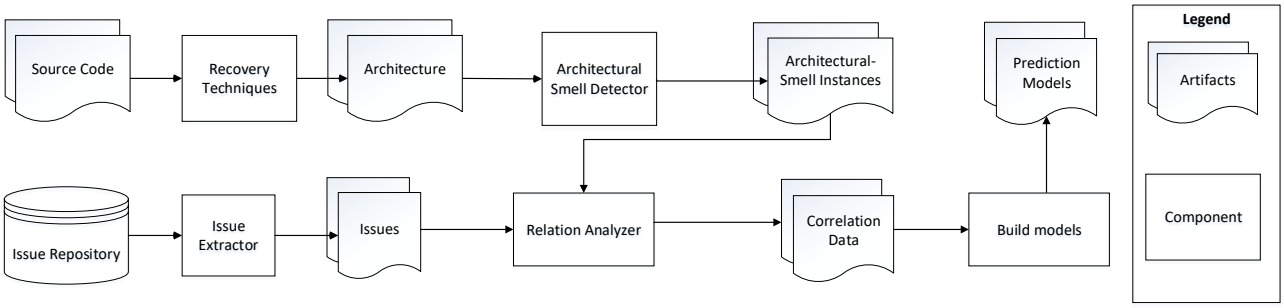


FIG. 1: ARCHITECTURE RECOVERY PIPELINE USED IN OUR STUDY AND ENABLED BY THE ARCADE TOOL SUITE.

proneness of that system at a given point in time. Our models yielded precision and recall scores of at least 70% (and as high as 95%) for specific recovered architectural views of the subject systems. This finding allows maintainers to foresee future problems involving new smell-impacted parts of a system.

2) Different, independently developed software systems tend to share issue- and change-proneness characteristics. This allows developers to use models created using data from a set of existing systems to predict the issue- and change-proneness of an unrelated system for which historical data does not exist (e.g., a newly developed system). While the accuracy of such general-purpose prediction models is lower than the system-specific models, the loss in accuracy is moderate, typically under 10%. Our results indicate that this is a fruitful area for further investigation, and that our models are already usable in practice for making certain types of decisions.

Section II introduces foundations for our study. Section III presents the research questions and describes the study. The results are detailed in Section IV. Threats to validity, related work, conclusions, and acknowledgment round out the paper.

II. FOUNDATION

Our work is directly enabled by three research threads: (1) software architecture recovery, (2) definition and analysis of architectural smells, and (3) tracking implementation issues. Figure 1 depicts how these threads are combined to answer our research questions in this paper.

A. Architecture Recovery with ARCADE

Garcia et al. [19] conducted a comparative evaluation of software architecture recovery techniques. Their objective was to measure the existing techniques' accuracy and scalability on a set of systems for which researchers had previously obtained "ground-truth" architectures [20]. To that end, the authors implemented a tool suite, named ARCADE, offering a large set of architecture recovery choices to an engineer.[1]

Garcia et al.'s results indicate that two techniques implemented in ARCADE consistently outperformed the rest: *ACDC* [66] and *ARC* [23]. We select these techniques for our study. *ACDC* leverages a system's *structural characteristics* to cluster implementation-level modules into architectural components, while *ARC* focuses on the *concerns* implemented by a system. *ACDC* relies on static dependency analysis; *ARC* uses information retrieval and machine learning.

PKG is another technique implemented in ARCADE. *PKG* extracts a system's implementation *package structure*. The package structure of a system is considered to be a reliable view of a system's "implementation architecture" [34]. We use it to complement the two selected clustering-based architectural views.

B. Architectural Smells

Architectural smells are instances of poor architectural design decisions [45]. They negatively impact system lifecycle properties, such as understandability, testability, extensibility, and reusability [21]. While code smells [16], anti-patterns [10], or structural design smells [17] originate from implementation constructs (e.g., classes, methods, variables), architectural smells stem from poor use of software architecture-level abstractions — components, connectors, interfaces, patterns, styles, etc. Detected instances of architectural smells are candidates for restructuring [9], to help prevent architectural decay and improve system quality.

Researchers have collected a growing catalog of architectural smells. Garcia et al. [21], [22] identified an initial set of four smells related to connectors, interfaces, and concerns. Mo et al. [46] introduced a new concern-related smell. Ganesh et al. [17] also summarized a catalog of structural design smells, some of which are at the architecture-level. Le et al. [35] described 11 different architectural smells and proposed a set of algorithms to detect them. Table I summarizes a consolidated list of smells that were identified in the above references, after removing duplicates and non-architectural smells.

C. Issue Tracking Systems

Issue tracking systems are commonly used development tools that allow users to report different problems and concerns about a system and monitor their status. All subject systems selected for analysis in this paper use Jira [4] as their issue tracking system. However, this is not a limitation; our approach can be applied to other issue trackers.

When reporting implementation issues, engineers categorize them into different types: *bug*, *new feature*, *improvement*, *task*

[1]The existing techniques implemented within ARCADE support structural clusterings of software systems' elements based on a range of criteria. While the resulting recovered models contain only partial architectural information for a given system, in this paper we will refer to them as "recovered architectures". We note that our use of this term is consistent with existing literature.

TABLE I: CONSOLIDATED CATALOG OF ARCHITECTURAL SMELLS

Category	Type	Definition	Consequences
Interface-based	Unused Interface	Component's interface is not linked to other components	Adds unnecessary complexity to the system
	Unused Brick	Component's interfaces are all unused	Same as Unused Interface, but more severe
	Sloppy Delegation	Component delegates functionality it could have performed	Reduces separation of concerns
	Functionality Overload	Component has an excessive amount of functionality	Reduced modularity
	Lego Syndrome	Component handles exceedingly small amount of functionality	High coupling
Change-based	Duplicate Functionality	Several components replicate the same functionality	Bugs if changing only one duplicate
	Logical Coupling	Parts of different components are frequently changed together	Similar to Duplicate Functionality
Dependency-based	Dependency Cycle	Set of components whose links form a circular chain	Changes to one component affect the entire cycle
	Link Overload	Component's interfaces have too many dependencies	Reduced isolation of changes
Concern-based	Scattered Parasitic Funct.	Multiple components responsible for realizing one concern	Changing a feature modifies multiple system parts
	Concern Overload	Component implements an excessive number of concerns	Violates separation of concerns

to be performed, etc. We consider all issue types in our study because they may result in relevant changes to a system. In other words, any issue type or individual issue instance may have an underlying architectural cause. Note that it would be possible to perform a finer-grained analysis using the same process we employed that would focus on a specific subset of issues or types.

