Detecting the Locations and Predicting the Maintenance Costs of Compound Architectural Debts

Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng

***Abstract*—Architectural Technical Debt (*ATD*) refers to sub-optimal architectural design in a software system that incurs high maintenance "interest" over time. Previous research revealed that *ATD* has significant negative impact on daily development. This paper contributes an approach to enable an architect to precisely locate *ATDs*, as well as capture the trajectory of maintenance cost on each debt, based on which, predict the cost of the debt in a future release. The *ATDs* are expressed in four typical patterns, which entail the core of each debt. Furthermore, we aggregate compound *ATDs* to capture the complicated relationship among multiple *ATD* instances, which should be examined together for effective refactoring solutions. We evaluate our approach on 18 real-world projects. We identified *ATDs* that *persistently* incur significant (up to 95% of) maintenance costs in most projects. The maintenance costs on the majority of debts fit into a linear regression model—indicating stable "interest" rate. In five projects, 12.1% to 27.6% of debts fit into an exponential model, indicating increasing "interest" rate, which deserve higher priority from architects. The regression models can accurately predict the costs of the majority of (82% to 100%) debts in the next release of a system. By aggregating related *ATDs*, architects can focus on a small number of cost-effective compound debts, which contain a relatively small number of source files, but account for a large portion of maintenance costs in their projects. With these capabilities, our approach can help architects make informed decisions regarding whether, where, and how to refactor for eliminating *ATDs* in their systems.**

***Index Terms*—software architecture, technical debt, software maintenance, debt quantification and prioritization**

I. Introduction

Technical Debt (TD) is a metaphor that describes the shortcuts taken in software development for achieving immediate goals but compromising the long-term benefits [1]. Architectural *TD* (*ATD*), a subset of *TD*, refers to sub-optimal architectural design decisions in a software system [2]–[4]. Previous research revealed that *ATD* has the most significant negative impact on the long-term success of a project [2], [3], [5], compared to other types of *TD*. Architects have a compelling need for an effective approach to detect and manage architectural *TD* [6]. This paper contributes an approach to 1) detect compound architectural *TD* that are composed of multiple, related architectural flaws, whose elimination should be treated together; and 2) quantify the "interest" and predict the future "cost" of different instances of *ATD* to help architects make informed refactoring decisions.

Although *ATD* has received significant attention [7]–[13], existing approaches suffer from several limitations. First, existing approaches identify usually architectural smells and anti-patterns [4], [14]–[18], which are not true "debts" if they do not generate maintenance "interest". In addition, architectural smells/anti-patterns usually contain focused groups of files with flawed architectural connections. However, multiple instances of anti-patterns may aggregate to form more complicated debts, i.e. we call them the "compound" *ATDs*, that should be treated together for eliminating the flaws. For instance, *Unstable Interface* [14] is featured by a change-prone "interface", changes to whom frequently propagate to its dependents. It is possible that one of the dependents is an unstable "interface" with respect to another group of files. Thus, they should be treated together for eliminating the "unstableness". Another main challenge in managing *ATD* is to quantify the "interest" for making informed refactoring decisions [19]–[21]. Currently, this relies heavily on architects' estimation and experience [22]–[25]. Architects need a predictive and repeatable approach to quantify the current and future cost on debts [19]–[21].

To overcome the above challenges, this paper propose a new approach to detect *ATDs* that incur high maintenance costs over time. We define an *ATD* as a tuple consisting of: 1) a group of architecturally connected files, and 2) a model describing the trajectory of maintenance cost on this group of files. Based on this definition, we contribute an approach to automatically locate *ATDs*. Once we locate each debt we model its growth using regression models. Our ATD detection has two parts. We first create a novel *history coupling probability* (HCP) matrix to manifest the probability of changing one file when another file is changed. Then we index file groups through the lens of 4 patterns of prototypical architectural flaws that have been shown to correlate with reduced software quality [26], namely *hub*, *anchor-submissive*, *anchor-dominant*, and *modularity violation*.

Given an *ATD*, we quantify the maintenance costs spent on the files involved in the debt. The actual maintenance costs in a software project—time and money—are almost never directly measurable. Thus, we approximate maintenance costs by bug-related churn—the lines of code committed to fix bugs. From the costs incurred in each release, we can model the growth trend using various regression models: linear, logarithmic, exponential or polynomial. These models represent scenarios of stable, reducing, increasing, and fluctuating maintenance interest rates respectively. This ensures that the identify file groups form true "debts" that incur maintenance "interests". As we will show in the evaluation, architects can use the debt model to predict the cost of a debt in a future release.

Finally, we aggregate *ATD* instances to form compound debts

with more complicated connections. Through the lens of the four debt patterns, each *ATD* instance is composed of an anchor file, which is the core of the debt, and a group of member files, which form the respective pattern centered around the anchor file. We aggregate compound *ATDs* through two criteria: 1) Transitive Anchors, where the anchor file of a debt is a member file of another debt. This reveals the hierarchical pattern in a compound debt; and 2) Compound Anchors, where the anchors files of two debts share overlapping members. Architects need to examine the complicated connections aggregated in compound debts for developing effective refactoring solutions.

We aim to evaluate the effectiveness of our approach in identifying and quantifying *ATDs* in software projects. Specifically, we evaluate whether the proposed approach can provide useful insights for software architects to make informed decisions regarding whether, where, and how to refactor to reduce *ATDs* in their software projects. We therefore focus on three questions.

- RQ1: Can significant *ATDs* in software projects be identified using a systematic approach?
- RQ2: Is it possible to quantify and accurately predict the future cost of an *ATD*?
- RQ3: Are compound *ATDs* common in software projects, and do they form cost-effective refactoring candidates?

As shown in our evaluation results, our approach has the potential to enable an architect to precisely locate architectural debts, in terms of identifying the source files and how they are involved in *ATDs*. Our approach also captures the trajectory of maintenance costs for each *ATD*. Based on this an architect can estimate the cost of a debt in a future release. Furthermore, the debt patterns described in this paper identify design flaws—architectural connections that incur high maintenance costs in a project—which should be analyzed and, if the debt is high enough, refactored. The *ATD* detection and quantification approach in this paper can be fully automated through mining a software repository. Architects can repeat this approach through the life-cycle of a software project, and thus make informed decisions to managed *ATD*.

The key contributions and novelty of our approach are

- An automatic approach to identify and quantify *ATDs* by mining project repositories. We are the first to combine structural and evolutionary connections to identify patterns that lead to true *ATDs* with significant maintenance costs.
- Four atomic *ATDs* patterns—hub, anchor submissive, anchor dominant, and modularity violation—that are potentially refactoring targets.
- The adoption of regression models to capture the maintenance cost trajectory of each *ATD*, to estimate their future costs. This provides an objective means for architects to prioritize debts based on predictions of a debt's future costs. To the best of our knowledge, we are the first to leverage regression models to estimate future debt costs.
- The compound *ATD* aggregation approach, which captures the connections among atomic patterns. This work is the first to consider the connections that form compound *ATDs*.
- A quantitative evaluation of the effectiveness of our approach on 18 large-scale software projects with varying characteristics.

The rest of this paper is organized as follows. Section II introduces the background of this paper. Section III provides the formal definition of *ATD* in the scope of this paper. Section IV introduces our *ATD* identification and quantification approach. Section V introduces the research questions and evaluation subjects. Section VI presents the evaluation results. Section VII discusses how architects can benefit from the proposed approach and the factors that may impact the results of our approach. Section VIII discuss related work. Section IX discuss the threads to validity and limitations of our approach. Section X concludes this paper.

II. Background

We now introduce the key concepts our work is based on.

Design Rule Space. In our prior work [27] we proposed a novel architectural model—*Design Rule Space*(DRSpace)—based on the Baldwin and Clark's *design rules* [28]. Building upon existing definitions of software architecture [29], we characterize a software architecture as a set of overlapping DRSpaces, each reflecting a unique aspect of the architecture. Each DRSpace is a subset of a system's source files and some kind of relationships (dependencies) among these files. Each DRSpace has one or more "leading file(s)", which all other files in the DRSpace depend on, directly or indirectly. The leading files are usually the files with architectural importance, such as interfaces or abstract classes, which we call *Design Rules*. The relations within a DRSpace may be structural—such as "*Implement*", "*Extend*", "*Call*"—or relations may be based on history coupling between source files—indicating the number of times two files changed together as recorded in the project's revision history.

There are numerous DRSpaces in any non-trivial software system, e.g., each dependency type forms a DRSpace: files connected by "*Extend*" and "*Inherit*" relationships form an *inheritance* DRSpace, and files that are coupled in the project's revision history form an *evolution* DRSpace. We created an *architecture root* detection algorithm that computes the intersection between DRSpaces and the project's "error space"—the set of error-prone files in a system [27]. We showed that the majority of the error-prone files are concentrated in just a few DRSpaces, suggesting that these error-prone files are not islands–they are architecturally connected [27]. Furthermore, we showed that these DRSpaces frequently contain architectural issues (flaws) that, we claim, are the root causes of error-proneness.

In this paper, we capture the architecture of a software system following the *DRSpace* modeling approach. The *ATD* detection approach is based upon the *DRSpace* modeling as we will introduce in detail in Section IV.

Design Structure Matrix (DSM). We use a DSM [28] to represent a DRSpace. Each element in the DSM is a source file, and each cell represents the relationships between the

	1	2	3	4	5	6	7	8
1 AbstractType	(1)							
2 UUIDSerializer	,100%	(2)	,50%			,100%		,50%
3 UUIDType	ext,dp,33%	dp,	(3)				,33%	,50%
4 AbstractCell	dp,50%			(4)				
5 TypeCast	dp,33%		,33%		(5)		,33%	,33%
6 IntegerSerializer	,100%	,100%	,50%			(6)		,50%
7 LongType	ext,dp,67%		,67%		,33%		(7)	dp,67%
8 DateType	ext,dp,40%		,60%				dp,40%	(8)

Fig. 1: DSM Example

file on the row and the file on the column. For example, Figure 1 is a DRSpace with leading file *ColumnParent*. Each cell shows the structural dependencies — "*implement*", or "*dp*" — between the file on the row and the file on the column, followed by the conditional probability of change propagation. In the original DRSpace [27], we used the number of times two files changed together in the project's revision history to represent their history dependency. In this paper, we replace this count with a *probability*. For example, cell[6,2] contains "*Implement*", meaning that the file on row 6, *CassandraServer*, implements the interface on row 2, *Cassandra*; cell[2,6] contains "48%", meaning that when *Cassandra* changes, there is a 48% probability that *CassandraServer* will change with it.

In this paper, we use DSMs to model source files and their relationships. In addition, instead of capturing an *ATD* as one snapshot using a DSM, we use a sequence of DSMs to reveal the growth of *ATDs* over time. In section VII, we will show an example ATD from an open source project, Camel, which evolves and grows over 11 releases. The snapshot of this debt in each release is represented in a separate DSM.

Architecture Issues. Our recent work [26] defined, implemented, and validated an algorithm for detecting recurring architectural issues in software systems, which we call *hotspot patterns*, including: 1) *unstable interface*, where an influential file changes frequently with its dependents in the revision history; 2) *modularity violation*, where structurally decoupled files frequently change together in the project's revision history; 3) *unhealthy inheritance*, where a super-class depends on its sub-class or where a client class depends on both a super-class and its sub-class; 4) *cyclic dependency*, where a set of files forms a dependency cycle. In the 9 projects we examined, we observed a strong correlation between the number of flaws a file has and: 1) the number of bugs reported and fixed in it, 2) the number of changes made to it, and 3) the amount of cost spent on it (in terms of committed lines of code to fix bugs and to make changes).

The four *ATD* patterns identified in this paper are generalizations of the four structural anti-patterns—namely, *unstable interfaces*, *modularity violations*, *unhealthy inheritance*, and *cyclic dependency*. In addition, to identify true *ATD* that incurs high maintenance costs over time, we developed a novel technique—the *history coupling probability* (HCP) matrix—to capture historical coupling among source files based on the evolution history. The identification of *ATD* relies on the combination of structural anti-patterns and historical coupling connections, as introduced in section IV.

III. *ATD* Definition

In this section, we formally define *Architectural Debt* (ATD) and illustrates an ATD example.

A. Definition

We define an *Architectural Debt (*ATD*)* as a group of *architecturally connected files* that persistently incur high maintenance costs over time. Each *ATD* is defined as a tuple of two elements:

$$ATD =< FileSetSequence, DebtModel > \quad (1)$$

The first element, $FileSetSequence$, is a sequence of file groups, each extracted from consecutive project releases:

$$FileSetSequence = (FileSet_1, FileSet_2..., FileSet_m) \quad (2)$$

where m is the number of releases that *ATD* impacts, $m \leq R$. Note that R is the total number of releases in a project. The $FileSet_r$, where $m = 1...m$, is the snapshot of the involved files in an *ATD* in release r. Note that the $FileSet_r$ changes dynamically with the architecture evolution of a project in different releases. However, the $FileSet_r$ in different releases are all originated from a same core file, which is called the *Anchor File*. We will explain the *Anchor File* in detail when we introduce the different debt patterns. When viewed (statically) in a release r, an *ATD* is a group of source files with flawed architectural connections, denoted as $FileSet_r$; when viewed (dynamically) via the long-term evolution of a system, an *ATD* is formed by the sequence of file groups in different releases, namely $FileSetSequence$.

The second element, $DebtModel$, is a regression model that describes the trajectory of the maintenance costs associated with each ATD. We use four representative regression models, namely the Linear, Logarithmic, Exponential, and Polynomial models to capture four general kinds of debt interests respectively: stable, decreasing, increasing, and fluctuating. We will discuss the details about how the $DebtModel$ is calculated later.

The first element $FileSetSequence$ identifies the location of a debt, in terms of which files are involved in different releases; while the second element $DebtModel$ quantify the maintenance interest of the debt over time.

IV. ATD Detection

There are hundreds and thousands of source files in a project. As the project evolves from release to release, the number of source files (in most cases) grows over time. To identify an *ATD*, we need to identify the two elements, namely $FileSetSequence$ and $DebtModel$. We search for $FileSetSequence$ just like searching for web pages on the Internet. Then we calculate the $DebtModel$ for each debt. The overall flow of our approach is illustrated in Figure 2, with the following five steps.

1) **Crawling**: this step collects the set of error-prone files from each release r, r from 1 to R, similar to crawling and collecting web pages.

2) **Indexing**: this step identifies (indexes) a specific file group, $FileSet$, starting from each error-prone file in each release, then locates sequences of related $FileSets$ in different releases as a $FileSetSequence$.
3) **Modeling**: this step quantify the maintenance costs associated with $FileSet_r$ in each time-stamp, release r. An *ATD* is identified as a $FileSetSequence$ whose costs gradually increase over time.
4) **Ranking**: this step ranks the identified *ATDs* according to the amount of maintenance costs they have accumulated in the project's evolution history.
5) **Aggregating**: this steps aggregates related *ATDs* into compound *ATDs* based on their relationship to help architects capture and examine the more complicated connections among debts.

Each of the above steps is fully automated by mining and analyzing the project repository, including the issue tracking system and revision history. In the following, we will elaborate each step in a separate subsection.

A. Crawling: Select Design Rule Spaces

In this step, we crawl the architectural connections among the error-prone files in a project, analogous to crawling web-pages from the Internet.

Firstly, we capture the software architecture of a system at release r as a set of overlapping design spaces:

$$SoftArch_r = \{DRSpace_1, DRSpace_2, ..., DRSpace_n\} \tag{3}$$

where n is the number of files in a project in release r. Each design space, namely $DRSpace_i$, in $SoftArch_r$ is a subset of the entire system in release r, composed of a leading source file, and all the other files that structurally depend on it (directly or indirectly). Thus each $DRSpace$ reveals a different aspect of the architecture [27].

Meanwhile, we retrieve the set of error-prone files, denoted as $ErrorSpace_r$, in each release r by mining the project revision history. Formally, $ErrorSpace_r$=$\{f_1, f_2, ..., f_n\}$. Each file $f_i \in ErrorSpace$ was revised to fix errors at least once between release 1 and r. By the definition: $ErrorSpace_r$ is a subset of $ErrorSpace_{r+1}$.

For each release r, we crawl $DRSpaces$ from $SoftArch_r$, which are led by the files from the $ErrorSpace_r$. More specifically, each selected space is led by a file in $ErrorSpace_r$.

$$SelectedDRSpace_r = Crawling(SoftArch_r, ErrorSpace_r) \tag{4}$$

If there are n files in $ErrorSpace_r$, there are n DRSpaces in $SelectedDRSpace_r$ for each release as the output of this *Crawling* step.

B. Indexing: Identify ATD Candidates

Now that for each release we have crawled a set of design spaces led by each error-prone file. In this step, we search for the $FileSetSequences$ which are the debt candidates from the $SelectedDRSpaces_r$, $r = 1...n$. The $FileSetSequences$ is a sequence of architecturally connected file set that persistently accumulate higher maintenance cost across multiple releases of a project. Thus, to identify the $FileSetSequences$, 1) we first calculate a simple history coupling model—HCP matrix—to capture the history evolution; and 2) then we match file groups using four *indexing patterns* to capture different architectural connection patterns in debts.

1) HCP Matrix: Previously, we used a symmetric DSM to represent how source files co-change together in the revision history [27]. Each cell in the DSM shows the number of times two files changed together. That is the number of times on cell[x,y] is identical to cell[y,x]. This model is not able to capture the direction of change propagation among files. To overcome this problem, we propose a new model: the **history coupling probability (HCP) matrix**. In this model, each cell records the *conditional probability* of changing the file on the column, if the file on the row has been changed, i.e., the odds of changes propagating from file to file.

Figure 3 shows an example of the creation of a HCP. Part 1 shows that 4 files A, B, C, and D, that change in 4 commits: Commit1$\{$A,B$\}$ (Commit1 changes A and B), Commit2$\{$A,B$\}$, Commit3$\{$B,D$\}$, and Commit4$\{$A,C$\}$. First, we compute the pair-wise conditional change probabilities for any pair of files. For example, the probability of changing file A, given that file C has changed, denoted by $Prob\{A|C\}$, is the number of times A and C change in the same commits divided by the total number of changes to C. Similarly, $Prob\{C|A\}$ is the number of times A and C change in the same commits divided by the total number of changes to A. Hence, $Prob\{A|C\}$ is $1/1$, indicating that A *always* changes with C, and $Prob\{C|A\}$ is $1/3$, indicating a probability of $1/3$ that C changes with A. In this relation, we label C as *dominant* and A as *submissive* because $Prob\{A|C\} > Prob\{C|A\}$. We compute the probabilities for every pair of files and get the graph in part 2 of Figure 3. It is the graph-representation of the *HPC* matrix.

For each release r of a project, we compute a HPC matrix (HPC_r), consisting of files in $ErrorSpace_r$, from the bug-fixing revision history between release 1 to release r.