Each issue has a status that indicates where the issue is in its lifecycle [3]. An issue starts as "open", progresses to "resolved", and finally to "closed". We restrict our study to closed and resolved issues that have been "fixed", and ignore those resolved issues that fall under "won't fix", "cannot reproduce", etc. We do so because any effects caused by the fixed issues presumably appear in certain system versions and disappear once the issue is addressed. Additionally, a fixed issue contains information that is useful for our study: (1) *affected versions* in which the issue has been found, (2) *type* of issue, and (3) *fixing commits*, i.e., the changes applied to the system to resolve the issue. Finding fixing commits is not always easy since there is no standard method for engineers to keep track of this information. Three ways of keeping track of an issue's fixing commits are commonly employed in our set of subject systems: (1) direct links to the commits, (2) specifying pull requests, and (3) specifying patch files. Our implemented tool supports collecting data from all three methods.

Based on the collected information, issues are mapped to detected smells. To do this, first, we find the system versions that the issue affects. Then we find the architectural smells present in those versions. We say the issue is infected by a given smell if and only if (1) both the issue and the smell affect the same system version and (2) the resolution of the issue changes files that are involved in the smell. Based on this relationship, we studied if the characteristics of an issue (e.g. issue type, number of fixing commits) depend on whether the issue is infected by a given smell.

Note that resolving an issue may not remove the smell that led to the issue in the first place. One reason is that developers could find a workaround. The smell may also correlate with more than one issue. In general, it is difficult to identify the exact relationship between a specific architectural smell instance and a specific implementation issue. Fortunately, we do not need to do that in our work, because we are looking for prediction models that uncover smell-issue correlations across most cases.

III. EMPIRICAL STUDY SETUP

This section describes our study setup. Our hypothesis and research questions are described in Section III-A. We then describe how we pre-processed the raw data in Section III-B.

A. Research Question and Research Hypothesis

Our ***hypothesis*** is that *it is possible to construct accurate models to predict the impact of architectural decay on a system's implementation*. To evaluate this hypothesis, we focus on the predictability of a system's issue- and change-proneness based on the identified architectural smells (i.e., the symptoms of decay). We define two research questions accordingly.

RQ1. *To what extent can the architectural smells detected in a system help to predict the issue-proneness and change-proneness of that system at a given point in time?*

The training data used to build the prediction models for a system is collected from different versions of that system. If these models can be shown to accurately predict issue- and change-proneness, this would indicate that architectural smells have consistent impacts on those two properties throughout a system's life span. In turn, this would confirm that the impact of architectural smells is not related to other factors, such as system size, which will change during a system's evolution. In addition, an accurate prediction model will be useful for maintainers to foresee the future issue- and change-proneness of newly smell-affected parts of a system, helping them to decide when and where they may need to refactor the system.

RQ2. *To what extent do unrelated software systems tend to share properties with respect to issue-proneness and change-proneness?*

This question investigates whether the issue- and change-proneness of a system can be accurately predicted by a general-purpose model trained using symptoms of architectural decay from unrelated systems. If such a model can be constructed, it can be reused by developers to predict properties of systems for which historical information is not (yet) available. An affirmative answer to this question would also have a deeper implication: software systems tend to share fundamental properties regardless of system type, application domain, developers, employed tools, programming languages, execution platforms, etc.

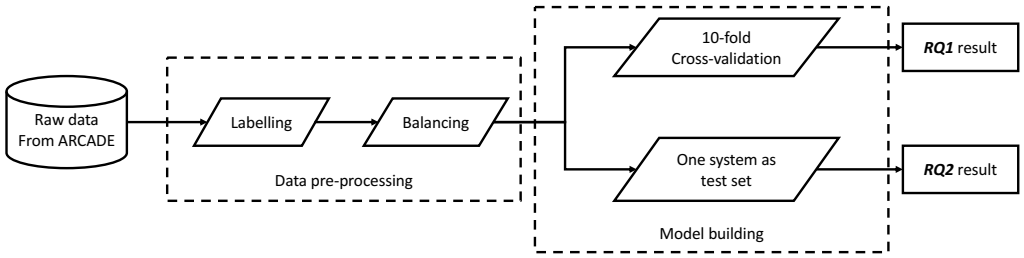


FIG. 2: DATA PROCESSING PIPELINE.

B. Building the Data Pipeline

To answer the two research questions, we build multiple prediction models based on different systems' architectural-smell data and assess the models' accuracy. We rely on ARCADE [36] to collect the underlying raw architectural-smell data, and WEKA[50]—a well-known ML framework—to pre-process the data, build prediction models, and evaluate their accuracy. The data pipeline we use is illustrated in Figure 2. Section III-B1 introduces the list of subject systems and the process of recovering their architectural artifacts with ARCADE. Two main pre-processing tasks are labeling and balancing the raw data, which are discussed in Sections III-B2 and III-B3, respectively. Creating the training and test sets, evaluating prediction models as well as determining the baseline models are discussed in Sections III-B4, III-B5, and III-B6, respectively.

1) ARCADE and Subject Systems: We collected data from ten open-source systems from the Apache Software Foundation, shown in Table II. Specifically, our study uses three types of data: (1) architectural smells detected in recovered architectures, (2) implementation issues collected from the Jira [4] issue repository, and (3) code commits extracted from GitHub [5].

Using ARCADE, we recover the subject systems' architectures using the three recovery techniques—ACDC, ARC, and PKG—whose accuracy and scalability have been demonstrated by prior work (recall Section II-A). We then analyze the recovered architectures for the presence of smells identified in the literature (recall Section II-B and Table I), as well as the systems' issue- and change-proneness. Those architectural artifacts are the raw data for building prediction models.

2) Labeling the Data: Data labeling is a key step to ensure the success of prediction models. In our prediction problem, we are interested in two properties—issue-proneness and change-proneness. These properties can be obtained by, first, counting the raw numbers of issues and changes in a system's development history and, then, finding a way of characterizing those numbers. Specifically, we assign nominal labels based on the raw numbers of issues and changes related to source files to represent different levels of issue- and change-proneness.

TABLE II: SUBJECT SYSTEMS IN OUR STUDY

System	Domain	# Versions	# Issues	Avg. LOC
Camel	Integration F-work	78	9665	1.13M
CXF	Service F-work	120	6371	915K
Hadoop	Data Proc. F-work	63	9381	1.96M
Ignite	In-memory F-work	17	3410	1.40M
Nutch	Web Crawler	21	1928	118K
OpenJPA	Java Persist.	20	1937	511K
Pig	Data Analysis F-work	16	3465	358K
Struts2	Web App F-work	36	4207	379K
Wicket	Web App F-work	72	6098	332K
ZooKeeper	Config. Mgmt F-work	23	1390	144K

Converting a set of numerical values to nominal labels depends on the values' distribution. In our problem, the numbers of issues and changes follow a heavy-tailed distribution [15], where many files are associated with small numbers of issues and code changes, while comparatively fewer files are associated with large numbers of issues and changes. This is not an uncommon type of distribution [72], [8]. As an illustration, the Pareto chart [68] in Figure 3 depicts the distribution of issues per file in Hadoop: while few files are associated with a large number of issues, the arc, which represents the cumulative percentage of file-groups' sizes, shows a clear heavy-tailed pattern.

One common labeling approach is to segment a heavy-tailed distribution into head and tail segments. A more sophisticated approach is to divide the distribution into three parts—head, body, and tail—which in our case represent the three levels of proneness: low, medium, and high. We choose the latter approach because the numerical values in our study span a wide range. Having these three levels gives developers a better estimation of architectural decay's impact.

To segment a dataset, we use the Pareto principle [52], a popular segmentation method for heavy-tailed distributions, widely used in software engineering (e.g., [8], [30], [61]). To obtain the three segments, we apply the Pareto principle twice, as suggested in literature [6]. Specifically, we divide the original dataset into two portions. The first portion contains 80% of the

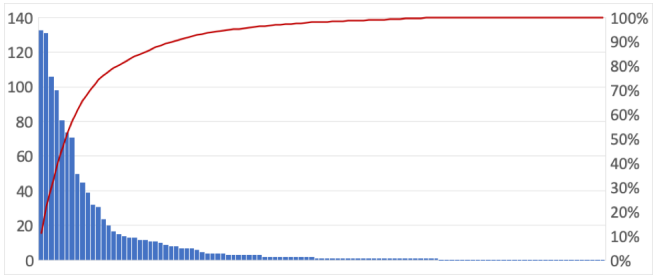


FIG. 3: PARETO CHART OF ISSUES PER FILE IN HADOOP. THE X-AXIS REPRESENTS THE HADOOP FILES GROUPED BY THE NUMBER OF ISSUES THEY CONTAIN, THE LEFT Y-AXIS THE NUMBER OF FILES IN SAME GROUPS, AND THE RIGHT Y-AXIS THE CUMULATIVE PERCENTAGE OF GROUPS' SIZES.

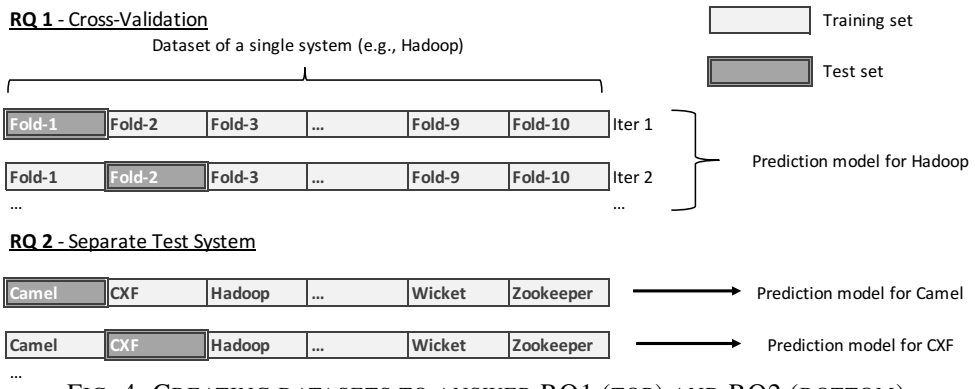


FIG. 4: CREATING DATASETS TO ANSWER RQ1 (TOP) AND RQ2 (BOTTOM).

original dataset's low-end, while the second portion contains 20% of the high-end. We apply the Pareto segmentation once more to the latter portion, thus obtaining two new portions that respectively contain the next 16% (80% of the 20%) and 4% (20% of the 20%) of the high-end data points.

In order to collect the data regarding architectural decay, for each version of a subject system, we first collect the list of "fixed" issues affecting that version. Next, we collect the files that were changed when fixing the issues. For each file, we gather its associated architectural smells, the number of issues whose fixing commits changed that file (used when determining the system's *issue-proneness* in Sections IV-RQ1-A and IV-RQ2-A), and the total number of changes (used when determining the system's *change-proneness* in Sections IV-RQ1-B and IV-RQ2-B). After the raw data is collected, we label it using the Pareto technique mentioned above before feeding it to supervised ML algorithms.

To determine the level of issue-proneness of a source file in a system version, first, the number of issues related to that file is collected. This is one data point. We collect data points for all files in all available versions of a system, and then sort the dataset by the numbers of issues, from low to high. Then, the first 80% of data points are marked with "low" labels; the next 16% and 4%, respectively, are marked with "med(ium)" and "high" labels. To determine the change-proneness of a source file in a system's version, we count the number of commits related to that file and repeat a similar labeling process.

Table III shows several data samples in our datasets after labeling. The shown features, i.e., architectural smells in our case, are CO (Concern Overload), SF (Scattered parasitic Functionality), LO (Link Overload), and DC (Dependency Cycle). The output features, i.e., labels, are the levels of issue-proneness and change-proneness. The two leftmost columns show the versions and filenames of each data point. The next eleven columns are binary features that indicate the presence (1) or absence (0) of a specific smell (recall Table I) in a given file.

TABLE III: DATA SAMPLES FROM HADOOP

Vers.	Filename	CO	SF	LO	DC	...	Iss	Chg
0.20.0	dfs/DFSClient.java	0	1	1	1	...	H	L
0.20.0	mapred/JobTracker.java	1	0	1	0	...	M	M
0.20.0	tools/Logalyzer.java	0	0	0	0	...	L	L
...	...	...	...	...	...	...	...	...

The two rightmost columns indicate the issue-proneness ("Iss") and change-proneness ("Chg") of the files. For example, in version 0.20.0 of Hadoop, `DFSClient.java` has three smells: SPF, LO, and DC. The file's issue-proneness is high (H), and its change-proneness is low (L). On the other hand, both issue- and change-proneness of `JobTracker.java` are medium (M).

3) Balancing the Data: Due to the distribution of data and the labeling approach, we need to balance our datasets [55]. Recall from Section III-B2 that the $low : med : high$ ratio of our datasets is 80:16:4 (i.e., 20:4:1). If such a dataset were used to train a prediction model, the most likely outcome would be a model that predicts "low" for every data point. As we are more interested in "high" and "med" labels, such a model would be useless. It is thus important to ensure that weighted metrics are not biased by less (or more) frequent labels.

We use SMOTE [11] to balance our dataset, oversampling "med" by a factor of 5 and "high" by a factor of 20. SMOTE is a technique that synthesizes new minority samples based on nearest neighbors between sample data points. Adding new minority samples guarantees that the dataset will be balanced, i.e., that the $low : med : high$ ratio will be 1:1:1.