2) Indexing Patterns: We search for the $FileSet_r$ in each release r by matching four patterns of prototypical architectural flaws. As mentioned earlier, the $FileSect_r$ of a debt dynamically evolve with the architecture evolution of a project. However, the $FileSect_r$ in different releases all contain one source file, named the *Anchor File*, from which the debt originate and accumulate over time. As an intuitive example, a base class, which frequently change with its child classes, could be the *Anchor File* and the child classes are the *Member Files* in a $FileSet_r$. We define $FileSet_r$ based on the notion of *Anchor File* and *Member Files* as:

$$FileSet_r = \{a, M_r | M_r = \{m_i : i \text{ from 1 to } n\} | \\ \forall m_i \in M_r, m_i \text{ architecturally connected with } a \text{ in release } r\} \tag{5}$$

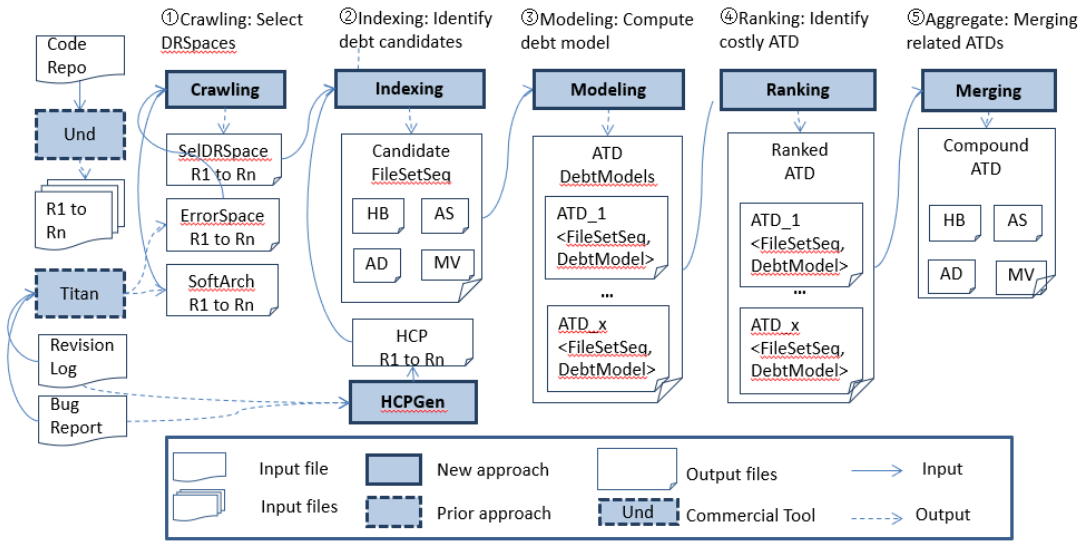


Fig. 2: Approach Framework

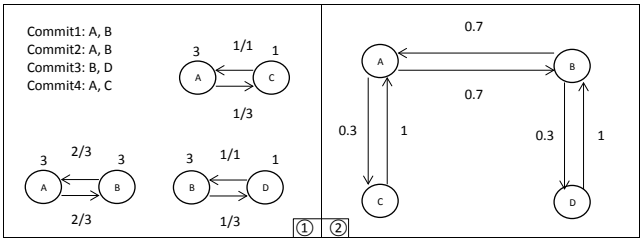


Fig. 3: Generate HPC Matrix

	1	2	3	4	5	6	7
1 PDA*Line	(1)	,100%	,100%	dp,100%	,100%	,100%	,100%
2 PDA*SquareCircle	,100%	(2)	,100%	dp,100%	,100%	,100%	,100%
3 PDA*FileAtt*	,100%	,100%	(3)	dp,100%	,100%	,100%	,100%
4 PDA*	dp,50%	dp,50%	dp,50%	(4)	dp,50%	dp,50%	dp,50%
5 PDA*Text	,100%	,100%	,100%	dp,100%	(5)	,100%	,100%
6 PDA*Link	,100%	,100%	,100%	Extend,dp,100%	,100%	(6)	,100%
7 PDA*Widget	,100%	,100%	,100%	Extend,dp,100%	,100%	,100%	(7)

A* stands for Annotation

Fig. 4: Hub

where $FileSect_r \in FileSetSequence$, a is the anchor file, and the files contained in M_r change with a in release r. We call M_r the member files of a in release r.

To explain the four indexing patters, we first define two boolean expressions to describe the relationships between two files (x and y) in release r: $S_r(x \rightarrow y)$ and $H_r(x \rightarrow y)$. $S_r(x \rightarrow y)$ means y structurally depends on x in release r. $H_r(x \rightarrow y)$ means x is dominant and y is submissive in their co-changes between release 1 to release r. In HCP_r, $HCP_r[x, y]$ is the probability of changing y, given x has changed. If $HCP[x, y] > HCP_r[y, x]$, then x is dominant and y is submissive. $HCP[x, y] = HCP_r[y, x]$ means x and y are equally dominant. Formally:

$$\begin{aligned} &\text{In release } r, \\ &S_r(x \rightarrow y) \text{ is true if } y \in DRSpace_r\_x, \text{otherwise it is false} \\ &H_r(x \rightarrow y) \text{ is true if } HCP[x, y] >= HCP_r[y, x] \\ &\wedge\, HCP[x, y] \neq 0, \text{otherwise it is false} \end{aligned} \tag{6}$$

For any pair of a and m in a $FileSet_r$, we identify 4 relationships: $S_r(a \rightarrow m)$, $S_r(m \rightarrow a)$, $H_r(a \rightarrow m)$, and $H_r(m \rightarrow a)$. Each relationship could be either true or false. We enumerated all 16 combinations of these 4 relationships. The 4 combinations with $H_r(a \rightarrow m)$ and $H_r(a \rightarrow m)$ $false$, which indicates that x and y are not likely to change together, are irrelevant to our analysis, since we need history to measure debt. From the remaining 12 possible combinations, we defined 4 indexing patterns—*Hub, Anchor Submissive, Anchor Dominant, Modularity Violation*. Each pattern corresponds to prototypical architectural issues that proved to correlate with reduced software quality [26].

Using any file $a \in ErrorSpace_r$ as the anchor file, we can identify its members to form the $FileSet_{r\_a}$. The members are identified by matching the structural dependency in $SelectedDRSpace_r$ and the evolutionary coupling in HCP_r through the lens of the 4 indexing patterns:

Hub—the anchor file and each member have structural dependencies in both directions and history dominance in at least one direction. The anchor is an architectural hub for its members. This pattern corresponds to cyclic dependency, unhealthy inheritance (if the anchor file is a super-class or interface class), and unstable interface (if the anchor file has many dependents). Informally such structures are referred to as "spaghetti code", or "big ball of mud". A $FileSet_{r\_a}$ with anchor file a in release r that matches a *hub* pattern is denoted by $HBFileSet_{r\_a}$ and is calculated as:

$$\begin{aligned} &HBFileSet_{r\_a} = Index_{HB}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r | \forall m \in M_r, S_r(a \rightarrow m) \wedge S_r(m \rightarrow a) \\ &\wedge\, (H_r(a \rightarrow m) \vee H_r(m \rightarrow a))\} \end{aligned} \tag{7}$$

Figure 4 is a Hub *FileSet* for the PDFBox project, anchored by *PDAnnotation*. The dark grey cell represents the anchor file

	1	2	3	4	5	6	7	8
1 AbstractType	(1)							
2 UUIDSerializer	,100%	(2)	,50%			,100%		,50%
3 UUIDType	ext,dp,33%	dp,	(3)				,33%	,50%
4 AbstractCell	dp,50%			(4)				
5 TypeCast	dp,33%		,33%		(5)		,33%	,33%
6 IntegerSerializer	,100%	,100%	,50%			(6)		,50%
7 LongType	ext,dp,67%		,67%		,33%		(7)	dp,67%
8 DateType	ext,dp,40%		,60%				dp,40%	(8)

Fig. 5: Anchor Submissive

	1	2	3	4	5	6
1 ColumnParent	(1)	,100%	,50%	,41%	,50%	,100%
2 Cassandra	dp,	(2)				,48%
3 CliClient	dp,	dp,	(3)			
4 Column*Reader	dp,	dp,		(4)		
5 ThriftValidation	dp,				(5)	
6 CassandraServer	dp,	Implement,			dp,	(6)

Fig. 6: Anchor Dominant

	1	2	3	4	5	6	7	8
1 JMXETPEMBean	(1)	,100%	,44%	,50%		,100%	,100%	,50%
2 DebuggableTPExecutor		(2)	,31%					
3 StorageService			(3)	dp,	dp,Use,			
4 ColumnFamilyStore			dp,	(4)				
5 MessagingService			dp,		(5)		dp,	
6 NodeProbe			,44%		dp,	(6)		
7 StatusLogger		,50%		dp,50%	dp,	,50%	(7)	
8 JMXCTPExecutor	,50%	,100%	,31%	,100%	,50%	,50%	,50%	(8)

Fig. 7: Modularity Violation

(cell[4,4] for *PDAnnotation*). The cells showing the historical and structural relationships between member files and the anchor file are in lighter grey. In this *HBFileSet*, the anchor file structurally depends on each member file, and each member file also structurally depends on the anchor file. When the anchor file changes, each member file has a 50% probability of changing as well. When a member file changes, the anchor file always changes with it. A *HBFileSet* is potentially problematic because the anchor file, like a hub, is strongly coupled with every member file both structurally and historically.

Anchor Submissive—each member file structurally depends on the anchor file, but each member historically dominates the anchor. This pattern corresponds to an unstable interface, where the interface is submissive in changes. An *Anchor Submissive FileSet* with anchor a in release rt is:

$$\begin{aligned} &ASFileSet_{r\_a} = Index_{AS}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r | \forall m \in M_r, S_r(a \to m) \wedge \\ &\rightharpoonup S_r(m \to a) \wedge H_r(m \to a) \end{aligned} \tag{8}$$

Figure 5 shows an *ASFileSet* with anchor *AbstractType* in Cassandra. Each member file directly or indirectly depends on the anchor file, but when the member files change, the anchor file changes with each of them, with historical probabilities of 33% to 100%. A *ASFileSet* is problematic because history dominance is in the opposite direction to the structural influences: the anchor should influence the member files, not the other way around.

Anchor Dominant—each member file structurally depends on the anchor file and the anchor file historically dominates each member file. This pattern corresponds to the other type of unstable interface, where the interface is dominant in changes. An *Anchor Dominant* $FileSet$ with anchor a in release rt can be calculated as:

$$\begin{aligned} &ADFileSet_{r\_a} = Index_{AD}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r | \forall m \in M_r, S_r(a \to m) \wedge \\ &\rightharpoonup S_r(m \to a) \wedge H_r(a \to m)\} \end{aligned} \tag{9}$$

Figure 6 shows an $ADFileSet$ calculated using anchor *ColumnParent* in Cassandra. Each member file (from row 2 to row 6) structurally depends on (cell[2 to 6:1]) the anchor file (row 1), and when the anchor file changes, the member files change as well with probabilities from 41% to 100% (cell[1:2 to 6]). A $ADFileSet$ presents potential problems where the anchor file is unstable and propagates changes to member files that structurally depend on it.

Modularity Violation—there are no structure dependencies between the anchor and any member, however they historically couple with each other. In a *modularity violation* the anchor and member files share assumptions ("secrets") that are not represented in any structural connection. A $MVFileSet$ with anchor a in release r is calculated as:

$$\begin{aligned} &MVFileSet_{r\_a} = Index_{MV}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r | \forall m \in M_r, \rightharpoonup S_r(a \to m) \wedge \rightharpoonup S_r(m \to a) \\ &\wedge (H_r(m \to a) \vee H_r(a \to m))\} \end{aligned} \tag{10}$$

Figure 7 is a $MVFileSet$ with anchor *JMXCTPExecutor* (row 8) in Cassandra. The anchor file, on the bottom of the matrix, is structurally isolated from the member files. However, when the anchor file changes, there are historically 31% to 100% probabilities that the member files change as well, and when the member file *JMXETPEMBean* (on row 1) changes, the anchor file has a 50% chance to change with it. This pattern identifies potential problems where the anchor file and the member files share common assumptions, without explicit structural connections, and these assumptions are manifested by historical co-change relationships.

For each release r, we use each a in $ErrorSpace_r$ as the anchor file to calculate a $FileSet$ for each of the 4 patterns: HB, AS, AD, and $MV\ FileSet_{r\_a}$. The *FileSetSequence* in the *Hub* pattern with anchor file a is denoted by *HBFileSetSequence*$_a$. Similarly, for anchor a, we can identify *AS*, *AD*, and *MV FileSetSequence*$_a$. Using any error-prone file as the anchor, we can identify 4 *FileSetSequences*, each of which is an *ATDCandidate*.

As a result, for each $a \in ErrorSpace_r$ and for each release r , we can exhaustively detect $4*|\cup_{r=1}^{n} ErrorSpace_r|$ candidates, which equals $4*|ErrorSpace_n|$ because $ErrorSpace_n$ is a super set of all $ErrorSpaces$ in earlier releases.

C. Modeling: Build Regression Model

Now that we have identified the first element of a potential debt, namely *ATDCandidate*, which is a $FileSetSequence$ composed of the $FileSet_r$ with anchor a in different releases.

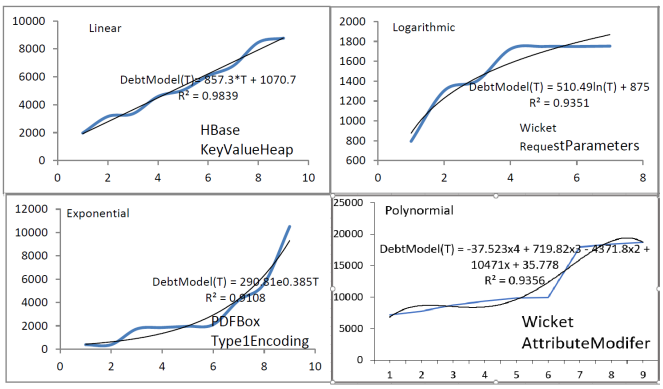


Fig. 8: 4 Types of Regression Model

Now, we need to: (1) quantify the maintenance costs associated with each $FileSet$ within a *FileSetSequence* to filter out unqualified candidates, and (2) calculate the *DebtModel* that can describe the cost trajectory of a debt over releases.

1) Quantify ATDCandidates*:* From each *FileSetSequence*, we first exclude any $FileSet_r$, that contains just 1 file, since this can not involve architecture problems. Next, we define the **age** of a *FileSetSequence* as the number of $FileSets$ in it after singleton $FileSets$ are filtered out. Then, for each $FileSet_r$, we measure the **maintenance cost**, denoted by $Cost\_FileSet_r$, that associated with the involved source files by the end of release r. For any file $f \in FileSet_r$, we estimate the maintenance cost as the amount of error-fixing churn expended on it by the end of release r (i.e. between release 1 and release r). We denote the maintenance cost for file f between release 1 and release r as $ErrorChurn_{r\_f}$. $Cost\_FileSet_r$ is the sum of maintenance costs on each file in the set:

$$Cost\_FileSet_r = \sum\nolimits_{\forall f \in FileSet_r} ErrorChurn_{r\_f} \qquad (11)$$

Not every *FileSetSequence* qualify as a true debt. The key characteristic of a debt is its long-lasting impacts. In other words, a true debt should survive a long time and incur increasing maintenance cost over time. Therefore, we examine the age of each *FileSetSequence* to filter out short-lived sequence. Second, *FileSetSequence* should require increasing maintenance cost over time. During each release window, the debt should have incurred more maintenance cost. That is, let $FileSet_i$ and $FileSet_{i+1}$ be two consecutive snapshots of an $ATDCandidate$, the cost on $FileSet_{i+1}$ should be higher than the cost on $FileSet_i$. Formally, the criterion for a $ATDCandidate$ to qualify as a true debt is:

$$\begin{cases} age >= n/c; \\ Cost\_FileSet_{i+1} > Cost\_FileSet_i, i = 1...age - 1. \end{cases}$$

The parameter c is a tunable. We use c=2, meaning that *FileSetSequence* is persistent for at least half of the releases in a project. A candidate in a younger age is not a meaningful debt (at least not yet). The second condition requires that the maintenance costs on *FileSetSequence* increase over time.

2) Formulate DebtModel*:* For each qualified *FileSetSequence*, we calculate a regression model, denoted as $DebtModel$, to describe the trajectory of its maintenance cost over time. The purpose is to capture the "interest rate" of a debt. We employ four typical regression models that indicates different types of "interest": linear, logarithmic, exponential, and polynomial (up to degree 10). Figure 8 shows typical examples of these 4 models. Each model represents a typical "interest" type. The linear model (part 1 of Figure 8) indicates a stable interest rate. This means that developers pay a stable amount of maintenance cost on this debt in each release window. The logarithmic model (part 2) indicates a decreasing interest rate. As show in the example, the maintenance cost of a debt increase more slowly over time. We conjecture that this could happen in the scenario of a successful refactoring, which made it easier to make the next change on the group of files. The exponential model (part 3) indicates an increasing interest rate. This could happen when the group of files become extremely tangled. It is especially likely to happen at the beginning of a project when developers have not started to worry about the architecture. Finally, the polynomial model (part 4) indicates a fluctuating interest rate. This is likely to happen due to uncontrolled factors, such as resource allocation.

Following equation 11, we calculate the maintenance cost—$Cost\_FileSet_r$ for each $FileSet_r$ in a *FileSetSequence*. Thus, the cost associated with a *FileSetSequence* form an array that we call $Cost\_Array$. $Cost\_Array[i]$ = $Cost\_FileSet_r$, where $FileSet_r$ is the ith element of *FileSetSequence*. We define an integer array $T[i] = r$, where r is the release number of the ith element in *FileSetSequence*. Each release r is numbered by its order in the release in history. In the $DebtModel$ of a *FileSetSequence*, the $Cost\_Array$ is the independent value and T is the dependent value. We developed a *ModelSelector* algorithm to select a regression model that best describes the relationship between T and $Cost\_Array$. The formula of the regression model are returned as $DebtModel$:

$$DebtModel = ModelSelector(CostArray, T) \qquad (12)$$

The following figure shows the pseudocode of the $ModelSelector$ algorithm.

To help us pick the best regression model, we introduce a parameter R^2_{thresh} (R^2 threshold) [1] which ranges from 0 to 1 in the *ModelSelector*. A regression model can be selected only if the R^2 associated with the regression model is higher than R^2_{thresh}. However, we do not pick the best model strictly based on the highest R^2. Our heuristic is to prioritize the linear regression model, as long as its R^2 reaches the threshold R^2_{thresh}. As shown in the pseudocode, between line 1 to line 5, we first calculate a linear regression model, and this model is returned only if its R^2_{Lin} is greater than R^2_{thresh}. Otherwise, we try both logarithmic and exponential models (line 6 to line 9), and select the model whose R^2 reaches the threshold line 16 to 21). If they both reach R^2_{thresh}, *ModelSelector* returns the model with a higher R^2 (line 10 to line 15). Finally, if none of the linear, logarithmic, or exponential model fit, we calculate a polynomial model of degree up to 10 (line 22).

[1] R^2 is a statistical measure of how close the data are to the regression line.

Algorithm 1 ModelSelector (CostArray, T)

$model_{lin} = LinearFit(CostArray, T)$
$R^2_{Lin} = model_{Lin}.R^2$
if $R^2_{Lin} >= R^2_{thresh}$ **then**
  return $model_{Lin}$
end if
$model_{Log} = LogFit(CostArray, T)$
$R^2_{Log} = model_{Log}.R^2$
$model_{Exp} = ExpFit(CostArray, T)$
$R^2_{Exp} = model_{Exp}.R^2$
if $R^2_{Log} >= R^2_{thresh}$ and $R^2_{Exp} >= R^2_{thresh}$ **then**
  if $R^2_{Log} >= R^2_{Exp}$ **then**
    return $model_{Log}$
  end if
  return $model_{Exp}$
end if
if $R^2_{Log} >= R^2_{thresh}$ **then**
  return $model_{Log}$
end if
if $R^2_{Exp} >= R^2_{thresh}$ **then**
  return $model_{Exp}$
end if
$model_{poly} = PolyFit(CostArray, T)$
return $model_{poly}$

A polynomial model where $R^2_{poly} >= R^2_{thresh}$ or the degree reaches 10, whichever is satisfied first, is selected.

The rationale of the heuristic in *ModelSelector* is that the linear, logarithmic, and exponential models present three typical types of penalty interest rate: stable, decreasing, and increasing. The polynomial model, however, catches all the fluctuations of the maintenance cost, which is very likely a result of noise due to extraneous factors, and will always be picked if based on the highest R^2. For example, the debt in part 1 of Figure 8, intuitively a linear model ($DebtModel(r) = 857*r+1070$ with R^2 of 0.98), can fit into a polynomial model $DebtModel(r) = -2*r^6+59*r^5-680*r^4+3874*r^3-11342*r^2+16538*r-6466$, with a higher R^2 (0.99). The polynomial model fits better (higher R^2), but the linear model is more meaningful.

The selection of *DebtModel* completes the ATD identification.

D. Ranking: Identify High-maintenance ATD

We have identified different *ATDs* following the first three steps, however the identified debts have the varying severity, costing different amount of maintenance cost. Practitioners should prioritize debts with higher maintenance costs. Therefore, this step ranks all the identified architectural debts according to their cumulative maintenance cost.

We define a pair p_f = $\langle f, ErrorChurn_f \rangle$, where f is an error-prone file, and $ErrorChurn_f$ is the maintenance cost associated with f, approximated by error-fixing churn on f. Let *CostMap* be the set of p_f, such that $\forall f \in ErrorSpace_n$ (n is the latest release), there exists a $p_f \in$ *CostMap*. *CostMap* is one of the inputs to the *ranking* algorithm. The other input is the identified $ATDs$.

$$RankedDebts = ranking(ATDs, CostMap) \tag{13}$$

In the ranking algorithm, we identify the most significant *ATDs* according to *CostMap* iteratively. In each iteration, we select *maxATD* that account for the largest portion of cost for files in *CostMap* from *ATDs*. The cost for duplicate files is excluded, and the iteration terminates when all *ATDs* are ranked. The top debts returned account for the largest maintenance cost, and deserve more attention and higher priority for refactoring.

E. Aggregating: Merge Compound ATDs

In addition to ranking debts based on costs, practitioners also need to investigate the relationship among debts to effectively "pay-off" the debts. Thus, this step aims to analyze the relationship among debts and merge related debts into compound debts which capture the complicated inter-relationship among debts. We believe that this step is necessary for the practitioners to develop an effective refactoring solution for debts that form more complicated structural patterns, and save duplicated effort on reviewing debts that have significant overlap.

The identified *ATDs* may share two different types of relationship between two debts $D_x = \{a_x, M_x\}$ and $D_y = \{a_y, M_y\}$:

1) *Transitive Anchors*: The anchor file of a debt is a member file of another debt. Formally,

$$Transitive\_Anchor(D_y, D_x) \text{ is true, if } a_x \in M_y \tag{14}$$

This indicates that the change propagates from the anchor of D_y, a_y, to its member files M_y, and then through $a_x \in M_y$ to the member files of D_x, M_x. This change propagation, through multiple debts, is analogous to the well-known "ripple effect". Figure 9 shows an example from Hadoop formed by three *Anchor Dominant* debts. The related files and cells of each debt are highlighted in a different background color to facilitate understanding. The first debt contains files on rows 1, 2, and 13. The second debt contains files from rows 3 to 13. The third debt contains files from rows 13 to 17. The member files in each debt structurally depend on the anchor file and historically change with the anchor file as well. For example, in debt 2, the files on rows 4 to 13 all structurally depend on the anchor *fs.FSDataInputStream* (row 3). When this anchor changes, there is a 16% chance that these member files will change as well. These three debts share *Transitive Anchor* relationships through the member file, *security.Credentials* (row 13). *security.Credentials* is the anchor of debt 3. This indicates that the changes tend to propagate from two different anchors, *io.WritableUtils* (row 1) and *fs.FSDataInputStream* (row 3), to their member *security.Credentials*, which further propagates changes to its members on rows 14 to 17. Note that there exists non-trivial history change coupling directly from the anchor and member files of debt 1 to the member files of debt 3, as well as from the anchor

and member files of debt 2 to the member files of debt 3. This is consistent with the complicated *Transitive Anchors* relationship that we have detected.

2) *Compound Anchors*: Two debts contain overlapping member files, and thus the two original anchor files should form compound anchors.

$$Compound\_Anchors(D_y, D_x) \text{ is true, if } M_x \cap M_y \neg \emptyset \quad (15)$$

This indicates that the two anchor files, a_x and a_y, propagate changes to the same set of member files in $M_x \cap M_y$. Therefore, these two anchor files, a_x and a_y, should be treated as the compound anchor after the aggregation. For example, Figure 10 is an example of such relationship between two *HUB* debts from HBase. The first debt is composed of files from row 1 to row 7, with the anchor on row 1, namely *AssignmentManager*. The second debt is composed of files from row 6 to row 9, with the anchor on tow 9, namely *Hmaster*. This debt is featured by the cyclic dependencies between each member file and the anchor file. Meanwhile, whenever each member file changes, there is a significant chance (44% to 100%) that the anchor file will change with it. Similarly, the files and related cells of the two debts are highlighted in green and orange background colors to help understanding. Similarly, whenever the members file change, the anchor file has a 37% to 54% change to change as well. As we can see from the view, there are two member files on row 6 and row 7, which are contained in both debts. Therefore, we highlight the files and cells in blue background. The two anchors of the *HUB* debts form the *compound anchor* for the two overlapping members. Not only this makes the structure of this merged debt more complicated, but also the files in debt 1 are also likely to propagate changes to the anchor file of debt 2. Therefore, it is important for the practitioners to capture this type of relationship among debts.

In Step2—*Indexing Patterns*, each debt pattern is retrieved from a single file as the *anchor*. The above aggregation strategies capture the complicated relationship among the *anchors*, which in turn form the compound *ATDs*. The *Transitive Anchors* capture the "ripple effects" among the four atomic debt patterns. The *Compound Anchors* captures cases where a debt originates from multiple files as the anchor. On the one hand, it is necessary for developers to develop an effective refactoring solution for eliminating the architectural flaws underlying the compound debts. For example, if two debts share the *Transitive Anchors*, the debts cannot be eliminated completely if the developers only examine one of the debts. On the other hand, if the *compound anchors* share a large number of overlapping members, it is more effective for the developers to treat the compound anchors together to avoid repeated effort.

We define a *Compound-ATD* as being composed of multiple related *ATDs*. We create a $DebtMerger$ algorithm to aggregate related debts into *Compound ATD* in two phases.

$$CompoundATDs = DebtMerger(ATDs) \quad (16)$$

In the first phase, we merge debts based on the *Transitive Anchors* relationship. We form a merge graph, G_{m_ta}, where the nodes are the original debts, and the edges are the *Transitive Anchors* relationship. If $Transitive\_Anchor(D_y, D_x)$ is true, there is a merge edge from D_x to D_y. This means that D_x should be merged into D_y, since D_y is at a higher propagation hierarchy, i.e. its anchor propagate changes to D_x's anchor. Next, we use a simple graph traversal algorithm to find all the sub-graphs in G_{m_ta}. Each sub-graph is a group of debts that should be merged together due to the *Transitive Anchors* relationship.

The second phase merges debts based on the *Compound Anchors*. Similarly, we calculate another merge graph, namely G_{m_om}, where the nodes are the output of phase 1 (or the original debts if the output of phase 1 is not appropriate), and the edges are the *Compound Anchors* relationship. For any two debts, D_x and D_y, we calculate their weighted relationship in two directions, namely $W_{D_x \to D_y} = \frac{|D_x \cap D_y|}{|D_x|}$ and $W_{D_y \to D_x} = \frac{|D_x \cap D_y|}{|D_y|}$. They measure the percentage of overlapping files between D_x and D_y in D_x and in D_y respectively. In G_{m_om}, there is a merge-edge from D_x to D_y, indicating that D_x can be merged into D_y when the following condition holds:

$$W_{D_x \to D_y} \geq W_{D_y \to D_x} \text{and } W_{D_x \to D_y} \geq Thred_{Overlap} \quad (17)$$

In our experiment, we pick $Thred_{Overlap} = 0.5$. The rationale of this heuristic is that if $W_{D_x \to D_y} \geq 0.5$, it indicates that the majority of files in D_x are also member files in D_y, therefore D_x should be merged into D_y. However, if $W_{D_x \to D_y}$ and $W_{D_y \to D_x}$ are both ≥ 0.5, it indicates that these two debts share mutually significant overlap with each other. In this case, we merge the smaller debt into the larger debt. For example, if $W_{D_x \to D_y} > W_{D_y \to D_x}$, we merge D_x to D_y, since D_y contains more source files besides the overlapping part. We did not consider overlap less than 50%, since it is not significant for a merge. For example, if two large debts only have one overlapping member file, it does not make sense to merge them. In section VII-E, we will discuss the impact of $Thred_{Overlap}$ on the merging results.

For a particular note, we merge the same type of debts. That is, we do not merge a *AD* debt and a *MV* debt even if they share the two relationship. This is because debts in the same type represents the same typical architectural flaws, thus practitioners benefit from reviewing related, same type of debts together.

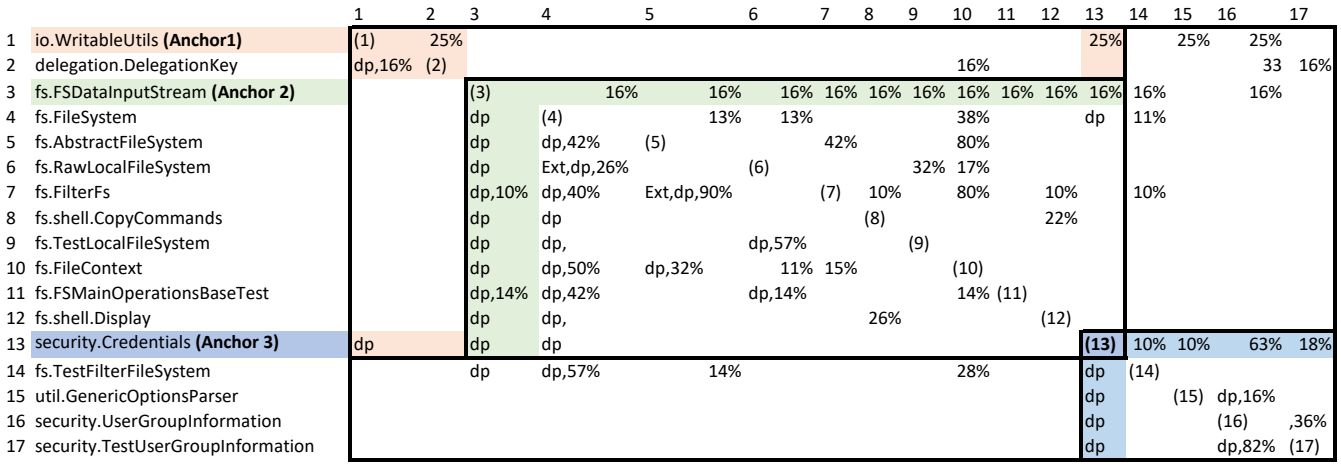
V. Evaluation Questions and Subjects

A. Research Questions

We aim to answer the following research questions:

- **RQ1: Can the proposed approach identify *ATDs* that deserve attention?** This RQ aims to evaluate whether our approach can identify file groups that cause significant

Fig. 9: Transitive Anchors in Hadoop



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	io.WritableUtils **(Anchor1)**	(1)	25%											25%		25%	25%	
2	delegation.DelegationKey	dp,16%	(2)								16%						33	16%
3	fs.FSDataInputStream **(Anchor 2)**			(3)		16%	16%	16%	16%	16%	16%	16%	16%	16%	16%		16%	
4	fs.FileSystem			dp	(4)		13%	13%			38%			dp	11%			
5	fs.AbstractFileSystem			dp	dp,42%	(5)			42%		80%							
6	fs.RawLocalFileSystem			dp	Ext,dp,26%		(6)			32%	17%							
7	fs.FilterFs			dp,10%	dp,40%	Ext,dp,90%		(7)	10%		80%		10%		10%			
8	fs.shell.CopyCommands			dp	dp				(8)				22%					
9	fs.TestLocalFileSystem			dp	dp,		dp,57%			(9)								
10	fs.FileContext			dp	dp,50%	dp,32%		11%	15%		(10)							
11	fs.FSMainOperationsBaseTest			dp,14%	dp,42%		dp,14%				14%	(11)						
12	fs.shell.Display			dp	dp,				26%				(12)					
13	security.Credentials **(Anchor 3)**	dp		dp	dp									**(13)**	10%	10%	63%	18%
14	fs.TestFilterFileSystem			dp	dp,57%		14%				28%			dp	(14)			
15	util.GenericOptionsParser													dp		(15)	dp,16%	
16	security.UserGroupInformation													dp			(16)	,36%
17	security.TestUserGroupInformation													dp			dp,82%	(17)

Fig. 10: Compound Anchor with Overlapping Members in HBase

		1	2	3	4	5	6	7	8	9
1	AssignmentManager **(Anchor1)**	(1)	dp	dp	dp	dp	dp	dp		29%
2	handler.ClosedRegionHandler	dp,81%	(2)	63%			27%	45%		63%
3	handler.OpenedRegionHandler	dp,78%	30%	(3)			13%	21%		39%
4	AssignCallable	dp,100%		50%	(4)	50%		50%		100%
5	UnAssignCallable	dp,100%			33%	(5)				66%
6	handler.DisableTableHandler **(Overlap)**	dp,46%	20%	20%			(6)	86%		dp,46%
7	handler.EnableTableHandler **(Overlap)**	dp,44%	17%	17%			44%	(7)		dp,37%
8	HMasterCommandLine	20%							(8)	Extend,dp,54%
9	Hmaster **(Anchor 2)**	dp,23%					dp	dp	dp	(9)

maintenance costs in software projects that are worthy of attention. We will address the question from different perspectives in three sub-questions :

- **RQ1-1: Do the file groups identified in *ATDs* generate more maintenance costs than what one would expect given their sizes?** Here, we investigate whether the *ATDs* identified by our approach account for significant maintenance costs in a project's evolutionary history. If the identified file groups only account for a small portion of the project's overall maintenance cost, then they do not deserve attention. Furthermore, we examine whether the maintenance cost associated with the *ATDs* is proportionally higher than the number of files contained in the *ATDs*. If the identified file groups contain a large number of source files, it is not surprising that they account for a large amount of maintenance costs. In either case, we cannot claim that the *ATDs* identified by the proposed approach worthy of attention.
- **RQ1-2: Will the file groups identified in *ATDs* based on project history keep incurring significant maintenance costs in the future?** In this part, we investigate whether the *ATDs*—identified using the proposed approach based on a project's revision history from release 1 to release $r-1$—will keep incurring significant maintenance costs between release $r-1$ and release r. If the *ATDs* identified based on history stop incurring substantial maintenance costs in the future, this indicates that the *ATDs* identified by our approach do not deserve attention.
- **RQ1-3: Is our approach simply identifying large files as *ATDs*?** Prior research has shown that file size usually correlates with error-proneness and churn. Therefore, large files tend to be identified as *ATDs*. Here, we want to investigate whether our approach simply identifies groups of large source files. If so, it indicates that we can (much more easily) identify *ATDs* using LoC. To answer this question, we analyzed the size, in terms of the Lines of Code (LoC), of files involved in debts in each project, and investigate whether the identified debts are composed of files of varying sizes. If so, it indicates that our approach is not simply identifying large files.

- **RQ2: Can a $DebtModel$ accurately predict the future cost of an ATD?** A $DebtModel$ is a regression model that describes the trajectory of the maintenance costs associated with an ATD. An accurate $DebtModel$ should not only characterize past costs, but it should also be able to predict future costs. If the cost of a debt in release n,

estimated using the $DebtModel$ calculated from release 1 to n-1, is close to (e.g. within 10% deviation from) the actual cost in release n, we can claim that the $DebtModel$ is accurate. If so then architects can confidently use the $DebtModel$ to predict the interest cost of each debt in the next release. In particular, we plan to investigate how the choice of the R^2 threshold influences the accuracy of prediction.

- **RQ3: Are compound *ATDs* common in software projects, and do they form cost-effective refactoring candidates?** This RQ investigate two aspects of the compound *ATDs* after the aggregation step. First, we evaluate whether the merging process, based on *Transitive Anchors* and *Compound Anchors*, is widely applicable to debts across projects. If the *ATDs* are mostly independent from each other, there will be few meaningful merging opportunities. And it indicates that the compound *ATDs* formed by multiple atomic patterns are rare. Second, if merging is widely applicable, we will examine whether the compound debts after merging are cost-effective for architects to inspect as refactoring candidates. A compound debt tends to aggregate more files and thus may become harder to review. Thus, we will examine two important characteristic of each compound debt: the percentage of files included in it, and the percentage of maintenance cost associated with it. A debt is more cost-effective to refactor if it contains a small portion of the system's files, but these files account for a high proportion of the system's maintenance costs.

B. Study Subjects

We chose 18 Apache open source projects as our evaluation subjects. These projects differ in scale, application domain, length of history, and many other project characteristics. They are: Camel—a integration framework based on Enterprise Integration Patterns; Cassandra—a distributed DBMS; CXF—a Web services framework; Hadoop—a framework for reliable, scalable, distributed computing; HBase—the Hadoop distributed, scalable, big data store; PDFBox—a library for working with PDF documents; and Wicket—a component-based web application framework. OpenJPA—an object-relational mapping solution for simplifying storing Java objects in databases. HIVE—a data warehouse software project built on top of Apache Hadoop for providing data query and analysis. Avro—a row-oriented remote procedure call and data serialization framework Mesos—an open-source project to manage computer clusters. Httpd—an open-source cross-platform web server software for Apache. Kudu—a column-oriented data store of the Apache Hadoop ecosystem. Mahout—it produces free implementations of distributed or otherwise scalable machine learning algorithms Chemistry—it provides Content Management Interoperability Services in different programming languages. Jena—a Semantic Web framework for Java Ambari—a project for provisioning, managing, and monitoring Apache Hadoop clusters. Finally, allura is an open source implementation of a software forge, a web site that manages source code repositories, bug reports, discussions, wiki pages, blogs, and more for any number of individual projects.

TABLE I: Study Subjects

Subject(L)	History	#R	#Cmt	#Iss	# Files
Camel(J)	7/2008 to 7/2014 (72)	12	14563	2790	1838 to 9866
Cassandra(J)	9/2009 to 11/2014 (62)	10	14673	4731	311 to 1337
CXF(J)	12.2007 to 5-2014 (77)	13	8937	3854	2861 to 5509
Hadoop(J)	8/2009 to 8/2014 (60)	9	8253	5443	1307 to 5488
HBase(J)	12/2009 to 9/2014 (53)	9	6718	6280	560 to 2055
PDFBox(J)	8/2009 to 9/2014 (62)	12	2005	1857	447 to 791
Wicket(J)	6/2007 to 1/2005 (92)	15	8309	3557	1879 to 3081
OpenJPA(J)	8/2007 to 6/2018 (130)	17	4265	1779	1266 to 4487
HIVE(J)	10/2010 to 2/2016 (76)	14	7309	11768	979 to 4424
Avro(J)	5/2010 to 5/2016 (72)	15	1288	1066	156 to 506
Mesos(c++)	5/2014 to 1/2019 (56)	10	14368	4589	393 to 1294
Httpd(c)	6/2005 to 1/2019 (163)	28	17239	8091	327 to 462
Kudu(c++)	2/2016 to 10/2018 (32)	11	3918	1423	953 to 1223
Mahout(J)	5/2010 to 4/2017 (83)	11	3046	801	455 to 1217
Chemistry(J)	4/2011 to 4/2017 (72)	11	1786	589	652 to 1019
Jena(J)	6/2012 to 9/2018 (75)	18	7488	890	2552 to 4214
Ambari(py)	5/2014 to 12/2019 (67)	19	15,549	15,777	682 to 1693
Allura(py)	8/2013 to 10/2019 (74)	14	7498	542	491 to 542

A summary of these projects is given in Table I. The first column is the project name and the main implementing programming language. In particular, 13 projects are implemented in Java, while the other 6 projects are implemented in C/C++ or Python. The second column is the start to end time and the total number of months (in parentheses) for each project. The third column "#R" shows the number of releases selected per project. We selected releases to ensure that the time interval between two releases is approximately 6 months. The column "#Cmt" is the number of commits happened over the selected history. The column "#Iss" is the number of bug reports, downloaded from the project's bug-tracking system. The last column shows the size range, measured as the number of files in the first and the last selected release.

The rationale behind the selection of the evaluation subjects is as follows. First, these projects have diverse characteristics, in terms of age (from 32 to 163 months), domains, scale, and programming language (Java, C/C++, and Python). The diversity of the projects ensures that our approach is generally applicable to projects of different domains, age, scale, and programming language. Second, since we aim at *ATDs* that incur long-term maintenance consequences, the identification and quantification approach relies on sufficient evolutionary history in software projects. The selected projects all have at least 9 releases to provide sufficient data. Projects that are new, without sufficient releasing history, are not appropriate for this study. Third, we select projects with high quality maintenance revisions and issue tracking data. This allows us to keep track and estimate the maintenance costs using the error-fixing churn, mined from the project repository.

VI. Evaluation Results

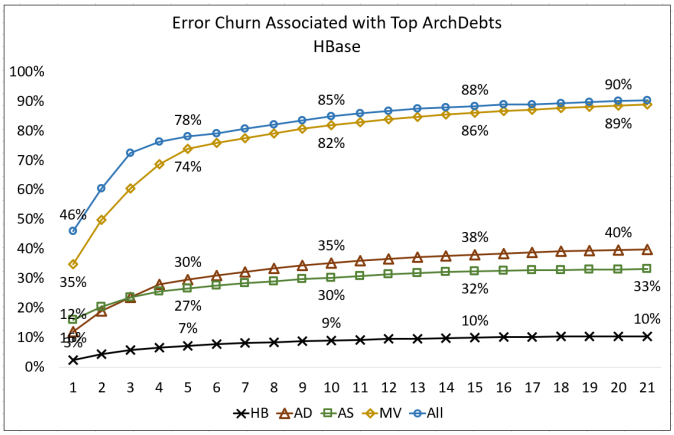
A. *Significance of* ATDs

As discussed in Section V, we evaluate whether the proposed approach can identify significant *ATDs* that deserve attention in three sub-questions:

RQ1-1: Do the file groups identified in *ATDs* generate more maintenance costs than what one would expect given

their sizes? We will report the percentage of maintenance costs associated with the identified *ATDs*, as well as the percentage of files that are contained in the *ATDs*. If the *ATDs* account for a larger percentage of maintenance cost compared to the size, it implies that they are causing extra cost in relative to their size, and thus deserves attention.