4) Training and Test Sets: To build and test our prediction models, we use two different approaches for the two research questions, as illustrated in Figure 4. In the first approach, used for RQ1, one dataset is created for each subject system with a cross-validation setup. Specifically, we use 10-fold cross-validation, where the dataset is randomly divided into ten equal-sized subsets. Then, we sequentially select one subset and test it against the prediction model built by the other nine subsets. The final result is the mean of the ten tests' results. In the second approach, used for RQ2, we combine all subject systems and then divide them into two independent datasets: a training set, which comprises nine systems, and a test set, which comprises the single remaining system.

5) Evaluation Metrics: To evaluate the accuracy of our models, we use precision and recall [54]. Precision is the fraction of correctly predicted labels over all predicted labels. Recall is the fraction of correctly predicted labels over all actual labels.

For illustration, consider the sample confusion matrix, shown in Table IV, that is produced after classifying 25 samples into "high", "med", and "low". The precision for the "high" label is

TABLE IV: EXAMPLE PREDICTED VS. ACTUAL VALUES

		True/Actual		
		High	Med	Low
Predict	High	4	6	3
	Med	1	2	0
	Low	1	2	6

the number of correctly predicted "high" samples (4) out of all samples predicted to be "high" (4+3+6=13), i.e., 30.8%; its recall is the number of correctly predicted "high" samples (4) out of the number of actual "high" samples (4+1+1=6), i.e., 66.7%. We can similarly calculate the precision and recall for "med" and "low". Finally, we compute the average values of all labels.

If a model predicts the correct labels, we consider this a true positive. On the other hand, if the model predicts any of the three labels ("high", "med", or "low") incorrectly, we consider this a false positive. This is the standard way of measuring the accuracy of multi-label problems [65].

6) Determining Baseline Models: To determine the effectiveness of the prediction models, we need to compare them to a baseline. In this case, we consider a baseline model to be the simplest possible prediction. The model can be obtained through different approaches. For some problems this may be a random result, and for others in may be the most common prediction. As our dataset has been balanced (Section III-B3), the simplest approach is "uniform" — generate predictions uniformly at random. This implies a prediction in which Table IV has equal values in all cells, giving us a model with both precision and recall of 33.3%.

IV. EMPIRICAL STUDY RESULTS

In this section, for each of the two research questions we discuss the validation method and the associated findings.

RQ1: *To what extent can the architectural smells detected in a system help to predict the issue-proneness and change-proneness of that system at a given point in time?*

In this prediction problem, all input features are binary (recall Table III), indicating whether a file contains an architectural smell. For this reason, decision-based techniques are most likely to yield good results [41]. Metrics collected from a range of models we built and evaluated using four different classification techniques—decision table [31], decision tree [56], logistic regression [39], naive bayes [28]—confirmed this. We thus only discuss the results obtained by the decision-table models.

A. Issue-Proneness

Recall from Section III-B that, to compute issue-proneness, for each file in each version of a given system, we gather the file's associated architectural smells and number of issues whose fixing commits changed the file. Table V shows the precision and recall of the models for predicting the issue-proneness of our subject systems from Table II. These metrics are computed using 10-fold cross-validation [32]. The bottom-most row shows the average values across all systems. For each system, we built different prediction models based on smells detected in the three architectural views (ACDC, ARC, and PKG). In total, 30 prediction models per system were created and evaluated.

TABLE V: PREDICTING ISSUE-PRONENESS

	ACDC		ARC		PKG	
System	Precision	Recall	Precision	Recall	Precision	Recall
Camel	69.9%	68.4%	70.8%	67.0%	68.2%	62.8%
CXF	78.0%	76.7%	68.9%	68.3%	64.7%	63.8%
Hadoop	81.2%	80.1%	76.6%	76.6%	72.8%	73.4%
Ignite	78.9%	78.1%	78.9%	79.1%	70.4%	71.0%
Nutch	80.8%	71.6%	82.5%	82.7%	68.3%	52.1%
OpenJPA	71.4%	68.3%	74.5%	73.2%	69.2%	67.9%
Pig	71.7%	69.1%	71.3%	71.1%	68.6%	69.5%
Struts2	89.2%	89.0%	95.0%	94.8%	79.1%	78.3%
Wicket	69.2%	70.1%	76.7%	77.1%	63.7%	65.4%
ZooKeeper	72.0%	72.6%	70.8%	69.2%	68.7%	69.4%
Average	76.2%	74.4%	76.6%	75.9%	69.4%	67.4%

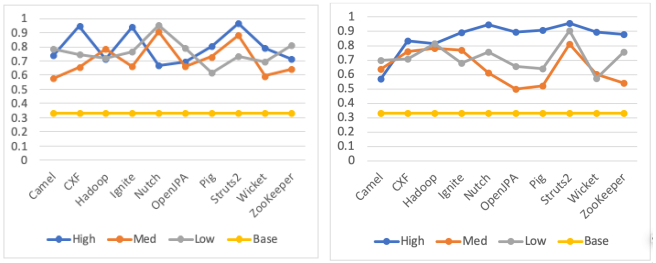
In general, the prediction models that relied on architectures recovered by ACDC and ARC were comparable in terms of accuracy: the average (precision, recall) for the ACDC and ARC models were (76.2%, 74.4%) and (76.6%, 75.9%) respectively. On the other hand, the models emerging from PKG yielded accuracy that was up to 13% lower. The models yielded very high predictive power in the cases of certain systems. For example, the ARC-based models for Struts2 achieved ≈95.0% and the ACDC-based models ≈90.0% for each of the two metrics.

As discussed in Section III-B3, our dataset has been balanced to ensure that the trained models will accurately predict "high" and "med" labels, in which we are interested. Table VI shows the precision and recall of issue prediction for all three labels. While there are variations across the three labels, the average precision and recall for the "high" label—79.6% and 85.9%, respectively—outstrip the average values for the other two labels. Figure 5 shows the comparison of our prediction models with the baseline model. Our prediction models are at least 1.5× better (2× in a majority of cases) than the baseline's 33.3%, further confirming that our models are useful for predicting files with high numbers of related issues.

Our results confirm that architectural smell-based models can accurately predict the issue-proneness of a system. In other words, architectural smells have a consistent impact on a system's implementation with respect to issue-proneness over the system's lifetime. This finding means that *architectural decay can be a powerful indicator of the health of a system's implementation*. It serves as a direct motivator for software engineers to pay more attention to the architecture, and architectural smells, in their systems. For example, system maintainers can use our models to foresee future problems, to devise refactoring plans, to prioritize their activities, etc.