Fig. 11: Churn associated with *ATDs* in HBase



We first use HBase as an example to illustrate our detailed observations, and then we will summarize the key information of the debts identified from different projects. Figure 11 shows the percentage of maintenance cost (approximated by the bug-fixing churn) associated with source files that are involved in *ATDs*. The four trendlines represent the maintenance cost of the four *ATD* patterns (from top to bottom), namely *Modularity Violations (MV)*, *Anchor Dominant (AD)*, *Anchor Submissive (AS)*, and *Hub (HB)*. The x-axis is the rank of each *ATD* based on the associated maintenance cost. The y-axis indicates the accumulative maintenance cost associated with the top *x ATDs*. We can make the following observations based on HBase:

1) **The top 21 *ATDs* account for a significant portion (89%) of maintenance cost in HBase.** Source files involved in *ATDs* tend to continuously accumulate a significant amount of cost in the project history. Therefore, to avoid paying excessively high maintenance costs, architects and developers should try to pay-off the "debts" by eliminating the underlying architectural design problems through refactoring. For instance, developers could encapsulate the "shared secrets" among those files identified as having *Modularity Violations* to separate the most change-prone files from the less volatile parts of the system [30]. Note that, as we look at more *ATDs* in HBase, the maintenance cost associated with them never reaches 100%. The reason is that not all files in the project are associated with debts.
2) **The top *five Architectural Debts* account for a large portion of maintenance cost**. For example, the maintenance cost of the top 5 *MV* debts take 74% of all the error fixing churn. And the next 16 debts only increase this to 89% of maintenance cost. Note that the top 5 *MV* debts only contain 34% of files in a project. Therefore, this small group of files deserves attention. Similar observation can be made for the other three types of debts. The four trendlines flatten out after the top 5 debts. The implication is that architects should prioritize the top *five* debts to capture the majority of the maintenance interests accumulated with debts.
3) ***Modularity Violations* are the most common and expensive type of debts.** *Hub* is the least common debt pattern. This is because *Hub* has the most complicated structure in the four debt patterns: the anchor file is both structurally and evolutionarily coupled with each member file in both directions (from anchor to member and vice versa). In comparison, *AD* debt and *AS* debts account for comparable amounts of maintenance cost: up to about 33% and 40% of the error-fixing churn. Note also that the sum of the maintenance cost associated with the four types of debts is more than 100%. This is because some source files are involved in multiple debts, thus the cost associated with these files are counted multiple times.

We have observed similar patterns in most of the 18 projects studied. Table II shows the summary of the top *five ATDs* in each project. As we can see, the top five *ATDs* in the projects on average account for 50% of error-fixing churn, while they only contain, on average, 28% files in a project. In other words, the maintenance cost associated with the *ATDs* almost double compared to their size (in terms of number of files). This indicates that the identified *ATDs* cause non-trivial extra cost in relative to their size, thus they deserve attention from the practitioners. This, as discussed earlier, indicates that architects should prioritize the top few debts for refactoring. *Modularity Violations* are the most common and expensive type of debts compared to the other three types in all the projects. In some cases projects have only a few *ATDs* identified; this indicates that the error-prone files in these projects tend to be structurally and/or evolutionarily decoupled and thus do not form significant debt patterns. In section VII-C, we will explain more about this phenomenon.

Furthermore, we also compare the percentage of LoC (%LoC) vs. the percentage of churn (%Churn) associated with the identified *ATDs*. The goal is to test whether the identified *ATDs* account for a larger percentage of Churn than would be expected given their LoC. The results are shown in Table III. In column 2 and column 3, we show the %LoC and the %Churn associated with all the identified debts. In column 4, we calculate the difference—%Churn minus %LoC; the larger the difference, the greater the extent to which the identified *ATDs* account for a disproportionately large portion of the churn. As we can observe from this table, the %Churn is *always* greater (typically more than 10% and up to 66% greater for the studied projects) than the %LoC of the associated debts. However, the difference is not as large as compared to %files. The reason is that, to a great extent, the total LoC in a file is the result of the accumulation of churn over time. This indicates that the identified *ATDs* account for a relatively larger amount of churn, comparing to the LoC they contain. This is consistent with the findings when comparing the percentage of

TABLE II: Significance of Debts:
The Percentage of Error-fixing Churn (Ch%) and Number of Files (Fls.)

Subject	#Debts (Ch%)	Top 5 Debts														
		All 4 Types			Modularity Vio.			Anchor Sub.			Anchor Dom.			Hub		
		Fls.	Ch%	Diff	Fls.	Ch%	Diff	Fls.	Ch%	Diff	Fls.	Ch%	Diff	Fls.	Ch%	Diff
Camel	512 (74%)	1398(19%)	32%	13%	1363(19%)	30%	11%	32(0.4%)	2%	1.6%	42(0.6%)	5%	4.4%	26(0.4%)	3%	2.6%
Cassandra	124 (91%)	1021(59%)	82%	23%	1012(58%)	81%	23%	57(3%)	5%	2%	31(2%)	22%	20%	20(1%)	10%	9%
CXF	183 (64%)	490(16%)	32%	16%	439(14%)	29%	15%	40(1%)	2%	1%	31(1%)	6%	5%	18(1%)	2%	1%
Hadoop	81 (58%)	261(17%)	40%	23%	218(15%)	35%	20%	90(6%)	17%	11%	28(2%)	13%	11%	17(1%)	6%	5%
HBase	282 (95%)	980(41%)	78%	37%	796(34%)	74%	40%	495(21%)	26%	5%	77(3%)	29%	26%	18(1%)	7%	6%
PDFBox	29 (62%)	199(28%)	57%	29%	159(23%)	48%	25%	84(12%)	34%	22%	25(4%)	14%	10%	11(2%)	8%	6%
Wicket	100 (50%)	385(14%)	29%	15%	352(13%)	28%	15%	66 (3%)	5%	2%	21(1%)	6%	5%	12 (0.4%)	4%	3.6%
HIVE	272 (75%)	1403(35%)	62%	27%	1178(30%)	57%	27%	575(15%)	32%	17%	39(1%)	22%	21%	83(2%)	6%	4%
Avro	29 (74%)	171(39%)	63%	24%	164(38%)	61%	23%	22(0.5%)	6%	5.5%	8 (2%)	4%	2%	4(1%)	3%	2%
Mesos	9 (54%)	70(29%)	54%	25%	70(29%)	54%	25%	15(6%)	9%	3%	3(1%)	6%	5%	2(6%)	9%	3%
OpenJPA	91 (61%)	394(15%)	35%	20%	352 (14%)	35%	21%	-	-	-	27(1%)	8%	7%	5(0.2%)	2%	1.8%
Httpd	17 (92%)	78 (67%)	91%	24%	77(66%)	91%	25%	-	-	-	12(1%)	48%	47%	11(9%)	31%	22%
Kudu	7 (31%)	55(18%)	31%	13%	46(15%)	28%	13%	10(3%)	5%	2%	-	-	-	-	-	-
Mahout	6 (23%)	185(14%)	23%	9%	185(14%)	23%	9%	-	-	-	-	-	-	-	-	-
Chemistry	2 (18%)	68 (14%)	18%	4%	68 (14%)	18%	4%	-	-	-	-	-	-	-	-	-
Jena	23 (29%)	236(14%)	22%	8%	236(14%)	22%	8%	7(0.4%)	1%	0.6% -	-	-	-	-	-	-
Ambari	29 (56%)	188(35%)	49%	14%	188(35%)	49%	14%	5(1%)	3%	2%	12(2%)	7%	5%	-	-	-
Allura	39 (89%)	96(53%)	84%	31%	92 (51%)	83%	32%	19(11%)	21%	10%	24(13%)	50%	37%	10(6%)	19%	13%
Min.	18%	14%	23%	4%	13%	28%	4%	0.5%	2%	1%	0.6%	4%	2%	0.4%	3%	1%
Max.	95%	67%	91%	37%	66%	81%	40%	21%	34%	22%	4%	48%	47%	9%	31%	22%
Avg.	61%	28%	50%	20%	28%	49%	19%	6%	11%	6.1%	6%	12%	14.7%	4%	6%	6.1%

files vs. the percentage of churn. Columns 5 to 7 show similar information, but for the top five *ATDs* in each project. The same observations and conclusions hold here.

TABLE III: %LoC VS. %Churn in *ATDs*

Proj.	All Debts			Top 5 Debts		
	LoC%	Churn%	Diff	LoC%	Churn%	Diff
Camel	52%	74%	22%	21%	32%	11%
Cassandra	85%	91%	6%	72%	82%	10%
CXF	44%	64%	20%	18%	32%	14%
Hadoop	11%	58%	47%	6%	40%	34%
HBase	88%	95%	7%	67%	78%	11%
PDFBox	39%	62%	23%	33%	57%	24%
Wicket	39%	50%	11%	19%	29%	10%
HIVE	63%	75%	12%	49%	62%	13%
Avro	67%	74%	7%	53%	63%	10%
Mesos	18%	54%	36%	17%	54%	37%
OpenJPA	53%	61%	8%	28%	35%	7%
Httpd	26%	92%	66%	24%	91%	67%
Kudu	15%	31%	16%	14%	31%	17%
Mahout	17%	23%	6%	17%	23%	6%
Chemistry	11%	18%	7%	11%	18%	7%
Jena	11%	29%	18%	8%	22%	14%
Ambari	12%	56%	44%	11%	49%	38%
Allura	36%	89%	53%	33%	84%	51%
Min.	11%	18%	6%	6%	18%	6%
Max.	88%	95%	66%	72%	91%	67%
Avg.	38%	61%	23%	28%	49%	21%

RQ1-2: Will the file groups identified in *ATDs* based on a project history keep incurring significant maintenance costs in the future? Here, we report on two measures: 1) the percentage of bug-fixing files and churn between release $r-1$ and release r that are directly identified as *ATDs* based on release 1 to release $r-1$; and 2) the total percentage of bug-fixing files and churn between release $r-1$ and release r that **are directly identified as or "growing out" of** the *ATDs* based on release 1 to release $r-1$ in each project. Here, "growing out" means more files are involved in the identified *ATDs* through the four debt patterns.

As shown in Table IV, on average, a significant portion (63%) of the future bug-fixing files (between release r-1 and release r) are from *ATDs* in the project history (between release 1 to release r-1). Similarly, 66% of the future bug-fixing churn, i.e. the maintenance costs, are spent on *ATDs* in history. Furthermore, a higher portion of future bug-fixing files (72%) and churn (77%) are directly from or "growing from" *ATDs* in the project history. Therefore, we can conclude that the *ATDs* identified based on history will keep incurring significant maintenance cost (interest) in the future.

TABLE IV: Percentage of Bug-fixing Files/Churn between Release r-1 to r that are Directly in and Growing from *ATDs* Identified based on Release 1 to r

Subject	Direct in *ATDs*		Plus Growing from *ATDs*	
	%Files	%Churn	%Files	%Churn
Camel	61%	74%	70%	76%
Cassandra	85%	90%	92%	95%
CXF	55%	57%	73%	72%
Hadoop	44%	46%	63%	68%
HBase	95%	99%	98%	100%
PDFBox	38%	62%	72%	89%
Wicket	35%	42%	42%	44%
OpenJPA	61%	38%	69%	79%
HIVE	72%	87%	80%	91%
Avro	78%	53%	85%	85%
Mesos	35%	18%	53%	48%
Httpd	87%	99%	87%	99%
Kudu	77%	92%	87%	93%
Mahout	40%	54%	40%	54%
Chemistry	59%	63%	66%	66%
Jena	54%	33%	56%	40%
Ambari	61%	81%	67%	86%
Allura	96%	98%	100%	100%
Avg	63%	66%	72%	77%
Min	35%	18%	40%	40%
Max	96%	99%	100%	100%

RQ1-3: Is our approach simply identifying files with large LoC as *ATDs*? Are all files with large LoC identified as *ATDs* by our approach? Here, we want to know if our approach simply identifies groups of source files with large LoC; and whether all the large files are identified by our

approach as debts. If so, we can much more easily identify *ATDs* by simply ranking source files using LoC, instead of the approach proposed in this paper.

We analyze the LoC of files involved in *ATDs* in each project, and found that the identified *ATDs* are composed of files of varying sizes. For example, Figure 12 shows the distribution of the LoC of files involved in the top five debts in HIVE. The x-axis shows the size range of files. For instance, "$\leq$10%" stands for files that are ranked in the top 10% percentile based on the LoC. The y-axis stands for the percentage of debt files that belong to the range indicated in the x-axis. As we can see, 18% of the debt files are ranked in the top 10% among all the source files in a project based on the LoC. We observe from Figure 12 that only from 13% to 18% of files involved in debts are very large files (i.e. reside in the top 10%, top 20% amd top 30% bins) in HIVE. This is consistent with prior study that large files tend to be problematic. But files involved in *ATDs* also appear in all size ranges—more than half (51%) of the *ATDs* files are ranked outside of the top 30%.

Fig. 12: Top 5 Debts File LoC Distribution (HIVE)

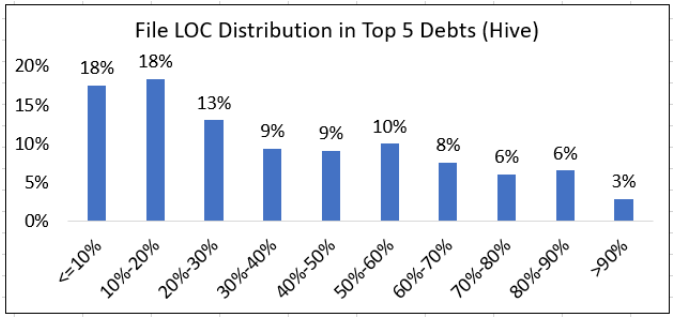


Table V summarizes the LoC distribution of the *ATDs* files from each project. As we can see, except for Allura, a non-trivial (24%) to a significant (56%) portion of the *ATDs* files in each project are ranked below the top 30% based on the LoC. In many projects, a non-trivial (up to 40%) of debts are led by source files that are not ranked in the top 30% based on their LoC. In particular, based on our investigation of the identified debts, a source file with a small number of LoC could also serve as the central file of a debt and leads to non-trivial amount of maintenance costs. For example, in Httpd, source file *os.os2.os_h*, whose LoC ranked in the 94% among other files in the project, is the anchor of an *Anchor Dominant* debt with 4 other member files. This debt survived 19 releases in Httpd, and account for 16% of the maintenance costs in Httpd. Therefore, we can conclude that our approach is not simply identifying large files.

We also investigate whether files with large LoC in a project are all identified as *ATDs* by our approach. To answer this question, we performed additional data analysis to show that not all of the large files—i.e. files ranked in the top 10%, 20%, and 30% percentile of LoC respectively—are identified as debts by our approach. This result is shown in Table VI. As we can see, in most projects only a small portion (as low as 8%) of the top 10% largest files are identified as debts. Similar observations hold for the top 20% and the top 30% largest files. That is, the majority of the large files in every project are *not* identified by our approach as debt. Therefore, LoC is neither a sufficient nor a necessary condition to identify debts.

In summary, we observed that 1) our approach can identify expensive *ATDs* that generate, on average, 2X more maintenance costs than what one would expect given their sizes in a project's evolutionary history; 2) the *ATDs* identified based on a project's history will grow and keep incurring significant maintenance costs (on average 77%) in future release; 3) the *ATDs* identified by our approach do not simply contain large files—non trivial (24%) to significant (56%) portion of the *ATD* files have less than the top 30% LoC. Only a small portion of files with large LoC are identified as *ATDs*. This indicates that LoC is neither a sufficient nor a necessary condition for our approach to identify debts. Thus, we conclude that our approach identifies *ATDs*, worthy of attention.

B. Debt Regression Models for Cost Prediction

RQ2: Can we use a $DebtModel$ to predict the future cost of an ATD? Specifically, we aim to investigate whether the regression models can accurately predict the cost of an *ATD* in release n based on the model calculated from the previous $n-1$ releases. To do this we first define an accuracy measure—*Prediction Deviation (PD)*—as follows:

$$PD = \frac{|Actual\_Cost\_R_n - Predicted\_Cost\_R_n|}{Actual\_Cost\_R_n} \quad (18)$$

Prediction Deviation (PD) measures how far the predicted cost $Predicted\_Cost\_R_n$ deviates from the actual cost $Actual\_Cost\_R_n$ in release n. Thus, *PD* is a percentage value. The lowest value is 0%, which means that the prediction is completely accurate. The larger the *PD*, the less accurate is the prediction. We calculate $Predicted\_Cost\_R_n$—the predicted cost at release n based on the model built from previous $n-1$ releases.

First, we evaluate the accuracy of prediction under different values of the R^2 threshold. If the majority of debts can be predicted with a small amount of *PD*, it indicates that the prediction is accurate. Thus, we investigate the percentage of *ATDs* whose models have $\leq 10\%$ *PD* (because no prediction model can be 100% accurate). We vary the R^2 threshold from 0.6 to 0.9 to see whether a higher percentage of debts can be predicted with $\leq$ 10% PD. The evaluation results are listed in Table VII: **1) prediction accuracy increases with the increase of R^2 threshold in most (14 out of the 18) projects; and 2) in most projects (except PDFBox and Kudu), we can predict the costs of the majority (between 64% to 100%) of debts with less than 10% *PD*.** Take Camel as an example, only 65% of debts can be predicted with less than 10% drift when R^2 threshold is 0.6. As we increase the threshold, the percentage of debts with $\leq$ 10% PD gradually increases to 68%, 80%, and 94%. In PDFBox, however, the percentage drops from 52% to 51% and to 46%, when the threshold increases to 0.8 and to 0.9 respectively. Similarly, in Kudu, the percentage drops from 60% to 40% when increasing the threshold from

TABLE V: LoC Distribution of *ATDs* Files

Proj	LoC Bins, Each Bin is 10%											
	Top 10%	(10%,20%]	(20%,30%]	Top 30%	(30%,40%]	(40%,50%]	(50%,60%]	(60%,70%]	(70%,80%]	(80%,90%]	(90%,100%]	(30%,100%]
Camel	22%	12%	10%	44%	11%	11%	7%	6%	7%	6%	8%	56%
Cassandra	18%	15%	12%	45%	11%	10%	8%	7%	7%	6%	6%	55%
CXF	32%	20%	12%	64%	11%	8%	5%	4%	2%	2%	4%	36%
Hadoop	41%	22%	13%	76%	6%	3%	5%	5%	2%	2%	1%	24%
HBase	19%	18%	15%	52%	12%	10%	9%	5%	4%	5%	3%	48%
PDFBox	32%	18%	13%	63%	12%	6%	5%	5%	4%	3%	2%	37%
Wicket	28%	17%	13%	58%	7%	12%	8%	6%	3%	3%	3%	42%
OpenJPA	39%	15%	12%	66%	8%	5%	5%	4%	4%	3%	5%	34%
HIVE	18%	18%	13%	49%	9%	9%	10%	8%	6%	6%	3%	51%
Avro	24%	14%	14%	52%	8%	9%	10%	6%	6%	5%	4%	48%
Mesos	33%	20%	9%	62%	6%	10%	5%	8%	6%	3%	0%	38%
Httpd	11%	20%	14%	45%	16%	5%	4%	7%	5%	14%	4%	55%
Kudu	35%	16%	14%	65%	8%	5%	3%	5%	0%	11%	3%	35%
Mahout	18%	13%	15%	46%	12%	8%	7%	8%	7%	7%	5%	54%
Chemistry	24%	22%	10%	56%	9%	8%	2%	16%	2%	5%	2%	44%
Jena	26%	20%	15%	61%	9%	8%	6%	7%	3%	3%	3%	39%
Ambari	14%	31%	12%	57%	14%	10%	8%	2%	2%	2%	5%	43%
Allura	100%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%
Min	11%	0%	0%	44%	0%	0%	0%	0%	0%	0%	0%	0%
Max	100%	31%	15%	100%	16%	12%	10%	16%	7%	14%	8%	56%
Avg	30%	17%	12%	59%	9%	8%	6%	6%	4%	5%	3%	41%

TABLE VI: % of Top LoC Files Identified in *ATDs*

Proj.	Top 10%	Top 20%	Top 30%
Camel	28%	21%	18%
Cassandra	78%	72%	65%
CXF	23%	19%	15%
Hadoop	8%	6%	5%
Hbase	70%	69%	65%
PDFBox	52%	40%	34%
Wicket	23%	19%	16%
HIVE	46%	46%	42%
Avro	74%	58%	54%
Mesos	17%	14%	11%
OpenJPA	31%	21%	17%
Httpd	19%	26%	26%
Mahout	21%	18%	18%
Chemistry	14%	13%	11%
Jena	9%	8%	7%
Ambari	8%	13%	11%
Allura	20%	20%	20%

0.8 to 0.9. We conjecture that there are exogenous factors that cause fluctuations in debts in these projects. Increasing the R^2 threshold to 0.9 in these cases will lead to over-fitting that compromises the prediction accuracy. For Mesos and Chemistry, the R^2 does not impact the accuracy. Note that these two projects only contain 9 and 2 debts, therefore the trends in these two projects are not reliable.