TABLE VI: PREDICTING ISSUE-PRONENESS WITH "HIGH", "MED" AND "LOW" LABELS UNDER ACDC

	High		Med		Low	
System	Precision	Recall	Precision	Recall	Precision	Recall
Camel	73.9%	56.9%	57.6%	63.6%	78.2%	69.9%
CXF	94.4%	83.3%	65.2%	76.0%	74.5%	70.7%
Hadoop	71.2%	81.5%	78.3%	78.1%	72.1%	81.5%
Ignite	93.8%	89.1%	66.4%	76.8%	76.6%	67.8%
Nutch	66.9%	94.7%	90.4%	61.2%	95.0%	75.8%
OpenJPA	69.3%	89.5%	65.9%	49.8%	79.1%	65.6%
Pig	80.5%	90.9%	72.9%	52.3%	61.8%	64.1%
Struts2	96.3%	95.7%	88.2%	81.1%	83.1%	90.4%
Wicket	78.8%	89.6%	59.3%	60.3%	69.5%	57.3%
ZooKeeper	71.0%	88.0%	64.2%	54.2%	80.7%	75.6%
Average	79.6%	85.9%	70.5%	65.3%	76.1%	71.9%



(a) Precision (b) Recall

FIG. 5: PRECISION AND RECALL OF ISSUE-PRONENESS PREDICTION FOR EACH LABEL IN ACDC.

The comparatively poorer performance of PKG in answering RQ1 suggests that implementation-package structure is not effective for measuring architectural decay and it can mask deeper architectural problems. This observation is in line with previous findings [36], which showed that, compared to ACDC and ARC, PKG is markedly less useful for understanding the underlying architectural changes and their impact.

This leads to another observation. Recall the categorization of architectural smells in Section II-B and Table I: two of the four categories are dependency-based and concern-based smells. This suggests that ACDC (dependency-based recovery) and ARC (concern-based recovery) should inherently outperform PKG when such smells are encountered. It further suggests that targeting specific recovery techniques to specific types of smells, and then finding a way to combine their results, may yield even higher accuracy in our prediction models. We are exploring this hypothesis in our ongoing work.

B. Change-Proneness

Recall from Section III-B that, to compute change-proneness, for each file in each version of a given system we gather (1) the file's associated architectural smells and (2) the total number of changes to the file reflected in the implementation issues' fixing commits. We used the same approach to evaluate the accuracy of the 30 architectural models for each system in predicting change-proneness as we did for predicting issue-proneness.

Table VII shows the accuracy of our models. The models based on PKG-recovered architectures again have the lowest accuracy. In some systems, e.g., CXF and Nutch, the values for PKG-recovered architectures are 10-20% lower than the corresponding values in the other two views. The average (precision, recall) are (74.7%, 71.6%) and (73.6%, 73.5%) for the ACDC- and ARC-based architectural views, respectively. Notably, the values yielded when analyzing Struts2 are, once again, very high. A further investigation of Struts2's dataset highlighted a distinguishing characteristic: 36 of the analyzed versions are distributed across just four minor Struts2 versions: 2.0.x, 2.1.x, 2.2.x and 2.3.x. In other words, the changes in most of these 36 versions were "patches". It is reasonable to expect that the architectures and detected smell instances between patches within a single minor version will be very similar. The prediction model for Struts2 benefits from this similarity and thus achieves very high accuracy in the cross-validation test. This suggests a promising strategy for building

TABLE VII: PREDICTING CHANGE-PRONENESS

	ACDC		ARC		PKG	
System	Precision	Recall	Precision	Recall	Precision	Recall
Camel	69.9%	63.4%	68.0%	67.1%	60.3%	61.0%
CXF	73.7%	70.8%	69.7%	63.4%	60.8%	63.4%
Hadoop	78.1%	73.2%	74.9%	74.8%	67.4%	70.0%
Ignite	77.5%	76.1%	75.8%	76.1%	68.7%	69.1%
Nutch	73.1%	66.8%	76.3%	78.0%	62.2%	46.1%
OpenJPA	78.3%	77.7%	74.3%	70.0%	68.2%	62.1%
Pig	70.1%	67.4%	69.6%	70.2%	65.9%	66.5%
Struts2	89.3%	85.8%	87.8%	96.7%	71.2%	73.7%
Wicket	66.6%	65.3%	72.1%	71.8%	62.7%	59.0%
ZooKeeper	69.9%	69.6%	67.8%	67.2%	65.5%	64.4%
Average	74.7%	71.6%	73.6%	73.5%	65.3%	63.5%

prediction models: to increase the accuracy of models used to predict properties of a system version, one should *select recent versions* instead of all versions across the entire system lifespan.

In summary, our results confirm that the historical data of a software system regarding its architectural smells, issues, and changes can be used to develop models to accurately predict the issue- and change-proneness of that system. The results also indicate that architectural smells have a consistent impact on software system implementations throughout the systems' lifetimes. Our architecture-based prediction approach, whose performance is usually two times better than the baseline, is useful for software maintainers to foresee likely future problems in newly smell-impacted parts of their system. The approach can also help in creating maintenance plans that can help to effectively reduce the system's issue- and change-proneness. Lastly, ACDC and ARC outperform PKG, emphasizing the importance of selecting the appropriate architecture recovery techniques and targeting them to the task at hand.

RQ2: *To what extent do unrelated software systems tend to share properties with respect to issue- and change-proneness?*

The results obtained in answering RQ1 showed that architectural smells consistently impact the issue- and change-proneness of a software system during its lifetime. In that sense, RQ2 can be considered an extension of RQ1: we aim to understand whether architectural smells have consistent impacts across *unrelated* software systems, more specifically, whether the issue- and change-proneness of a system can be accurately predicted by models trained with data from unrelated systems. More deeply, this research question tries to assess whether there are fundamentally shared traits across software systems, regardless of their developers and development processes, implementation features, application domains, underlying designs, etc.

To answer this question, instead of using 10-fold cross-validation, we selected each subject system as the test system and used its dataset as the test set; the training set was then created by combining datasets of the remaining nine systems. For reference, we also built a prediction model by combining all ten systems, i.e., including the test systems. Note that the datasets of different subject systems have different sizes; we had to resample those datasets to the same size before combining them.

A. Issue-Proneness

Tables VIII, IX, and X summarize the precision and recall values of *RQ2* experiments with regard to predicting issue-

TABLE VIII: PREDICTING ISSUE-PRONENESS – PRECISION (TOP) AND RECALL (BOTTOM) UNDER ACDC

System	10-fold (RQ1)	All 10	9 Others
Camel	69.9%	64.8%	53.6%
CXF	78.0%	71.4%	66.4%
Hadoop	81.2%	71.1%	62.8%
Ignite	78.9%	73.9%	60.2%
Nutch	80.8%	74.9%	59.6%
OpenJPA	71.4%	68.8%	63.9%
Pig	71.7%	66.8%	61.4%
Struts2	89.2%	77.1%	69.1%
Wicket	69.2%	66.7%	55.0%
ZooKeeper	72.0%	65.4%	56.0%
Camel	68.4%	57.5%	46.7%
CXF	76.7%	71.3%	65.7%
Hadoop	80.1%	69.2%	62.9%
Ignite	78.1%	73.5%	59.3%
Nutch	71.6%	68.8%	54.4%
OpenJPA	68.3%	63.0%	57.3%
Pig	69.1%	64.1%	58.8%
Struts2	89.0%	76.4%	68.8%
Wicket	70.1%	66.0%	54.9%
ZooKeeper	72.6%	60.3%	56.9%

TABLE IX: PREDICTING ISSUE-PRONENESS – PRECISION (TOP) AND RECALL (BOTTOM) UNDER ARC

System	10-fold (RQ1)	All 10	9 Others
Camel	70.8%	64.9%	59.7%
CXF	68.9%	55.2%	49.0%
Hadoop	76.6%	67.6%	59.6%
Ignite	78.9%	66.9%	62.3%
Nutch	82.5%	64.6%	62.3%
OpenJPA	74.5%	66.9%	63.9%
Pig	71.3%	62.1%	61.7%
Struts2	95.0%	76.1%	63.8%
Wicket	76.7%	63.3%	62.0%
ZooKeeper	70.8%	66.3%	50.4%
Camel	67.0%	59.4%	48.5%
CXF	68.3%	62.3%	54.5%
Hadoop	76.6%	67.4%	59.4%
Ignite	79.1%	66.5%	61.6%
Nutch	82.7%	58.1%	53.9%
OpenJPA	73.2%	65.5%	62.0%
Pig	71.1%	62.5%	61.1%
Struts2	94.8%	75.7%	63.7%
Wicket	77.1%	65.3%	63.6%
ZooKeeper	69.2%	67.1%	56.4%

proneness under ACDC, ARC, and PKG, respectively. The left-most columns of these tables show the lists of systems. The precision and recall values are presented for three different cases:

1) "10-fold" column – 10-fold cross-validation on the test set. We reproduce this result from RQ1 for easy reference.
2) "All 10" column – Models trained by datasets from all 10 systems, including the test set.
3) "9 Others" column – Models trained by 9 other systems' datasets, not including the test set.

In total, beside the 300 issue-proneness prediction models per system that emerged from RQ1's analysis, we built and evaluated 60 additional issue-proneness models to answer RQ2.

We found several consistent trends across all three architectural views. First, a prediction model built by combining data sets of multiple different software systems, even if the test system itself is included, has lower accuracy than the model

TABLE X: PREDICTING ISSUE-PRONENESS – PRECISION (TOP) AND RECALL (BOTTOM) UNDER PKG

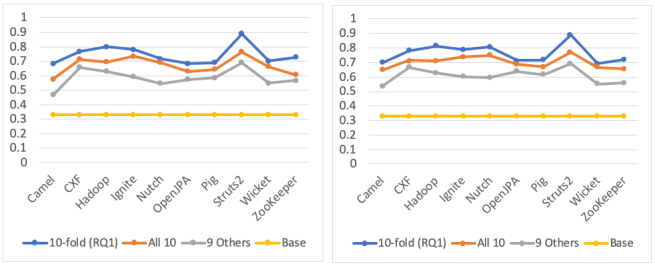
System	10-fold (RQ1)	All 10	9 Others
Camel	68.2%	59.5%	46.0%
CXF	64.7%	62.7%	59.1%
Hadoop	72.8%	61.8%	50.2%
Ignite	70.4%	70.2%	62.6%
Nutch	68.3%	66.9%	51.9%
OpenJPA	69.2%	71.2%	53.1%
Pig	68.6%	68.0%	53.6%
Struts2	79.1%	92.4%	67.6%
Wicket	63.7%	66.1%	60.2%
ZooKeeper	68.7%	66.3%	44.0%
Camel	62.8%	50.9%	43.5%
CXF	63.8%	60.0%	44.7%
Hadoop	73.4%	61.5%	50.3%
Ignite	71.0%	69.5%	62.3%
Nutch	62.1%	54.1%	50.9%
OpenJPA	67.9%	68.3%	39.2%
Pig	69.5%	68.0%	44.5%
Struts2	78.3%	92.0%	67.1%
Wicket	65.4%	66.1%	58.9%
ZooKeeper	69.4%	66.8%	42.7%

built for that specific test system. This can be seen in all three Tables VIII, IX, and X, where the "All 10" columns have lower values for precision and recall than the corresponding "10-fold" (results from RQ1) columns.

More interesting is the case where the test system is excluded and the model is trained on the datasets from the remaining nine systems (the "9 others" column). This represents the scenario of using a generic predictive model comprising entirely different systems. The precision and recall values predictably decrease further across all three architectural views. These results are reflective of the intuition that using datasets from different systems can create a more general-purpose model, but is also likely to add noise and reduce the model's ability to predict the properties of a specific system. Therefore, if a sufficiently large dataset for a given system is available, the system's prediction models should be trained only on that dataset.

At the same time, it is interesting to note that the loss of accuracy between the "10-fold" and "9 Others" models is relatively moderate: with few exceptions, it is on the order of 10-20%. On the lower end, one example exception is PKG's precision for Wicket's issue-proneness (Table X-top), where the discrepancy is only 3.5%. On the higher end, an interesting exception are the precision and recall values obtained by ARC for Struts2 (Table IX), which are both more than 30% lower for the "9 Others" models. This ties to the above discussion of the limited types of smells that exist in Struts2: its uniqueness decreased the ability of other systems to predict its issue-proneness, just like it helped ensure highly accurate models when using only its own historical data.

Figure 6 shows a comparison of precision and recall between different combinations of ACDC based models. We observe that using data from "9 Others" systems can yield a relatively good prediction model with at least 50% improvement compared to the baseline (0.5 vs. 0.33). In addition, the accuracy of "All 10" models lends support to a hypothesis that if a system has a short history of development, then including generic data



(a) Precision (b) Recall

FIG. 6: PREDICTING ISSUE-PRONENESS UNDER ACDC.

can help improve predictive performance. We are currently evaluating this hypothesis more extensively.

B. Change-Proneness

We observed analogous trends to those discussed above in the experiments that attempt to predict the change-proneness using unrelated systems' datasets. We elide this data for space.

In summary, the results of the experiments conducted in the context of *RQ2* confirm that software systems tend to share properties with respect to issue- and change-proneness. The accuracy of general-purpose models is lower than that of specific models, but the gap is not prohibitive. Our results suggest that developers can use general-purpose models to get an overall sense of the likely issue- and change-proneness of a new software system in the early stages of its development, before sufficiently large numbers of system versions become available. Similarly, developers can use such models to predict important properties of any existing systems for which historical data is missing, spotty, or unreliable.

An interesting question is whether restricting general-purpose models to systems that are likely to share certain key characteristics can improve the models' predictive power. This is something we have not done in our current study: while the set of test systems we used share some characteristics (e.g., Java-based enterprise systems and Apache Projects), they are also inherently different systems targeting a variety of domains. Our ongoing work is investigating whether taking into account factors such as the role of the employed development processes, off-the-shelf frameworks, system design principles and patterns, application domains, etc. can be used to increase the accuracy of the general-purpose models.