Next, for the purpose of comparison, we also examine a simple baseline prediction model, one which architects could easily use to predict the cost without calculating a regression model. In this simple model the cost of a debt in release n is estimated based on the cost of the two prior releases: $n-2$ and $n-1$. Specifically, $Baseline\_Prediction\_R_n = Cost\_R_{n-1} + \delta$, where $\delta = Cost\_R_{n-1} - Cost\_R_{n-2}$, which is the increment between the most recent two releases. We want to investigate whether the regression models can predict the future costs of debts better than this simple model. Based on the data shown in Table VII, we make the comparison by using an R^2 threshold of 0.9.

Table VIII shows the results. The first column is the project name. The other columns show the percentage of debts whose costs can be predicted with up to 30% *PD*, using both the regression models (sub-column "RM") and the baseline (sub-column "BL"). We focus on *PD* up to 30% because 1) a majority of the predictions have less than 30% drift; 2) there will always be a portion of debt that cannot be predicted due to accidental reasons, such as a change in project direction; and 3) a higher deviation is not useful for architects. The sub-column "Win" shows the percentage of debts in which the regression models outperform the baseline. We draw the following observations from Table VIII:

TABLE VII: Impact of R^2 Threshold on Prediction Drift

Subject	% of Debts with $\leq$ 10% Prediction Drift				Trend
	$R^2 \geq 0.6$	$R^2 \geq 0.7$	$R^2 \geq 0.8$	$R^2 \geq 0.9$	
Camel	65%	68%	80%	94%	up
Cassandra	83%	84%	84%	87%	up
CXF	69%	69%	69%	72%	up
Hadoop	81%	82%	82%	89%	up
HBase	63%	63%	64%	64%	up
PDFBox	52%	52%	51%	46%	down
Wicket	83%	83%	84%	88%	up
OpenJPA	73%	77%	89%	100%	up
HIVE	66%	66%	67%	70%	up
Avro	88%	88%	90%	96%	up
Mesos	85%	85%	85%	85%	-
Httpd	73%	76%	76%	95%	up
Kudu	50%	50%	60%	40%	down(*)
Mahout	55%	55%	64%	82%	up
Chemistry	100%	100%	100%	100%	-
Jena	73%	76%	82%	82%	up
Ambari	64%	64%	64%	74%	up
Allura	58%	59%	63%	96%	up
Min	50%	50%	51%	40%	
Max	100%	100%	100%	100%	
Avg	71%	72%	75%	81%	

- **Regression models can predict the future costs of the majority (72% to 100%) of debts with less than 20% deviation from the actual costs.** The regression models outperform the baseline model in up to 60% of debts in the 18 projects.
- **Except for PDFBox and Kudu, the regression models can predict the the future costs of the majority debts with less than 10% *Prediction Drift*.** Regression models provide poorer predictions of the cost, as compared to the baseline model, in only 4 out of the 18 projects, CXF, Wicket, Httpd, and Ambari (highlighted with blue

TABLE VIII: Regression Model Cost Prediction Deviation ($R^2 \geq 0.9$)

Subject	% of Debts with PD $\leq$ X								
	PD $\leq \pm 10\%$			PD $\leq \pm 20\%$			PD $\leq \pm 30\%$		
	RM	BL	Win	RM	BL	Win	RM	BL	Win
Camel	94%	95%	0%	99%	96%	3%	99%	97%	2%
Cassandra	87%	81%	6%	94%	86%	8%	97%	88%	9%
CXF	72%	74%	-2%	88%	81%	7%	93%	84%	9%
Hadoop	89%	67%	22%	94%	83%	11%	95%	85%	10%
HBase	64%	47%	17%	79%	66%	13%	87%	73%	14%
PDFBox	46%	39%	7%	72%	39%	33%	82%	43%	39%
Wicket	88%	93%	-5%	98%	96%	1%	98%	99%	-1%
OpenJPA	100%	97%	2%	100%	98%	2%	100%	98%	2%
HIVE	70%	67%	3%	88%	72%	16%	94%	75%	19%
Avro	96%	70%	26%	100%	86%	14%	100%	90%	10%
Mesos	85%	54%	31%	100%	85%	15%	100%	92%	8%
Httpd	95%	100%	-5%	100%	100%	0%	100%	100%	0%
Kudu	40%	40%	0%	100%	40%	60%	100%	40%	60%
Mahout	82%	45%	36%	100%	100%	0%	100%	100%	0%
Chemistry	100%	100%	0%	100%	100%	0%	100%	100%	0%
Jena	82%	33%	48%	88%	55%	33%	91%	70%	21%
Ambari	74%	87%	-13%	99%	94%	4%	100%	97%	3%
Allura	96%	86%	10%	100%	96%	4%	100%	98%	2%
Min	40%	33%	-13%	72%	39%	0%	82%	40%	-1%
Max	100%	100%	48%	100%	100%	60%	100%	100%	60%
Avg	81%	71%	10%	94%	82%	13%	96%	85%	12%

background). On average, however, the regression models outperform the baseline model by 10%.

In summary: 1) The regression models are effective in predicting the future costs of the majority (72% to 100%) of the debts with less than 20% *Prediction Deviation*. They outperform the baseline model in most (up to 60%) debts; and 2) As we increase the R^2 threshold, the regression models can provide more accurate predictions.

C. *Compound* ATDs

RQ3: Can we aggregate related *ATDs* into compound debts that are cost-effective refactoring candidates? As discussed in Section IV-E, we merge the identified *ATDs* in two phases based on 1) the *Transitive Anchors* relationship and 2) the *Compound Anchors* relationship. In this RQ, we first investigate whether such a merge is broadly applicable to debts in different projects. If not, it indicates that the debts are mostly independent and should be treated separately. If these merges do apply then architects need to examine the more complicated connections among these larger debts to developing effective refactoring solutions. Next, we investigate whether the compound debts are *cost-effective* refactoring candidates, since each compound debt is composed of multiple debts and thus contains a larger number of source files.

Table IX shows how debts can be merged based on *Transitive Anchors*. In column "#Debts", for each debt type, we listed the total number of debts of this type and the percentage (in parentheses) of debts that can be merged together due to *Transitive Anchors*. In column "Merge", we list the number of debts that can be merged and the number of compound debts resulting from the merge. For example, in Camel, there are a total of 100 *AD* debts, among these, 6 (6%) debts have *Transitive Anchors*, and they can be merged into 3 compound debts. We make the following observations from Table IX:

- ***Transitive Anchors* are common in *AD*, *AS*, and *HB* debts.** In projects where *Transitive Anchors* are not applicable, there are usually only a small number of debts, thus the opportunities to merge are few. For example, as highlighted in blue, there are just 1 to 12 *AD* debts in the seven projects where merge is not applicable.
- ***Transitive Anchors* are prevalent in *MV* debts.** The majority (62% to 100%) of *MV* debts share *Transitive Anchors*. In addition, the *MV* debts can be merged into just a few (from 1 to 4) compound debts based on *Transitive Anchors*. The implication is that architects need to only review these few compound *MV* debts when exploring refactoring opportunities. However, the downside is that each cluster contains more files.

Similarly, Table X shows the application of the *Compound Anchors* merge. We use an overlap threshold of 0.5 here. In Section VII-D, we will discuss how this threshold impacts the merge results. As discussed in Section IV-E, we perform the *Compound Anchors* merge as a second phase, based on the results of the *Transitive Anchors* merge. However, as shown above, the *Transitive Anchors* merge is prevalent for *MV* debts. The majority of *MV* debts can be merged into just a few compound debts. Thus the *Compound Anchors* merge is not commonly applicable due to the small number of debts resulting from the first phase. Therefore, for *MV* debts, we apply the *Compound Anchors* merge directly on the original debts. This will also help us to avoid very large compound debts that aggregate all the original debts. We now make similar observations:

- **The *Compound Anchors* merge is generally applicable to *AD*, *AS*, and *HB* debts.** In particular, when there is a large number of debts, the merge is likely to be prevalent. For example, the merge is applicable to 82% of the 97 *AD* debts in Camel. In contrast, the merge is less likely to occur when there is just a small number of debts, as highlighted in blue cells.
- The ***Compound Anchors* merge is also prevalent for *MV* debts.** More specifically, 74% (in Hadoop) to 100% (in Mesos, Httpd, and Chemistry) of the *MV* debts can be merged into just 1 to 12 compound debts.

As shown in the examples of Figure 9 and Figure 10, merging debts based on the two relationships helps to capture the complicated structural patterns among debts. However, as mentioned above, the downside is that each compound debt will contain more files and thus may be cumbersome to review.

Therefore, we need to evaluate whether the compound debts are *cost-effective* refactoring candidates. We capture four kinds of information for this purpose, as shown in TableXI. First, the original number of debts and the number of compound debts in column "#Debts". This shows the reduction in the number of refactoring candidates for the architects to review as the result of merging. Second, the average size of the merged debts, in terms of the percentage of files, in column "S.%". If the size of the merged debts is too large, they are difficult to review. Third, the average amount of maintenance cost associated with the merged debts, in terms of the percentage of bug-fixing churn, in column "Ch.%". This measures how significant each compound debt is. And finally, the ratio in column "R.", calculated as "Ch.%" divided by "S.%". This measures the potential cost

TABLE IX: Prevalence of *Transitive Anchors* Merge

Subject	AD		AS		HB		MV	
	#Debts	Merge	#Debts	Merge	#Debts	Merge	#Debts	Merge
Camel	100 (6%)	6→3	9 (0%)	–	46 (4%)	2→1	357 (95%)	338→4
Cassandra	35 (37%)	13→2	11 (18%)	2→1	21 (19%)	4→2	57 (100%)	57→1
CXF	27 (7%)	2→1	3 (0%)	–	3 (0%)	–	150 (96%)	144→2
Hadoop	16 (44%)	7→3	11 (36%)	4→2	7 (57%)	4→2	47 (62%)	29→4
HBase	107 (21%)	22→7	34 (88%)	30→1	30 (0%)	–	111 (87%)	97→1
PDFBox	6 (33%)	2→1	3 (0%)	–	3 (67%)	2→1	17 (100%)	17→1
Wicket	12 (0%)	–	2 (0%)	–	9 (22%)	2→1	77 (71%)	55→2
HIVE	113 (7%)	8→4	9 (89%)	8→1	14 (79%)	11→3	136 (80%)	109→3
Avro	2 (0%)	–	1 (0%)	–	1 (0%)	–	25 (96%)	24→2
Mesos	1 (0%)	-	2 (0%)	-	1 (0%)	-	5 (80%)	4→1
OpenJPA	5 (0%)	–	–	–	2 (0%)	–	84 (89%)	75→3
Httpd	3 (0%)	–	–	–	4 (0%)	–	10 (90%)	9→1
Kudu	-	-	1 (0%)	-	-	-	6 (83%)	5→1
Mahout	-	-	-	-	-	-	6 (83%)	5→2
Chemistry	-	-	-	-	-	-	2 (100%)	2→1
Jena	1 (0%)	-	-	-	-		22 (86%)	19→2
Ambari	4 (0%)	–	2 (0%)	–	–	–	23 (96%)	23→1
Allura	10 (20%)	2→1	4 (0%)	–	4 (50%)	2→1	21 (86%)	18→1

TABLE X: Prevalence of *Compound Anchors* Merge (Threshold = 0.5)

Subject	AD		AS		HB		MV	
	#Debts	Merge	#Debts	Merge	#Debts	Merge	#Debts	Merge
Camel	97 (82%)	80→8	9 (0%)	-	45 (67%)	30→1	357 (91%)	324→12
Cassandra	24 (46%)	11→4	10 (0%)	-	19 (68%)	13→2	57 (93%)	53→1
CXF	26 (69%)	18→6	3 (0%)	-	3 (0%)	-	150 (84%)	126→8
Hadoop	12 (42%)	5→2	9 (0%)	-	5 (60%)	3→1	47 (74%)	35→7
HBase	92 (77%)	71→4	5 (60%)	3→1	30 (53%)	16→2	111 (87%)	97→2
PDFBox	5 (40%)	2→1	3 (67%)	2→1	2 (0%)	-	17 (88%)	15→3
Wicket	12 (92%)	11→3	2 (0%)	-	8 (75%)	6→1	77 (97%)	75→2
HIVE	109 (85%)	93→1	2 (0%)	-	6 (50%)	3→1	136 (96%)	130→2
Avro	2 (0%)	-	1 (0%)	-	1 (0%)	-	25 (80%)	20→3
Mesos	1 (0%)	-	2 (0%)	-	1 (0%)	-	5 (100%)	5→1
OpenJPA	5 (0%)	-	-	-	2 (100%)	2→1	84 (87%)	73→4
Httpd	3 (0%)	-	-	-	4 (100%)	4→1	10 (100%)	10→1
Kudu	-	-	1 (0%)	-	-	-	6 (83%)	5→1
Mahout	-	-	-	-	-	-	6 (83%)	5→2
Chemistry	-	-	-	-	-	-	2 (100%)	2→1
Jena	1 (0%)	-	-	-	-	-	22 (77%)	17→3
Ambari	4 (0%)	-	2 (0%)	-	-	-	23 (87%)	20→1
Allura	9 (67%)	6→2	4 (0%)	-	3 (100%)	3→1	21 (76%)	16→2

effectiveness of examining the merged debts. To be effective refactoring candidates the merged debts should contain a small portion of files but account for a large portion of maintenance costs. We conjecture that the higher the ratio the more effective it is for architects to review and refactor a debt.

Table XI shows the average characteristics of the compound debts after the merging. We can make the following observations:

- **Merging can significantly reduce a large number of *ATDs* in a project to a much smaller number of compound debts.** For example, in Camel there are 100 original *AD* debts that can be merged into just 25 compound debts. Thus architects can review a much smaller number of debt instances after the merge. We highlighted cells where merge is not applicable in blue—there are usually just a small number of such debts.
- **On average, the size-to-cost ratio of most debts is between 2 and 13. This means that the compound debts usually only contain a small portion of files in the system, but account for a relatively large portion of maintenance costs.** In particular, the sizes of the merged *AD* or *HB* debts are below 1% for most projects. The size-to-cost ratio of the *AD* and *HB* debts is, on average, 5 across the projects. The *MV* debts usually contain more files, containing 13.1% of the files in a project on average. And, the size-to-cost ratio of the *MV* debts is above 2 in most projects (except Httpd and Chemistry). The implication is that these merged debts are potentially cost-effective refactoring candidates, since architects only need to focus on a few groups of files to "pay off" a large amount of maintenance cost.
- **The *AD* and *HB* debts offer the highest cost effectiveness ratio in most projects, compared to the *AS* and the *MV* debts.** For each project, we highlighted one of the four types of debt with the highest size-to-cost ratio with a yellow background. As we can see, the *AD* and *HB* debts are the "winners" in most projects. Thus architects should prioritize the *AD* and *HB* debts as their refactoring candidates for even better cost effectiveness.

VII. Discussion

In this section, we discuss how architects can benefit from our approach, and some factors that may impact the results of our approach.

TABLE XI: Average Size-and-Churn Ratio for the Compound Debts (Threshold = 0.5)

Subject	AD				AS				HB				MV			
	#Debts	S.%	Ch.%	R.	#Debts	S.%	Ch.%	R.	#Debts	S.%	Ch.%	R.	#Debts	S.%	Ch.%	R.
Camel	100→25	0.2%	0.6%	5	9→9	0.1%	0.4%	4	46→16	0.1%	0.4%	7	357→45	1.8%	2.8%	3
Cassandra	35→17	0.3%	2.1%	7	11→10	0.5%	0.6%	2	21→8	0.3%	1.4%	4	57→5	18.8%	24.9%	2
CXF	27→14	0.2%	0.9%	4	3→3	0.4%	0.7%	2	3→3	0.2%	0.6%	3	150→32	2.1%	3.2%	2
Hadoop	16→9	0.5%	2.4%	5	11→9	1.1%	2.2%	3	7→3	0.4%	2.5%	4	47→19	2.5%	4.9%	3
Hbase	107→25	0.6%	1.9%	1	34→3	9.1%	10.3%	1	30→16	0.2%	0.7%	3	111→16	6.9%	12.1%	2
PDFBox	6→4	1.1%	4.7%	6	3→2	6.5%	16.2%	2	3→2	0.8%	4.0%	5	17→5	10.1%	17.5%	2
Wicket	12→4	0.3%	1.6%	6	2→2	1.3%	1.9%	1	9→3	0.2%	1.2%	8	77→4	9.2%	12.7%	2
HIVE	113→17	0.4%	1.9%	4	9→2	6.8%	13.9%	2	14→4	0.6%	1.7%	3	136→8	8.5%	12.2%	2
Avro	2→2	0.9%	2.1%	2	1→1	4.1%	6.0%	1	1→1	0.9%	2.7%	3	25→8	9.8%	14.2%	2
Mesos	1→1	1.2%	6.4%	5	2→2	3.1%	4.5%	2	1→1	0.8%	8.9%	11	5→1	29.1%	54.5%	2
OpenJPA	5→5	0.2%	2.2%	9	-	-	-	-	2→1	0.2%	1.7%	9	84→15	3.5%	7.2%	3
Httpd	3→3	4.9%	26.6%	5	-	-	-	-	4→1	9.5%	31.8%	3	10→1	74.1%	92.3%	1
Kudu	-	-	-	-	1→1	3.3%	4.9%	1	-	-	-	-	6→2	7.8%	14.2%	2
Mahout	-	-	-	-	-	-	-	-	-	-	-	-	6→3	4.9%	8.3%	2
Chemistry	-	-	-	-	-	-	-	-	-	-	-	-	2→1	13.7%	18.1%	1
Jena	1→1	0.4%	0.9%	2	-	-	-	-	-	-	-	-	22→8	3.3%	4.6%	2
Ambari	4→4	0.7%	2.2%	3	2→2	0.5%	1.7%	4	-	-	-	-	23→4	14.8%	18.5%	1
Allura	10→5	4.5%	14.7%	4	4→4	2.7%	5.2%	2	4→1	5.6%	19.3%	3	21→7	15.7%	26.6%	2
Min		0.2%	0.6%	1		0.1%	0.4%	1		0.1%	0.4%	3		1.8%	2.8%	1
Max		4.9%	26.6%	9		9.1%	16.2%	4		9.5%	31.8%	11		74.1%	92.3%	3
Avg		1.1%	4.7%	5		3.0%	5.3%	2		1.5%	5.9%	5		13.1%	19.4%	2

A. ATD Evolution

As discussed in Section III, our approach identifies an *ATD* as a $FileSetSequence$, which is a sequence of file groups, each extracted from consecutive releases. This provides a dynamic view to examine the evolution of *ATDs*. We manually inspected the evolution of these debts, and now illustrate how architectural flaws evolve into debts over time.