Overall, the predictive models we developed provide developers another tool to check and maintain their software system's health and track technical debt. A straightforward way to identify "unhealthy" parts of a system is to look for *long-lived smelly files*, i.e., files that have been involved in architectural smells across a large number of system versions. These files have a high potential to introduce new issues. Figure 7 shows examples of such files from Hadoop and Struts2. The x-axes in both plots indicate system versions, while the the y-axes indicate the numbers of smells in which each of the files is involved.

From the collected data such as that depicted in Figure 7, we have observed that long-lived smelly files are repeatedly involved in new issues during a system's lifetime. For example, `DFSClient.java` is mentioned in ≈2,900 Hadoop issues to

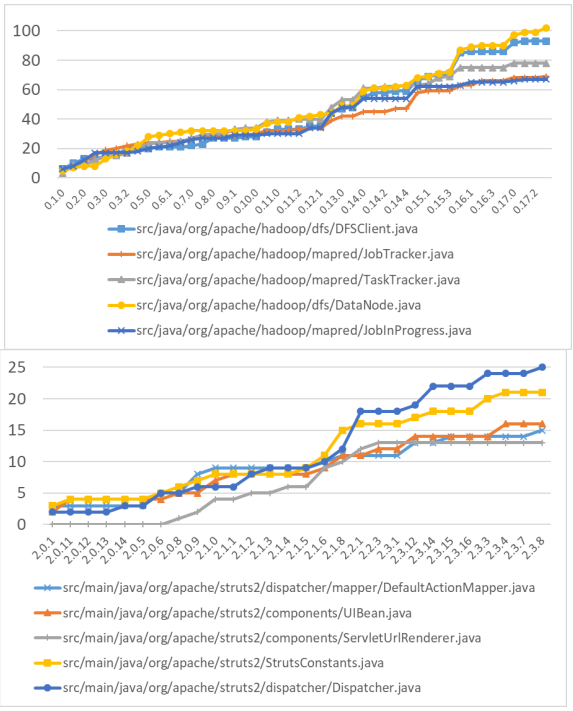


FIG. 7: TOP-5 LONG-LIVED SMELLY FILES IN HADOOP (TOP) AND STRUTS2 (BOTTOM).

date; `JobTrackers.java` is mentioned in ≈2,200 Hadoop issues; `Dispatcher.java` is mentioned in ≈670 Struts2 issues; and so on. We posit that stemming such trends and properly addressing the underlying problems will require considering the architectural causes of these issues.

V. THREATS TO VALIDITY

The key threats to **external validity** include our subject systems. Most of the steps in our data gathering process are automated. However, manual intervention is required since each system has different implementation conventions. Due to the manually-intensive data gathering process, we have used data from ten subject systems in our dataset. We mitigate a possible threat stemming from the number of systems by using data from their 466 versions and evaluating 720 prediction models.

All our subject systems are Apache projects, implemented in Java, and use the Jira issue tracking system. The reason for this is that it helped to simplify our data gathering and analysis workflow. In our on-going work, we are expanding our analysis beyond Apache. The diversity of the chosen systems, however, helps to reduce this threat, as does the wide adoption of Apache software, Java, and Jira. Further, all the recovery techniques and smell definitions in this paper are language-independent.

Our study's **construct validity** is threatened by (1) the accuracy of the recovered architectural views, (2) the detection of architectural smells, and (3) the relevance of implementation issues. To mitigate the first threat, we applied three architecture recovery techniques (ACDC, ARC, and PKG) that had previously exhibited the greatest usefulness in an extensive comparative analysis of available techniques [19] and in a study of architectural change during system evolution [36], [7],

[62]. The three techniques were developed independently and use different strategies for recovering a system's architecture. To mitigate the second threat, we selected architectural smell types that were previously studied on a smaller scale [42], [46], [38], [22], [21], and were shown to be strong indicators of architectural problems. Finally, to mitigate the third threat, we only collected "resolved" and "closed" issues, i.e., those issues that have been independently verified and fixed by developers.

The primary threat to our study's **internal validity** and **conclusion validity** involves the predictability relationship between reported implementation issues and architectural smells. Our prediction models are built based on significant correlations between architectural smells and implementation issues, which have been confirmed in other work [35]. Although correlation does not imply causality, we have shown examples of the causal relationship's existence. Prior work has also confirmed the causality between implementation issues and architectural smells via manual inspection [70], [44]. In addition, our observations are consistent across the ten systems.

VI. Related Work

Predicting implementation issues and code change have been widely studied research problems in software maintenance. The main type of implementation issues that researchers were interested early on were defects. Li et al. [40] used OO metrics as predictors of software maintenance effort. Subramanyam et al. [63] also demonstrated that a set of metrics [12] has significant implications on software defects. Nagappan et al. [49] found a representative set of code complexity measures to determine failure-prone software entities. However, the metrics considered in prior work cannot prevent defects at higher abstraction levels, such as architectural problems.

Issue prediction based on bug-fixing history is also an established area. Rahman et al. [58] developed an algorithm that ranks files by their numbers of past changes. The algorithm helps developers find hot spots in the system that need developers' attention. There are more sophisticated methods that combine historical information and software change impact analysis to increase the efficiency and accuracy of the prediction [67], [27], [57]. However, as before, these approaches do not explain higher-level defects caused by architectural decay.

Code changes have a close connection with defects in software. Nagappan et al. [48] used code churn to predict the defect density of software systems. Hassan et al. [26] used complexity metrics based on code changes to predict faults. Code change has been used in a number of other research efforts [71], [14], [37], [36] to evaluate system maintainability.

To predict code changes, Romano et al. proposed two approaches, relying on code metrics [59] and anti-patterns [60]. Xia et al.'s approach [69] predicts a system's change-proneness using co-change information of unrelated systems. While their approach is similar to the one we employed in the context of RQ2, it yields relatively low accuracy. Malhotra et al. [43] used hybridized techniques to identify change-prone classes. However, their empirical study is relatively small. Kouroshfar et al. [33] do use architectural information to study the correlation between co-changes across architectural models and defects. However, they restrict their study to cross-module changes.

VII. Conclusion

This paper's contributions are twofold. First, we have developed an approach that can identify parts of a software system that are likely targets of future maintenance activities based on architectural characteristics as well as the change- and issue-proneness of different architectural elements. Second, we have conducted an empirical study that highlights the impact of architectural decay on ten well known open-source systems.