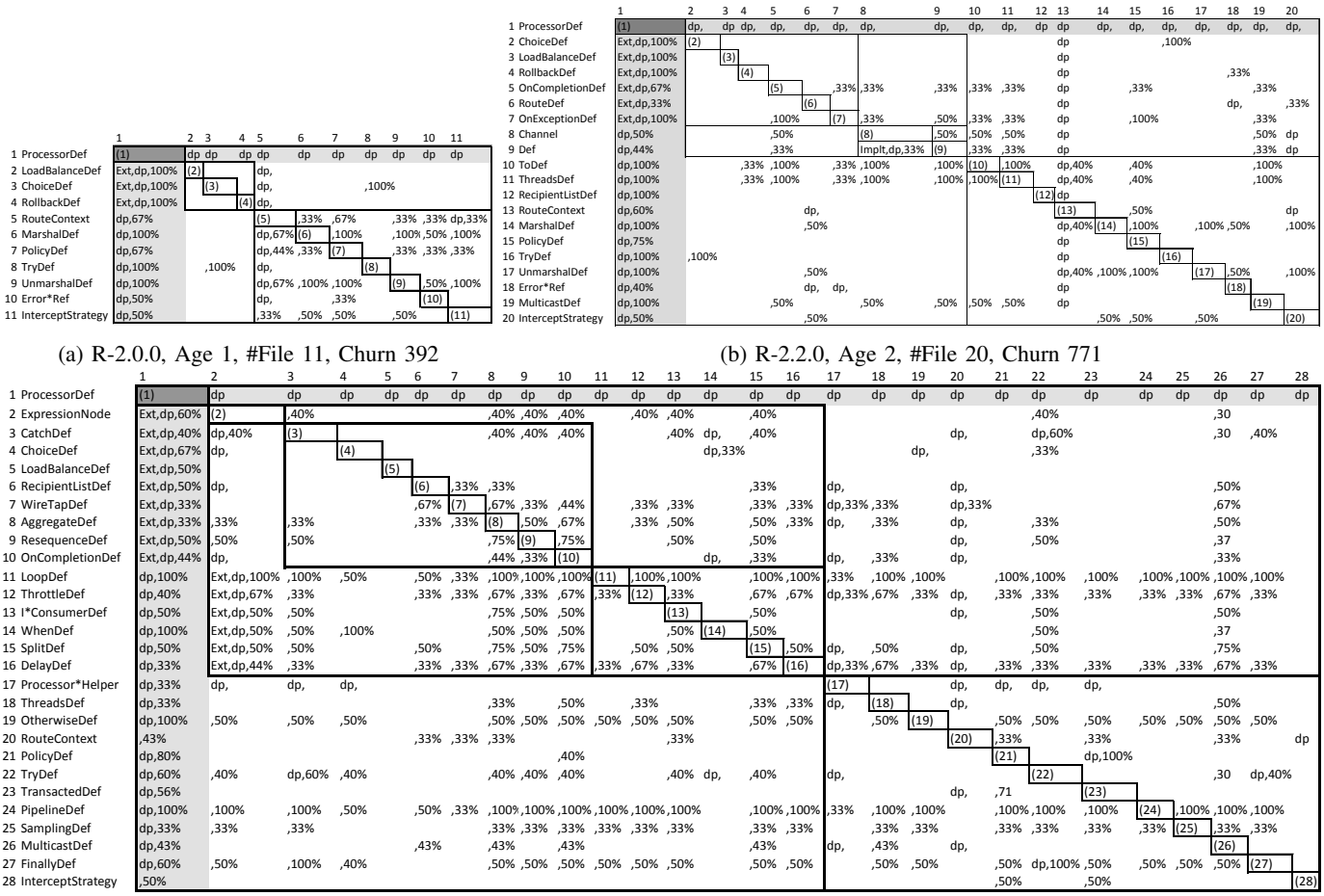
Figure 13 shows a debt we identified from Camel. We have provided 3 snapshots (i.e. $FileSet$) of this debt—in release 2.0.0 (age 1), release 2.2.0 (age 2), and release 2.12.4 (age 11)—to show its evolution. Snapshots from age 3 to 10 are similar to age 11. "Ext" and "Impl" stand for "extend" and "implement", "dp" denotes all other types of structural dependencies.

In release 2.0.0, $PDef$ forms a dependency "hub" with 10 other files: 3 files are its subclasses, 7 files are its general dependents, and $PDef$ structurally depends on all of them. We call these files a hub, which is a typical debt pattern as we will introduce later. Note that in this snapshot, all files, except *InterceptStrategy*, depend on *RouteContext* (column 5). The 11 files in this hub structurally form a strongly connected graph. According to the revision history, $PDef$ changes with all member files with probabilities from 50% to 100% (column 1). The dependents (on rows 5 to 11) of $PDef$ are highly coupled with each other. This is problematic in 3 ways: 1) the parent class $PDef$ depends on each subclass and each dependent class (unhealthy inheritance [26]); 2) the parent class is unstable and often changes with its subclasses and dependent classes (unstable interface [26]). 3) *RouteContext* forms cyclic dependencies with 9 files (cycles). Without fixing these flaws, we expect the maintenance costs of this group to grow.

In release 2.2.0, the impacts of this hub have enlarged—$PDef$ has 3 more subclasses and 6 more general dependents, and it depends on each of them as well. Each newly involved file also depends on *RouteContext* (column 13). The revision history shows that $PDef$ changes with its subclasses and dependents with probabilities of 33% to 100%. Also, the subclasses and dependents (rows 5 to 11) of $PDef$ are highly coupled with each other—changing any of them is likely to trigger changes to the rest. In following releases, the hub grows further. Up to release 2.12.4, $PDef$ has 9 subclasses and 18 general dependents—the size of the hub tripled compared to the start, and, as always, $PDef$ depends on each of them. In addition, 6 of the 18 general dependents (rows 11 to 16) of $PDef$ also become its grandchildren. The inheritance tree has increased in width and depth. The revision history shows $PDef$ still changes with its dependents with probabilities from 33% to 100%. The files in this snapshot are tightly coupled with each other, and so changing any file is likely to trigger changes to others.

The maintenance costs spent on this debt fit a linear regression model: $DebtModel(rt) = 158.75 * rt + 509.35$ with $R^2 = 0.89$. This means that, in every release, developers contribute 158.75 more lines of code to fix errors in the hub anchored by $PDef$. Although this model can only be obtained *after* the costs and penalty have accumulated, one could use our approach to detect architecture flaw patterns at any point (as described in [26]), monitor how file groups grow, monitor the formation of debts, and prevent significant costs by investing in proper refactorings [31].

In addition, we performed additional analyses to understand how and "why" the debt in Figure 13 accumulated. The debt center file *"ProcessorDefinition"* has a large number of dependents, which tend to change together with it over time. This is the result of change propagation through the structural coupling—when *"ProcessorDefinition"* changes, its dependents change accordingly. Throughout the project's history, more and more files are added and/or become dependent on *"ProcessorDefinition"*. To further understand why this happens from the perspective of LoC, we found that the LoC of *"ProcessorDefinition"* increased from 923 in release 2.0.0 to 1382 in release 2.12.4 (50% increase). In addition, its member files, including *"LoadBalanceDefinition", "ChoiceDefinition", "RollBackDefinition", "MarshallDefinition"*, and



(a) R-2.0.0, Age 1, #File 11, Churn 392

(b) R-2.2.0, Age 2, #File 20, Churn 771

(c) R-2.12.4, Age 11, #File 28, Churn 2134

Fig. 13: Camel Hub Debt Evolution—Anchor ProcessorDefinition

"UnmarshallDefinition" increased in LoC by 35% to 78% during the releases. We believe this analysis suggests: 1) large files are likely to be the center of a debt due to their poor design—encompassing too many responsibilities and thus causing a large amount of co-changes, so that their sizes have to keep growing; 2) the member files, as defined in our debt pattern, tend to grow and change with the anchor file.

B. ATD Patterns and Potential Refactoring Guidelines

One of the key contributions of this work is to identify *ATD* in terms of the four indexing patterns: hub, anchor submissive, anchor dominate, and modularity violations. We manually reviewed the debt patterns identified by our approach, and summarized how each *ATD* pattern maps to potential refactoring resolutions.

The hub pattern features a "hub" like central file, which structurally depends on, and is also depended by, all the member files in this pattern. Hub pattern instances are likely to overlap with several design flaws that are formed by a subset of the involved files. For example, the "hub" file and its depends are likely to form unstable interface or unhealthy inheritance. And each "hub" file also forms cyclic dependencies with its members. However, an architect cannot treat a hub pattern as separate cyclic dependencies or unstable interfaces to successfully refactor this pattern. The reason is that the involved source files form a more complicated whole. To successfully eliminate a "hub", an architect needs to analyze the internal elements in the "hub" file, and decompose it into separate modules, which decouple files that originally depend on it and that it depends on.

The anchor submissive features an anchor file that frequently changes with files that structurally depend on it. The anchor file in this pattern most likely defines a fundamental API or basic utility functions for its dependents. Therefore, the anchor and its dependents form an unstable interface. The anchor is a servant interface that frequently accommodates the needs of its dependents and change its interface. The guideline to refactor this pattern is two-fold: 1) identify the most change-prone interfaces in the anchor; and 2) encapsulate the change-prone interfaces in a separate class and/or stablize the interfaces following the "design-by-contract" principle. The anchor dominate pattern is similar to the anchor submissive pattern. The difference is that in the anchor dominate the interface changes, while its dependents accommodate the change. The refactoring guideline is therefore similar: identify the most change-prone interfaces in the anchor and encapsulate

or stablize those interfaces.

The modularity violation pattern is different than the above three patterns. The files involved do not share structural dependencies, however they frequently change together due to latent dependencies. These latent connections could be the result of "share secrets", such as the semantic and/or structural similarity among the involved files. The semantic similarity could be the usage of parameters that represent the same concepts, such as time units [32], or might simply be due to code cloning. The structural similarity could be common dependencies on other interfaces; when these interfaces change, the files with shared dependencies often need to change together. To refactor the modularity violation, an architect needs to identify the "shared secrets" among the involved files, and encapsulate these secrets into an abstraction. A common example of eliminating a modularity violation is to encapsulate code clones into separate methods or classes.

C. *Why do some projects only have a few* ATDs?

As shown in Table II, Kudu, Mahout, Chemistry, and Jena contain debts that only account for a small percentage (up to 31%) of maintenance costs. This section discusses the underlying reason for this.

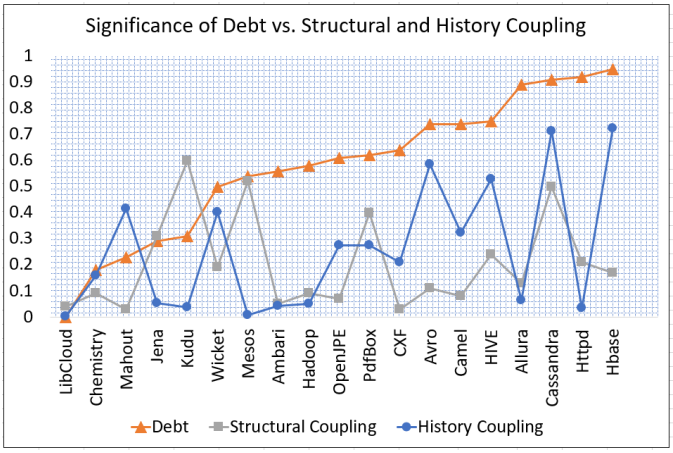
As described in Section III, an architectural debt is a group of connected files that keep incurring higher maintenance costs over time. Based on this definition, the debts are identified based on two criteria: 1) the structural dependencies and 2) the history coupling among files. Debts are matched by the four indexing patterns (Section IV) combining structural and history couplings among files.

Thus, we should understand how the debts correlate with both structural coupling and history coupling. We measure the coupling using the Propagation Cost (PC) metric proposed in [33]. PC is the density of the n-transitive closure of a dependency matrix. A P value ranges between 0 and 1; the larger the value, the more coupled the elements in the matrix. For example, a PC of 1 indicates that every element in the matrix is connected to every other element, directly or indirectly. In Figure 14, the x-axis lists the projects and the three trend lines describe 1) the significance of debts in each project (triangle marker); 2) the propagation cost calculated based on the structural dependency DSM (square marker); and 3) the propagation cost calculated based on the HCP matrix (round marker). The projects are ranked by the significance of debts in ascending order. The key observations are that:

- the significance of debts is not correlated with the structural coupling, with a correlation coefficient of 0.1. The rationale is that a project can have very tangled structural coupling, but there is no debt if the project seldom changes (i.e. no maintenance "interest"). For example, Kudu has one of the highest PC values (0.6, based on structural dependencies) among the 18 projects. However, the evolutionary coupling (PC < 0.1) is almost the lowest. This indicates that although Kudu has tangled structural dependencies, the project is relatively stable, therefore no persistently change-prone architectural flaws are identified.
- the significance of debts is only moderately correlated with history coupling, with a correlation coefficient of 0.5. This is because our approach identify debts based on *persistent* co-change file groups in the evolution history. Note that high history coupling does not always lead to debts, because a group of files may change together for many reasons (e.g. new functionality, minor syntactic changes, new coding or documentation conventions, etc.) that do not impact architectural debt.

In addition low evolutionary coupling could be due to missing links, in some projects, between bug reports and the code commits that fix those bugs [34]. We acknowledge that this is a potential limitation and threat to validity of our approach in that it relies on the quality of history data. For example, if developers regularly make fragmented commits, only changing one source file a time, our approach will not be able to identify meaningful debts. And if developers "hoard" their changes on a private branch and then commit large changes (with dozens of files), our approach will tend to identify more debts. In our study, we attempted to mitigate this problem by only analyzing changes that involve fewer than 30 files. This helps to eliminate the "noise" introduced by very large commits.

Fig. 14: Debt Significance vs. Structural and History Coupling



D. *How does the* R^2 *threshold impact the regression models?*

For each *Architectural Debt*, we searched for a suitable regression model to describe the trajectory of the associated maintenance costs. We used an R^2 threshold as described in Section IV. In this section, we investigate how the regression models are impacted by the R^2 threshold.

Table XII lists the distribution of the four regression models: Linear (Li), Logarithmic (Lg), Exponential (Ep), and Polynomial (Pl). For each model we examined four R^2 thresholds: 0.6, 0.7, 0.8, and 0.9. In general, a lower R^2 threshold better captures the rough trend of maintenance costs over time, while a higher R^2 threshold better captures detailed maintenance cost fluctuations.

The first two columns show the project name and the total number of *Architectural Debts* in each project. The following

TABLE XII: Regression Model Types

Subject	#Dts	R^2=0.6				R^2 = 0.7				R^2=0.8				R^2 = 0.9			
		Li	Lg	Ep	Pl	Li	Lg	Ep	Pl	Li	Lg	Ep	Pl	Li	Lg	Ep	Pl
Camel	512	97.5%	2.1%	0.4%	-	91.8%	7.4%	0.4%	0.4%	68.9%	28.5%	-	2.5%	31.8%	50.8%	0.2%	17.2%
Cassandra	124	100.0%	-	-	-	96.0%	3.2%	0.8%	-	93.5%	5.6%	0.8%	-	82.3%	7.3%	2.4%	8.1%
CXF	183	97.8%	1.1%	0.5%	0.5%	96.2%	0.5%	2.2%	1.1%	90.2%	2.2%	4.9%	2.7%	58.5%	9.8%	17.5%	14.2%
Hadoop	81	93.8%	3.7%	-	2.5%	85.2%	4.9%	6.2%	3.7%	71.6%	7.4%	11.1%	9.9%	55.6%	3.7%	6.2%	34.6%
HBase	282	97.9%	-	1.8%	0.4%	96.1%	0.4%	2.1%	1.4%	89.4%	1.8%	6.0%	2.8%	67.7%	5.7%	12.1%	14.5%
PDFBox	29	100.0%	-	-	-	96.6%	-	3.4%	-	72.4%	3.4%	24.1%	-	51.7%	-	27.6%	20.7%
Wicket	100	99.0%	1.0%	-	-	95.0%	4.0%	1.0%	-	85.0%	8.0%	5.0%	2.0%	57.0%	21.0%	3.0%	19.0%
OpenJPA	91	92.3%	6.6%	1.1%	-	78.0%	17.6%	3.3%	1.1%	51.6%	39.6%	2.2%	6.6%	8.8%	53.8%	2.2%	35.2%
HIVE	272	99.3%	-	0.7%	-	96.7%	-	2.9%	0.4%	92.6%	0.4%	5.5%	1.5%	66.5%	1.1%	24.3%	8.1%
Avro	29	100.0%	-	-	-	100.0%	-	-	-	89.7%	-	6.9%	3.4%	44.8%	20.7%	17.2%	17.2%
Mesos	9	100.0%	-	-	-	100.0%	-	-	-	100.0%	-	-	-	100.0%	-	-	
Httpd	17	100.0%	-	-	-	100.0%	-	-	-	100.0%	-	-	-	82.4%	5.9%	5.9%	5.9%
Kudu	7	100.0%	-	-	-	100.0%	-	-	-	85.7%	14.3%	-	-	71.4%	28.6%	-	
Mahout	6	66.7%	33.3%	-	-	66.7%	-	-	33.3%	50.0%	16.7%	-	33.3%	16.7%	16.7%	-	66.7%
Chemistry	2	100.0%	-	-	-	100.0%	-	-	-	100.0%	-	-	-	100.0%	-	-	
Jena	23	95.7%	4.3%	-	-	82.6%	4.3%	8.7%	4.3%	39.1%	30.4%	13.0%	17.4%	8.7%	21.7%	8.7%	60.9%
Ambari	29	100.0%	-	-	-	100.0%	-	-	-	96.6%	3.4%	-	-	82.8%	6.9%	-	10.3%
Allura	39	100.0%	-	-	-	97.4%	2.6%	-	-	92.3%	5.1%	2.6%	-	35.9%	56.4%	-	7.7%
Min	2	66.7%	1.0%	0.4%	0.4%	66.7%	0.4%	0.4%	0.4%	39.1%	0.4%	0.8%	1.5%	8.7%	1.1%	0.2%	5.9%
Max	512	100.0%	33.3%	1.8%	2.5%	100.0%	17.6%	8.7%	33.3%	100.0%	39.6%	24.1%	33.3%	100.0%	56.4%	27.6%	66.7
Avg	102	96.7%	7.4%	0.9%	1.1%	93.2%	5.0%	3.1%	5.7%	81.6%	11.9%	7.5%	8.2%	56.8%	20.7%	10.6%	22.7%

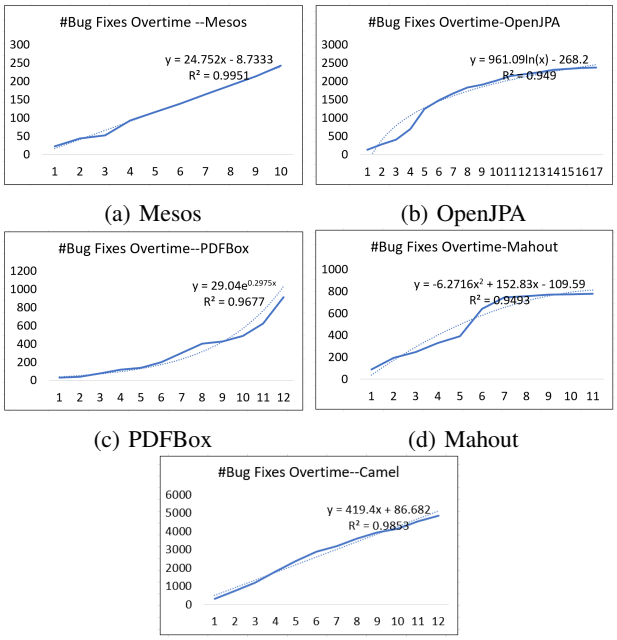
columns show the distribution of the four regression models at each threshold. When using 0.6 for the R^2 threshold, the majority (66.7% to 100%) of debts in each project can be described as a linear model. On average, only 7.4%, 0.9%, and 1.1% of debts fit into Logarithmic, Exponential, and Polynomial models, respectively. This indicates that the maintenance costs associated with most debts increase linearly over time in general. However, as we increase the R^2 threshold to 0.7, 0.8, and 0.9, more debts are described by other regression models to capture the detailed cost fluctuations. For example, when choosing 0.9 for the R^2 threshold, on average, 20.7%, 10.6% and 22.7% of the debts fit into Logarithmic, Exponential, and Polynomial regression models, respectively.

Of particular note, when the R^2 threshold is 0.9, the coverage of each regression model in different projects differs substantially, compared to the dominance of the *Linear* model when the R^2 threshold is 0.6 to 0.8. For example, when the threshold is 0.9, 56.4% of Allura's debts are best fit by a *Logarithmic* model; PDFBox scores 27.6% with an *Exponential* model; Mesos scores 100% with a *Linear* model; while Mahout scores 66.7% with a *Polynomial* model. We assume that the fit of different debt models is impacted by the trend of maintenance activities in each project. For instance, Figure 15 shows the trend of bug fixing per release in Mesos, OpenJPA, PDFBox, and OpenJPA respectively. The x-axis is the release number in each project and the y-axis is the total number of bug fixes by the time of each release.

We can observe that, for example, 53.8% of debts in OpenJPA fit a logarithmic model. We see that OpenJPA stabilized in later releases, and bug fixing activities were slowing down. This could be the result of a refactoring that improved the architecture of OpenJPA, or simply because the project was not as active as in earlier releases. Similarly, 100% of debts in Mesos fit a linear model, as illustrated in Figure 15a. PDFBox's maintenance activities increase with releases following a exponential model, thus, PDFBox has the highest percentage of *Exponential* debts compared to other projects. Finally, 66% of the debts in Mahout fit a *Polynomial* model, as shown in Figure 15d.

Camel is an interesting outlier. As seen in Figure 15e, the trend of maintenance activity of Camel follows a linear model, indicating that developers invest a steadily increasing amount of cost on fixing bugs over time. However, 50% of debts in Camel fit a *Logarithmic* model, indicating that these debts are costing less maintenance cost over time. We conjecture that this is because the 31% of debts fitting the *Linear* model are incurring higher maintenance costs, which offset the reduced "interest" from the 50% of *Logarithmic* debts. In other words, although 50% of debts are incurring lower interest, the other 31% of debts are incurring a higher interest, resulting in a stably increasing trend line for the overall maintenance costs.

Despite this one exception in Camel, we observe that, in every other case, the best fitting debt model type is impacted by the trend of the maintenance activities in a project.



(a) Mesos (b) OpenJPA

(c) PDFBox (d) Mahout

(e) Camel

Fig. 15: Maintenance Activity Trend Line

E. How does the Compound Anchors threshold impact the characteristics of compound debts after merging?

As shown in Equation 17, we use an overlap threshold to decide whether to merge two debts. In this section, we evaluate how this threshold impacts the characteristics of the compound debts after merging with *Compound Anchors*. In particular, we are interested in: 1) the number of compound debts after merging; and 2) the average size (number of files), the average maintenance cost (in terms of the churn associated with those files), and the ratio between the two. The first aspect determines the maximum number of debts architects need to review, while the second aspect reveals the potential cost-effectiveness of inspecting each debt. We tested thresholds from 0.1 to 0.9, and the results are shown in Table XIII. We can make the following observations:

- As we increase the overlap threshold, the number of compound debts increases for all four debt types. This is consistent with the intuition that a higher threshold poses a more strict criterion thus preventing some merges. Therefore, more distinct debts will remain after merging. The implication is that the architects can expect to review more debt instances if the overlap threshold is high.
- As we increase the overlap threshold, the potential cost-effectiveness of each compound debt increases for the *AD*, *HB*, and *MV* debts across projects. For example, the size-and-churn ratio of the *AD* debts increases from 4 to 7 as the threshold increases from 0.1 to 0.9. In other words, as the threshold increases, each compound tends to contain fewer source files, but each debt still accounts for relatively higher maintenance costs with respective to its number of files. Therefore, the implication is that a higher overlap threshold helps architects focus on compound debts with higher cost effectiveness. Of particular note, the cost effectiveness of the *AS* debts remain stable as we increase the threshold; we conjecture that the reason is that merge is only applicable to 7 projects even when the threshold is 0.1. Therefore, the cost effectiveness rate stays close to that of the original debts (i.e., without merging) for *AS* debts.

VIII. Related Work

A. Technical Debt.

The concept of *TD* was first coined by Cunninghan [1]. It refers to the trade-offs developers make between long term benefits, such as maintainability, and short term goals, such as meeting a deadline. Much research has focused on *TD*, to understand the *TD* landscape, on practitioners' perceptions of *TD*, as well as best practices for managing *TD*. We have observed several trends in *TD* research in recent years [2], [3], [5]–[7], [7]–[13], [19]–[25], [35]–[55], [55]–[68], [68]–[72].

a) Self-Admitted Technical Debt: An important body of work focuses on identifying and analyzing Self-Admitted *TD* (*SATD*) [46]–[58]. *SATD* refers to *TD* that is noted by developers in comments and issue reports, such as "TODO" items. The key to this type of *TD* is that the developers intentionally introduce these debts to strategically balance long-term and short-term goals in software development. Most work in this area has focused on detecting *SATD* using text mining [49], [51], natural language processing [56] and deep learning techniques [53]. Yan et. al [48] focused on identifying changes that introduce *SATD*. Wehaibi et. al [50] investigated the impact of *SATD* on software quality, revealing that the introduction of *SATD* correlates with defectiveness of the involved files, and files involved in *SATD* are more difficult to change. Mensah et. al [54] proposed a prioritization scheme for treating *SATD* in practice. Kamei et. al focused on measuring the interest of *SATD* in terms of LoC and Fan-In [55]. Maldonado et. al [47] and Zampetti et. al [57] investigated how *SATD* is removed in practice. One large-scale study revealed that *SATD* is mostly code debt (30%), defect debt (20%), and requirement debt (20%) [46]. In addition, Sierra et. al [52] evaluated whether *SATD* can serve as an indicator of architectural debt. The answer is that 14% of the architectural debt can be traced through *SATD*, but the effort of doing so is high and generally inefficient. Our paper specifically focused on *Architectural Debts*, which are largely overlooked by developers when documenting *SATD*. We conjecture that *Architectural Debts* are more likely to be introduced accidentally without developers' awareness. Literature about *SATD* focused on identifying debts that are admitted by developers through code comments and issue reports. In comparison, our study proposed an automated approach to identify and quantify *ATDs* by mining project repository and matching the architectural and evolutionary patterns. Therefore, our work complements the existing work in *SATD* by identifying potential *ATDs* that developers are not aware of.

b) TD Survey/Interview: An important research methodology for understanding *TD* is through surveys or interviews involving practitioners [5], [19]–[23], [59]–[63]. These studies focused on understanding practitioners' perceptions of *TD* and discovering current practices of *TD* management. Rios et. al [22], [23] revealed that *TD* is usually caused by a combination of factors in project planning/management such as deadlines and inadequate planning, and the lack of problem domain and technology knowledge. Perez et. al [19] had similar findings in another survey. They emphasized that the concept of TD is well understood by practitioners, and that researchers need to offer strategies and tools to support TD management. Yli-Huumo et. al [20] interviewed 25 professionals from 8 development teams to understand current practices in *TD* management of *TD* repayment, identification, measurement, monitoring, prioritization, communication, prevention, and documentation. They found that different teams have different maturity levels and pointed out that one of the challenges in *TD* management is the lack of tools. Besker et. al [59] conducted an interview with 16 professionals in software startups, showing that startups intentionally use *TD* as a strategy to achieve a "good enough" product with rapid time to market. In addition, Besker et. al [5] conducted another study, focusing on the impact of *ATD* on daily development. They reported that *ATD* has the highest negative impact and impacts all roles in a

TABLE XIII: Impact of Overlap Threshold (0.1 to 0.9) on Merging

Overlap	AD				AS				HUB				MV			
Thresh	#MDebts	S.%	Ch.%	R.	#MDebts	S.%	Ch.%	R.	#MDebts	S.%	Ch.%	R.	#MDebts	S.%	Ch.%	R.
0.1	4	2.8%	10.3%	4	2	5.5%	9.3%	2	3	1.6%	6%	5	1	40%	53	%1
0.2	4	2.8%	10.3%	4	3	4.3%	8%	2	3	1.5%	6%	5	2	35%	45%	1
0.3	5	1.51%	6.11%	5	3	3.43%	6.41%	2	3	1.52%	5.91%	5	3	28.89%	37.83%	2
0.4	8	1.37%	5.70%	5	3	3.03%	5.25%	2	4	1.52%	5.90%	5	6	16.64%	23.13%	2
0.5	8	1.15%	5.03%	5	3	3.03%	5.25%	2	4	1.52%	5.90%	5	10	13.73%	20.25%	2
0.6	19	1.02%	4.65%	6	3	2.72%	4.76%	2	8	0.71%	3.55%	7	29	10.06%	16.26%	3
0.7	21	0.97%	4.54%	6	3	2.62%	4.64%	2	8	0.71%	3.56%	7	44	6.65%	12.43%	3
0.8	23	0.97%	4.55%	7	3	2.62%	4.64%	2	8	0.71%	3.56%	7	55	6.11%	11.64%	3
0.9	23	0.96%	4.52%	7	3	2.62%	4.64%	2	8	0.71%	3.56%	7	63	6%	11.2%	3
Trend	↑			↑	—			—	↑			↑	↑			↑

development team compared to other types of *TD*. In other studies, Besker et. al [62], [63] found that *TD* cripples software development productivity. Holvitie et. al [61] conducted a survey with 184 responses for understanding *TD* and agile development practices. Their main finding is that agile practices and processes help to reduce technical debt; in particular, techniques such as coding standards and refactoring positively affect technical debt management. Martini et. al [21] conducted a survey with 226 respondents from 15 organizations. They found that *TD* Management requires substantial dedication from developers, probably due to the lack of effective tools. Currently, the most used and effective tools are backlogs and static analyzers.

The main findings of these works reveal that practitioners are aware of the negative impact of *TD*, but the most challenging part is *TD* management. We believe that the approach presented in this paper has the potential in facilitating *TD* management with its ability to automatically detect, to quantify the interest, and to predict the future costs of *TD*.

c) TD Secondary Studies: Due to the large volume of *TD* research, a set of secondary studies has reviewed the status of the *TD* research [2], [3], [6], [64]–[69]. This helps researchers understand whether current research aligns with the practitioners' expectation. Rios et. al [2] studied 13 studies of *TD* between 2012 and 2018. They identified a taxonomy of *TD* types, including design debt, code debt, architectural debt, test debt, documentation debt, defect debt, infrastructure debt, etc. Among these, design, code, and architecture debt are the most cited types. Similar findings have been reported by Alves et. al [3] and Li et. al [4]. Behutiye et. al [64] studied 38 papers about *TD* in the context of Agile development and found that architectural and design issues are the most common causes of *TD* in Agile development. Verdecchia et. al [65] studied 47 papers focusing on architectural *TD*. They found that most studies focused on architectural anti-patterns and smells, and modularity analyses, based on source code and evolutionary data. Besker et. al [6] pointed out that there is a compelling need for supporting tools and methods for system monitoring and evaluating ATD. A key challenge in this area is to quantify and predict the economic consequences of architectural *TD*. Fernández-Sánchez et. al [66]'s study concluded that *TD* management should be context dependent, considering the history of product development, prospects, and time to market. They also pointed out that quantifying and visualizing *TD* is important for communication in the decision-making process. Lenarduzzia et. al [68] noted, based on 44 studies of *TD*, that prioritization is preliminary and researchers need to put more effort on determining the important factors and how to measure them. Becker et. al [69] suggested that more effort should be put on investigating the decision making process for *TD* management, in particular the "intertemporal" choices in *TD* management.

d) TD Quantification and Prioritization: Some work specifically focused on quantifying the principle, interest, and probability of interest of *TD* in software projects [24], [25], [55], [68], [70]–[72]. Lenarduzzi et. al [68] summarized ten different aspects of impacting factors considered during debt prioritization in research and practice, based on 44 primary studies. The ten aspects of impact factors considered for debt prioritization are 1) business factors, such as lead time and market competitiveness; 2) customer factors, such as satisfaction and expectations; 3) evolution, such as impact on features; 4) maintenance, such as number of bugs and maintenance cost; 5) system quality, such as security and robustness; 6) quality debt—# of issues or their co-occurrence; 7) productivity, such as wasted development hours; 8) project factors, such as project size and complexity; 9) social factors, such as developers' morale and team culture; 10) other factors, such as user perception and number of users affected. Martini et. al [24], [25] measured the negative impact of *TD* in development speed, bugs, quality compromised, extra costs, frequency of issues, and users affected. These inputs were provided by developers working on the studied projects. Kosti et. al [70] focused on estimating the principal of *TD* by modeling seven structural metrics. The rationale is that the higher the structural metrics, such as coupling and cohesion, the higher the principle of a *TD*. Codabux et. al [71] used the metrics of defect- and change-prone classes to build a prediction model for debt proneness of classes. This helps to prioritize debts. Amanatidis et. al [72] considered developer characteristics, such as expertise level, to estimate the principle of *TD*.

e) Architectural TD: [7], [7]–[13]. As noted above, architectural *TD* is one of the most cited types, due to its significant negative impacts on software development [2], [3]. Li et. al [4] further categorized Architectural *TD* into seven sub-categories, including architectural smells [14], architectural anti-patterns [15], [16], complex architectural behavioral dependencies [73], violations of good architectural practices [17], architectural compliance issues [74], system-level structural

quality issues, and all others. Martini et al. [18] conceptualized two patterns of Architectural *TD*: contagious debt and vicious circle. Contagious debt leads to ripple effects in projects. Vicious circle refers to a more severe contagious debt where the ripple effects form a loop. Martini et. al [7] proposed a framework to help practitioners decide if and when to refactor architectural *TD*. Towards this end, the authors considered the coupling among debt elements for estimating the interest, and also collected data about the impact on development speed, maintainability, learning, etc. Skiada et. al [8] explored the relationship between modularity and *TD*, finding that a lack of modularity often co-locate with *TD*. MacCormack et. al [12] revealed that high coupling is an essential factor that contributes to system architectural *TD*. Roveda et. al [13] proposed an *Architectural Debt Index* to quantify the significance of architectural smells based on existing tools, including the approach proposed in our own prior work [75], [76].

f) Industrial Application of the ATD Patterns: In our previous studies [31], [32], [77], [78], we specifically focused on evaluating and verifying the harmfulness of the kinds of ATD patterns identified in this paper, including extensive user studies and interviews of developers. In each of these empirical studies of industrial projects the architects and developers were intensively involved, providing feedback regarding the harmfulness of the identified debts and benefits of identifying and fixing them. In particular, in study [32], our analysis resulted in quantifiable evidence to support a refactoring proposal, convincing the project manager to invest in a major refactoring. In the other two studies [31], [77], the developers carried out long-term refactorings following the suggestions provided by the identified ATD patterns. These studies all provided real-world evidence that the ATD patterns formalized in this paper are truly harmful and deserve attention. The main contribution of this paper is to provide a formalized, systematic approach to identify, quantify and predict the costs of ATDs.

g) TD Management Tools/Platforms: There are several tools/platforms developed for managing *TDs* in large-scale systems, including CodeScene [2], AnaConDebt [3] and Arcan [4]. CodeScene is an online platform providing code visualization dashboard based on software repository [79]. It identifies social patterns and hidden risks in code. More specifically, CodeScene detects "hotspots"—complex code that an organization has to work with frequently. It prioritizes *TD* based on the frequency [80]. In addition, CodeScene also focuses on the team dynamics of the software development team, such as their social networks and code ownerships. AnaConDebt is a TD management tool that consists of a TD-enhanced backlog, developed by Martini et. al based on empirical experience with 6 software development companies [7], [81]. The backlog allows the creation of TD items and allows performing TD-specific operations on the item, including tracking TD items, assessing TD principal and interest, comparing and ranking TD items, visualizing TD items, etc. These operations can aid decisions on *ATD* refactoring. AnaConDebt provides a generalized framework to track and manage TD items, but the identification and assessment of the TD items requires project expert's input and experience. Arcan is a tool for architectural smell detection [82]. It detects four smell patterns, namely class level cyclic dependency, package level cyclic dependency, unstable dependency, and hub like dependency. In a recent study, Martini et al. [83] conducted a case study using Arcan for identifying *ATD* in four industrial projects with a questionnaire, interviews and thorough inspection of the code with the practitioners. The goal is to reveal whether and how practitioners could identify and prioritize *ATD* through the architectural smells identified by Arcan. Their study focused on three architectural smell patterns, including unstable dependency, hub-like dependency, and cyclic dependency. They found that using these architectural smells to identify and prioritize *ATD* was considered useful to identify unknown problems by the developers. In addition, cyclic dependency and hub-like dependency are considered more harmful and thus having higher priority than unstable dependency. Our study also investigate similar *ATD* patterns. We leverage the history coupling as an additional layer of architectural connections to pin-point *ATDs* that incur higher maintenance costs (approximately by changes) compared to their size. And, *modularity violations* is a unique pattern in our work. Furthermore, the debt aggregation step in our approach helps to identify compound structures that are composed by "atomic" patterns. Lastly, our study focuses on the formalization and quantification of *ATD* identification and prioritization; while Martini et al.'s work [83] is an empirical case study applying Arcan and taking practitioners' opinion .

h) Comparison of Technical Debt Detection Tools: A recent empirical study compares the technical debts identified by 6 well-known tools—including Structure 101, SonarQube, Designite, DV8, Archinaut, and SCC [84]. One of the selected tools, DV8, identifies the architectural debt patterns in this work. The authors compared the consistency of the identified results over 10 projects. The conclusion is that DV8 identified significantly different debts than the other tools. In addition, the debts identified by DV8 were strongly associated with true maintenance difficulties across the studied projects. Most importantly, the debts identified by DV8 provide additional insights into the root causes of the debt, that is, problematic structures among files, and hence they have the potential to guide refactoring. For example, the *Modularity Violation* and *Anchor Dominant* anti-patterns that DV8 identifies give insight into not only *what* files are problematic, but also *how* to refactor those files to remove the root cause of the debt. This empirical study highlights the necessity of leveraging change-based information in technical debt detection.

While the main contribution of this paper is to formalize the definition, identification, quantification, and prediction of architectural debt. In particular, in RQ2 and RQ3, we focused on architectural debt cost prediction, and aggregating pieces of architectural debts into compound debts based on their more

[2]https://codescene.io

[3]https://anacondebt.com

[4]http://essere.disco.unimib.it/wiki/arcan

architectural connections. We have not found any existing work that addresses similar goals. Therefore, we could not make any direct comparison of these parts with existing studies.

B. Co-change Analysis

Analysis of co-changes in software projects at the package, class, method, and statement level has been used to gain insight in and provide solutions for problems in software development. Prior research has focused on analyzing co-change patterns in software evolution for different purposes such as predicting software change impact [85]–[88], predicting defects [89]–[91], revealing design problems [92]–[99], and visualizing co-changes to improve understanding [100]–[103].

Zimmermann et al. [104] applied data mining on revision histories to predict likely changes given a change that has already occurred. They contributed the ROSE tool to predict files to be changed based on a given change [85]. Kagdi et al. [105] proposed an approach to calculate the change impact scope of a software entity by combining structural coupling, reflected in source code, and change coupling, recorded in the project's revision history. Their approach improved the accuracy of change impact analysis, compared with either technique used independently. Gethers et al. [106] proposed an integrated approach to identify the impact set of a change request (e.g. a bug ticket in bug-tracking database), based on data mining of past source code commits and run-time traces. Others aimed at improving the accuracy of impact analysis [86]–[88]

Analysis of co-changes has also been used in reverse-engineering. Beck et al. [107] used co-change analysis to compute clusterings. They used an *Evolutionary Class Dependency Graph* to represent co-change coupling. They calculated three types of clusterings using (1) only co-change coupling, (2) only structure dependencies, and (3) a combination of the two. They found that clustering based on the combined approach yielded the best results.

Co-change analysis has also been applied to investigate problems in software projects, such as bugs and code smells. Wiese et. al. combined historical data with social metrics collected from developer interactions for defect prediction [89]. Kouroshfar et al. [108], [109] investigated how co-changes impact bugs. They found that co-changes dispersed across different sub-systems are more likely to result in bugs than localized co-changes. Girba et al. [110] used co-change patterns to identify hidden dependencies among different areas of a software system that reveal bad smells. They defined history patterns at three granularity levels: method, class, and package. These patterns can reveal code smells, such as similar code, cloned code, and shot-gun surgery. Code smells have also been used as a heuristic for approximating *TD*. Zazworka et al. [111] reported that not all *TD* approximated by code smells will lead to high maintenance costs, and not all *TD* has code smells. Zhou conducted a study of open-source systems employing 6 types of co-change relationships to reveal design problems [92]. Wong proposed a approach to identify modularity violations through co-change analysis, which reveal symptoms of poor design [93]. Mondal presented a study of co-change patterns in methods, which has the potential to pinpoint design deficiency [98]. Silva classified co-change patterns into three types and collected the perceptions of expert developers on these patterns [99].