We leverage the identified correlations between symptoms of architectural decay and reported implementation issues to develop an architecture-based approach that accurately predicts a system's issue- and change-proneness. Our approach has been validated on ten existing systems, considering 11 different types of smells under three different architectural views. This is the first study of its kind and, as such, its results can be treated as a foundation on which subsequent work should build. At the same time, the study has resulted in several important findings regarding the predictive power of architecture-based models.

Our study confirmed that architectural smells consistently impact a system's implementation during the system's lifecycle. In other words, the impact does not change significantly with other factors such as system size. This means that the detected architectural smells can help to accurately predict the issue-proneness and change-proneness of a system at any relevant point in time. In turn, such architecture-based prediction can serve as a useful tool for maintainers to recognize future problems associated with newly smell-impacted parts of the system and to plan their activities.

As a perhaps more unexpected result, we have shown that unrelated software systems tend to share properties with respect to issue- and change-proneness. This allows developers to use general-purpose models created with the available data from a set of existing systems to predict the properties of systems for which such information is missing. Unsurprisingly, the accuracy of such general-purpose models is lower than that of system-specific models, but not prohibitively so. Our results suggest that it is possible to develop such models sufficiently accurately to use them as a basis of actionable advice.

It is important to keep in mind that this was an initial attempt at constructing general-purpose prediction models. Our models were trained using all architectural smells and software systems without particular prior planning. Our future work will investigate how to select an appropriate set of systems to improve the accuracy of these models. We will also explore whether further accuracy improvements can be achieved by restricting the types of architectural smells on which the models are trained.

VIII. Acknowledgments

This work is supported by the U.S. National Science Foundation under grants 1717963, 1823354, and 1823262 and U.S. Office of Naval Research under grant N00014-17-1-2896.

REFERENCES

[1] CXF Implementation Issue - CXF-223. https://issues.apache.org/jira/browse/CXF-223, 2007.

[2] Pig Implementation Issue - PIG-1178. https://issues.apache.org/jira/browse/PIG-1178, 2010.

[3] What is an issue. https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html, 2018.

[4] Apache jira. https://issues.apache.org/jira, 2019.

[5] GitHub. https://github.com/, 2019.

[6] J. Arthur. *Six Sigma simplified: quantum improvement made easy*. KnowWare International, 2001.

[7] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, 2016.

[8] B. W. Boehm. Value-based software engineering: Overview and agenda. In *Value-based software engineering*, pages 3–14. Springer, 2006.

[9] I. Bowman, R. Holt, and N. Brewster. Linux as a case study: its extracted software architecture. In *ICSE*, 1999.

[10] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John wiley & sons, 2007.

[11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[13] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[14] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software evolution*, pages 37–67. Springer, 2008.

[15] S. Foss, D. Korshunov, S. Zachary, et al. *An introduction to heavy-tailed and subexponential distributions*, volume 6. Springer, 2011.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[17] S. Ganesh, T. Sharma, and G. Suryanarayana. Towards a principle-based classification of structural design smells. *Journal of Object Technology*, 12(2):1–1, 2013.

[18] J. Garcia. *A Unified Framework for Studying Architectural Decay of Software Systems*. PhD thesis, University of Southern California, 2014.

[19] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496, 2013.

[20] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. *ICSE*, 2013.

[21] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, 2009.

[22] J. Garcia, D. Popescu, G. Edwards, and M. Nenad. Identifying Architectural Bad Smells. In *13th European Conference on Software Maintenance and Reengineering*, 2009.

[23] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *ASE*, 2011.

[24] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.

[25] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):33:1–33:39, Sept. 2014.

[26] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[27] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 200–210. IEEE, 2012.

[28] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[29] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[30] A. R. Kiremire. The application of the pareto principle in software engineering. *Consulted January*, 13:2016, 2011.

[31] R. Kohavi. The power of decision tables. In *European conference on machine learning*, pages 174–189. Springer, 1995.

[32] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[33] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai. A study on the role of software architecture in the evolution and quality of software. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 246–257. IEEE Press, 2015.

[34] P. B. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 1995.

[35] D. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *ICSA*, 2018.

[36] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Proc. Mining Software Repositories, 2015*.

[37] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. Relating architectural decay and sustainability of software systems. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016.

[38] D. M. Le and N. Medvidovic. Architectural-based speculative analysis to predict bugs in a software system. In *Proceeding ICSE '16 Proceedings of the 38th International Conference on Software Engineering*, pages 807–810. ACM New York, NY, USA ©2016, 2016.

[39] S. Le Cessie and J. C. Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

[40] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993. Object-Oriented Software.

[41] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine learning*, 40(3):203–228, 2000.

[42] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012.

[43] R. Malhotra and M. Khanna. An exploratory study for software change prediction in object-oriented systems using hybridized techniques. *Automated Software Engineering*, 24(3):673–717, 2017.

[44] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda. Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In C. E. Cuesta, D. Garlan, and J. Pérez, editors, *Software Architecture*, pages 320–335, Cham, 2018. Springer International Publishing.

[45] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE TSE*, Jan. 2004.

[46] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 39–46, 2013.

[47] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[48] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[49] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[50] T. U. of Waikato. Weka 3: Data mining software in java. https://www.cs.waikato.ac.nz/ml/weka/, 2018.

[51] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. v. Staa. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *Software Engineering (SBES), 2014 Brazilian Symposium on*, pages 91–100, Sept 2014.

[52] V. Pareto and A. Page. Manuale di economia politica (manual of political economy). *Milan, Italy: Societa Editrice Libraia*, 1906.

[53] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SEN*, 1992.

[54] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[55] F. Provost. Machine learning from imbalanced data sets 101. In *Proceedings of the AAAI'2000 workshop on imbalanced data sets*, pages 1–3, 2000.

[56] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[57] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.

[58] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[59] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 303–312. IEEE, 2011.

[60] D. Romano, P. Raila, M. Pinzger, and F. Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 437–446. IEEE, 2012.

[61] A. S. Sayyad and H. Ammar. Pareto-optimal search-based software engineering (posbse): A literature survey. In *2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 21–27. IEEE, 2013.

[62] A. Shahbazian, D. Nam, and N. Medvidovic. Toward predicting architectural significance of implementation issues. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018.

[63] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, April 2003.

[64] R. Taylor, N. Medvidovic, and E. Dashofy. Software Architecture: Foundations, Theory, and Practice. 2009.

[65] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[66] V. Tzerpos and R. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE)*, 2000.

[67] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.

[68] L. Wilkinson. Revising the pareto chart. *The American Statistician*, 60(4):332–334, 2006.

[69] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. Cross-project build co-change prediction. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 311–320. IEEE, 2015.

[70] L. Xiao. Detecting and preventing the architectural roots of bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 811–813, New York, NY, USA, 2014. ACM.

[71] 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*, ICSE '16, pages 488–498, New York, NY, USA, 2016. ACM.

[72] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pages 46–55. ACM, 2015.

[73] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.