IX. Limitations and Threats to Validity

We now explain the threats to the validity of this research.

First, we acknowledge that the consequences of *ATDs* can include many other aspects of costs and issues, such as availability, evolvability, scalability, reliability etc. However, the approach proposed in this paper specifically focuses on detecting and quantifying *ATDs* that incur high maintenance costs over time—approximated by the bug-fixing churn associated with source files. The reason is that other aspects of costs and issues are often not directly measurable, or the measurements are not available in most software project repositories. We acknowledge this limitation to our approach.

Second, we have only examined 18 open source projects from the Apache Software Foundation. We cannot guarantee that our approach will be as effective for projects with different cultures or conventions. We also cannot guarantee that the same observations achieved in this paper will hold for other projects. To mitigate this limitation, we selected projects of different programming languages (c/c++, Java, and Python) and from different domains. We believe that our dataset is representative of a diverse set of projects. However, we acknowledge that proprietary projects may have different cultures and convention from the open source projects. We plan to conduct case studies with our industrial collaborators in the future to further evaluate our approach.

Third, as discussed in section VII-C, the debt identification approach relies on history data since we mine the coupling data from the code repository, version control systems, and bug tracking systems. We need quality history data to identify how source files are revised due to bug fixing, and how source files change together over time. This is both a limitation and a potential threat to validity. If developers do not commit changes following good project practices, the quality of the identified debts will be compromised. For example, if developers frequently combine irrelevant changes into big commits the identified debts may be connected by "false positive" history coupling. Alternatively if developers making frequent trivial commits, our approach may miss some debts, as history coupling cannot be properly captured by the commits. We attempted to address these threats by only analyzing commits that affected between 2 to 30 source files. We believe that this mitigation helps us to eliminate trival changes as well as large changes that may not have a cohesive purpose. In our future work, we plan to employ pre-processing to untangle complicated commits and merge relevant commits to improve the quality of history coupling.

Fourth, we used bug-fixing churn as a proxy for maintenance costs. In open source projects, there is no good way to track actual costs, such as person-hours. Therefore, code churn has been commonly used as a surrogate for maintenance cost in previous studies [112]–[119]. Sjøberg et. al [112] found a significant correction between maintenance cost and code churn.

Their study suggested that, given the lack of real effort data, code churn is a reasonable surrogate for maintenance cost/effort in software development. While churn may not be a perfect proxy for measuring cost, this bias, if it indeed exists, should not significantly impact our debt identification. Our approach examines the *trajectory* of the churn over time, instead of focusing on a specific release. Thus we do not depend on the absolute value of churn. In future work, we plan to explore other cost measures. In addition, we plan to collaborate with industry partners to analyze commercial projects, where more accurate cost estimation may be available.

Fifth, *ATDs* are sub-optimal architectural design decisions that cause "extra" maintenance effort. The mere presence of architectural smell patterns does not qualify as *ATD* if not causing extra maintenance effort. In this study, we compare the difference between the percentage of maintenance cost associated with *ATDs* (in terms of error-fixing churn) with the size of identified *ATDs*. The size of *ATDs* is measured in both the percentage of associated files (Table II) and the percentage of LoC (Table III). This is to estimate the potential "extra cost" of *ATDs* as the additional portion of maintenance cost compared to their size. We acknowledge that this estimation does not capture the exact amount of "extra cost" caused by *ATDs*. However, there is no practical way to directly measure the exact extra cost. This method of estimation for the "extra cost" has been adopted in multiple previous studies [26], [31], [77], [120]. We acknowledge that the problem of measuring the interest of ATD is still open.

Lastly, we acknowledge that the regression models only quantify the trajectory of maintenance costs associate with each identified *ATD*. Based on the type of the trajectory, we made conjectures to interpret the interest rate associated with *ATDs*. That is, a linear regression model describes a debt with stable interest over time. A logarithmic model indicates flattening interest rate; while the exponential model indicates ever-increasing interest rate. And polynomial model indicates fluctuating interest rate. We acknowledge that the interpretation of each regression model instance might depend on many factors, such as project constraints, goals, and evolution. Our general interpretation of different regression model types has not taken those factors into considerations. We also acknowledge the limitation that we only used the regression models to predict the future cost of debts in the most recent future release (about 6 months of time). We have not evaluated whether the models can predict the costs of releases farther in the future. We conjecture that the accuracy of the prediction will decrease with the additional prediction distance. This applies to any model that predicts the future. For example, it is common sense that the weather forecast is more accurate for the next three days than for the next month. In addition, it is most important for practitioners to predict the cost of the most recent release to develop a practical task prioritization plan. In our future work, we plan to evaluate how far the models can predict future costs of debts.

X. Conclusion

This paper contributed an approach to automatically detect the precise locations of *ATDs*, to quantify the "interest" rate of each debt using four typical regression models, and to predict the cost of each debt in a future release using these models. Furthermore, our approach revealed the more complicated connections among different debt instances that deserve to be examined together for effective refactoring solutions.

We have evaluated the effectiveness of our approach in identifying and quantifying *ATDs* in 18 real-world projects of varying characteristics. First, our approach can identify significant *ATDs* in software projects that deserve attention. The identified debts usually only contain a small portion of a project's files, but account for a large portion of maintenance costs. Therefore, by focusing on the identified debts, the architects can potentially focus on cost-effective refactoring opportunities. The debts identified by our approach will keep incurring high maintenance costs in a project's future. Second, the regression models not only quantify the trajectory of past maintenance costs on each debt, but also accurately predict the cost of each debt in a future release. Therefore, architects can use such regression models to objectively prioritize *ATDs* based on estimates of future costs. Finally, the compound debt aggregation method in this approach can help architects focus on cost effective refactoring candidates that capture the connections among debt patterns.

We believe that this research represents a valuable and novel addition to the existing literature in *ATDs* identification, quantification, and prioritization.

XI. Acknowledgements

This work was supported in part by awards CNS-1823074, CNS-1823177, CNS-1823214, CCF-1817267, CCF-1816594, and OAC-1835292 from the National Science Foundation. We would also like to thank the anonymous reviewers for their valuable feedback, which helped us to improve this work.

References

[1] Ward Cunningham. The WyCash portfolio management system. In *Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 29–30, October 1992.

[2] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102:117–145, 2018.

[3] Nicolli SR Alves, Thiago S Mendes, Manoel G de Mendonça, Rodrigo O Spínola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[4] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *J. Syst. Softw.*, 101(C):193–220, March 2015.

[5] Terese Besker, Antonio Martini, and Jan Bosch. Impact of architectural technical debt on daily software development work—a survey of software practitioners. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 278–287. IEEE, 2017.

[6] Terese Besker, Antonio Martini, and Jan Bosch. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, 135:1–16, 2018.

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

[8] Peggy Skiada, Apostolos Ampatzoglou, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, and Ioannis Stamelos. Exploring the relationship between software modularity and technical debt. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 404–407. IEEE, 2018.

[9] Jeffrey C Carver, Jordi Cabot, Rafael Capilla, and Henry Muccini. Github, technical debt, code formatting, and more. *IEEE Software*, 34(2):105–107, 2017.

[10] Terese Besker, Antonio Martini, and Jan Bosch. Impact of architectural technical debt on daily software development work - a survey of software practitioners. 09 2017.

[11] Terese Besker, Antonio Martini, and Jan Bosch. Managing architectural technical debt: A unified model and systematic literature review. *J. Syst. Softw.*, 135:1–16, 2018.

[12] Alan MacCormack and Dan Sturtevant. Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, 120, 06 2016.

[13] Riccardo Roveda, Francesca Arcelli Fontana, Ilaria Pigazzini, and Marco Zanoni. Towards an architectural debt index. pages 408–416, 08 2018.

[14] Ran Mo, Joshua Garcia, Yuanfang Cai, and Nenad Medvidovic. Mapping architectural decay instances to dependency models, 2013.

[15] Isaac Griffith and Clemente Izurieta. Design pattern decay: The case for class grime. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 39:1–39:4, New York, NY, USA, 2014. ACM.

[16] Lawrence Peters. Technical debt: The ultimate antipattern - the biggest costs may be hidden, widespread, and long term, 2014.

[17] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. Estimating the principal of an application's technical debt. *IEEE Software*, 29(6):34–42, 2012.

[18] A. Martini and J. Bosch. The danger of architectural technical debt: Contagious debt and vicious circles. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 1–10, May 2015.

[19] Boris Pérez, Juan Pablo Brito, Hernán Astudillo, Darío Correal, Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman. Familiarity, causes and reactions of software practitioners to the presence of technical debt: A replicated study in the chilean software industry. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7. IEEE, 2019.

[20] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. How do software development teams manage technical debt?–an empirical study. *Journal of Systems and Software*, 120:195–218, 2016.

[21] Antonio Martini, Terese Besker, and Jan Bosch. Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, 163:42–61, 2018.

[22] Nicolli Rios, Rodrigo Oliveira Spinola, Manoel G de Mendonça Neto, and Carolyn Seaman. A study of factors that lead development teams to incur technical debt in software projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 429–436. IEEE, 2018.

[23] Nicolli Rios, Manoel G Mendonça, Carolyn Seaman, and Rodrigo O Spinola. Causes and effects of the presence of technical debt in agile software projects. 2019.

[24] Antonio Martini, Simon Vajda, Rajesh Vasa, Allan Jones, Mohamed Abdelrazek, John Grundy, and Jan Bosch. Technical debt interest assessment: from issues to project. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–6, 2017.

[25] Antonio Martini and Jan Bosch. The magnificent seven: towards a systematic estimation of technical debt interest. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–5, 2017.

[26] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proc. 15th Working IEEE/IFIP International Conference on Software Architecture*, May 2015.

[27] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36th International Conference on Software Engineering*, 2014.

[28] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[29] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.

[30] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proc. 33rd International Conference on Software Engineering*, pages 411–420, May 2011.

[31] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyevy, Volodymyr Fedaky, and Andriy Shapochkay. A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*, 2015.

[32] Robert Schwanke, Lu Xiao, and Yuanfang Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[33] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[34] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106, 2010.

[35] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. The evolution of technical debt in the apache ecosystem. In Antónia Lopes and Rogério de Lemos, editors, *Software Architecture*, pages 51–66. Springer International Publishing.

[36] Reem Alfayez, Pooyan Behnamghader, Kamonphop Srisopha, and Barry Boehm. An exploratory study on the influence of developers in technical debt. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, pages 1–10. Association for Computing Machinery.

[37] Md Abdullah Al Mamun, Antonio Martini, Miroslaw Staron, Christian Berger, and Jörgen Hansson. Evolution of technical debt : An exploratory study. volume 2476, pages 87–102. CEUR-WS.

[38] Michael Mohan, Des Greer, and Paul McMullan. Technical debt reduction using search based automated refactoring. 120:183–194.

[39] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163. ISSN: null.

[40] Angeliki-Agathi Tsintzira, Apostolos Ampatzoglou, Areti Ampatzoglou, Alexander Chatzigeorgiou, Oliviu Matei, D Department, and Holisun Srl. Technical debt quantification through metrics: An industrial validation. page 5.

[41] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE'19*, pages 2–11. ACM Press.

[42] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. The evolution of technical debt in the apache ecosystem. In Antónia Lopes and Rogério de Lemos, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 51–66. Springer International Publishing.

[43] Rodrigo O. Spínola, Nico Zazworka, Antonio Vetro, Forrest Shull, and Carolyn Seaman. Understanding automated and human-based technical debt identification approaches-a two-phase study. 25(1):5.

[44] Narayan Ramasubbu and Chris Kemerer. Integrating technical debt management and software quality management processes: a framework and field tests. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 883. Association for Computing Machinery.

[45] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. Exploring the costs of technical debt management – a case study. 21(1):159–182.

[46] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 315–326, 2016.

[47] Everton da S Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248. IEEE, 2017.

[48] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering*, 45(12):1211–1229, 2018.

[49] Ke Dai and Philippe Kruchten. Detecting technical debt through issue trackers. In *QuASoQ@ APSEC*, pages 59–65, 2017.

[50] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 179–188. IEEE, 2016.

[51] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451, 2018.

[52] Giancarlo Sierra, Ahmad Tahmid, Emad Shihab, and Nikolaos Tsantalis. Is self-admitted technical debt a good indicator of architectural divergences? In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 534–543. IEEE, 2019.

[53] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(3):1–45, 2019.

[54] Solomon Mensah, Jacky Keung, Jeffery Svajlenko, Kwabena Ebo Bennin, and Qing Mi. On the value of a prioritization scheme for resolving self-admitted technical debt. *Journal of Systems and Software*, 135:37–54, 2018.

[55] Yasutaka Kamei, Everton da S Maldonado, Emad Shihab, and Naoyasu Ubayashi. Using analytics to quantify interest of self-admitted technical debt. In *QuASoQ/TDA@ APSEC*, pages 68–71, 2016.

[56] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[57] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Was self-admitted technical debt removal a real removal? an in-depth perspective. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 526–536. IEEE, 2018.

[58] Stephany Bellomo, Robert L Nord, Ipek Ozkaya, and Mary Popeck. Got technical debt? surfacing elusive technical debt in issue trackers. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 327–338. IEEE, 2016.

[59] Terese Besker, Antonio Martini, Rumesh Edirisooriya Lokuge, Kelly Blincoe, and Jan Bosch. Embracing technical debt, from a startup company perspective. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 415–425. IEEE, 2018.

[60] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service-and microservice-based systems. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 125–133, 2018.

[61] Johannes Holvitie, Sherlock A Licorish, Rodrigo O Spínola, Sami Hyrynsalmi, Stephen G MacDonell, Thiago S Mendes, Jim Buchan, and Ville Leppänen. Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*, 96:141–160, 2018.

[62] Terese Besker, Antonio Martini, and Jan Bosch. Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 105–114, 2018.

[63] Terese Besker, Antonio Martini, and Jan Bosch. The pricey bill of technical debt: When and by whom will it be paid? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–23. IEEE, 2017.

[64] Woubshet Nema Behutiye, Pilar Rodríguez, Markku Oivo, and Ayşe Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139–158, 2017.

[65] Roberto Verdecchia, Ivano Malavolta, and Patricia Lago. Architectural technical debt identification: The research landscape. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 11–20. IEEE, 2018.

[66] Carlos Fernández-Sánchez, Juan Garbajosa, Agustín Yagüe, and Jennifer Perez. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. *Journal of Systems and Software*, 124:22–38, 2017.

[67] Francesca Arcelli Fontana, Riccardo Roveda, and Marco Zanoni. Technical debt indexes provided by tools: A preliminary discussion. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pages 28–31. IEEE, 2016.

[68] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. Technical debt prioritization: State of the art. a systematic literature review. *arXiv preprint arXiv:1904.12538*, 2019.

[69] Christoph Becker, Ruzanna Chitchyan, Stefanie Betz, and Curtis McCord. Trade-off decisions across time in technical debt management: a systematic literature review. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 85–94, 2018.

[70] Makrina Viola Kosti, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Georgios Pallas, Ioannis Stamelos, and Lefteris Angelis. Technical debt principal assessment through structural metrics. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 329–333. IEEE, 2017.

[71] Zadia Codabux and Byron J Williams. Technical debt prioritization using predictive analytics. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 704–706. IEEE, 2016.

[72] Theodoros Amanatidis, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Ioannis Stamelos. Who is producing more technical debt? a personalized assessment of td principal. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–8, 2017.

[73] John Brondum and Liming Zhu. Visualising architectural dependencies. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 7–14, Piscataway, NJ, USA, 2012. IEEE Press.

[74] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, April 1999.

[75] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 488–498, May 2016.

[76] 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*, pages 51–60, May 2015.

[77] Maleknaz Nayebi, Yuanfang Cai, Rick Kazman, Guenther Ruhe, Qiong Feng, Chris Carlson, and Francis Chew. A longitudinal study of identifying and paying down architecture debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 171–180. IEEE, 2019.

[78] Derek Reimanis, Clemente Izurieta, Rachael Luhr, Lu Xiao, Yuanfang Cai, and Gabe Rudy. A replication case study to measure the architectural quality of a commercial system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, 2014.

[79] Peter Caron. Creating and using quality software delivery measurements and metrics.

[80] Adam Tornhill. Prioritize technical debt in large-scale systems using codescene. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 59–60, 2018.

[81] Antonio Martini. Anacondebt: a tool to assess and track technical debt. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 55–56. IEEE, 2018.

[82] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285. IEEE, 2017.

[83] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In *European Conference on Software Architecture*, pages 320–335. Springer, 2018.

[84] Jason Lefever, Yuanfang Cai, Humberto Cervantes, Rick Kazman, and Hongzhou Fang. On the lack of consensus among technical debt detection tools. In *To Appear in 2021 International Conference in Software Engineering, Software Engineering in Practice*.

[85] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[86] Sunny Wong and Yuanfang Cai. Generalizing evolutionary coupling with stochastic dependencies. In *2011 26th IEEE/ACM International*

Conference on Automated Software Engineering (ASE 2011), pages 293–302. IEEE, 2011.

[87] Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave W Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212. IEEE, 2016.

[88] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128. IEEE, 2010.

[89] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In *IFIP International Conference on Open Source Systems*, pages 3–12. Springer, 2015.

[90] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, 2009.

[91] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 190–198. IEEE, 1998.

[92] Daihong Zhou, Yijian Wu, Lu Xiao, Yuanfang Cai, Xin Peng, Jinrong Fan, Lu Huang, and Heng Chen. Understanding evolutionary coupling by fine-grained co-change relationship analysis. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 271–282. IEEE, 2019.

[93] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420, 2011.

[94] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE, 2013.

[95] Robert Schwanke, Lu Xiao, and Yuanfang Cai. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 891–900. IEEE, 2013.

[96] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. On the use of line co-change for identifying crosscutting concern code. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 213–222. IEEE, 2006.

[97] Bram Adams, Zhen Ming Jiang, and Ahmed E Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 305–314, 2010.

[98] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 103–112. IEEE, 2013.

[99] Luciana L Silva, Marco Tulio Valente, Marcelo de A Maia, and Nicolas Anquetil. Developers' perception of co-change patterns: An empirical study. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 21–30. IEEE, 2015.

[100] Dirk Beyer. Co-change visualization. In *ICSM (Industrial and Tool Volume)*, pages 89–92, 2005.

[101] Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5):720–735, 2009.

[102] Dirk Beyer and Ahmed E Hassan. Animated visualization of software history using evolution storyboards. In *2006 13th Working Conference on Reverse Engineering*, pages 199–210. IEEE, 2006.

[103] Adam Vanya, Rahul Premraj, and Hans van Vliet. Interactive exploration of co-evolving software entities. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 260–263. IEEE, 2010.

[104] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering*, pages 563–572, May 2004.

[105] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proc. 17th Working Conference on Reverse Engineering*, pages 119–128, October 2010.

[106] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[107] Fabian Beck and Stephan Diehl. Evaluating the impact of software evolution on software clustering. In *Proc. 17th Working Conference on Reverse Engineering*, pages 99–108, October 2010.

[108] E. Kouroshfar. Studying the effect of co-change dispersion on software quality. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1450–1452, May 2013.

[109] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang 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*, MSR '15, pages 246–257, Piscataway, NJ, USA, 2015. IEEE Press.

[110] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Raţiu Daniel. Using concept analysis to detect co-change patterns. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 83–89, New York, NY, USA, 2007. ACM.

[111] Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.

[112] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2012.

[113] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software*, pages 23–31. IEEE, 2010.

[114] Bente CD Anda, Dag IK Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 2008.

[115] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.

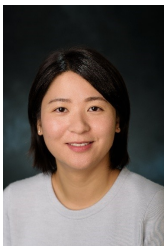
[116] Hong Wu, Lin Shi, Celia Chen, Qing Wang, and Barry Boehm. Maintenance effort estimation for open source software: A systematic literature review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 32–43. IEEE, 2016.

[117] Fabrizio Fioravanti and Paolo Nesi. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Transactions on software engineering*, 27(12):1062–1084, 2001.

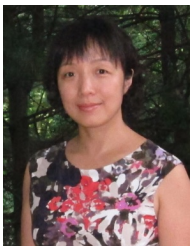
[118] Jane Huffman Hayes, Sandip C Patel, and Liming Zhao. A metrics-based software maintenance effort model. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 254–258. IEEE, 2004.

[119] Frank Niessink and Hans Van Vliet. Predicting maintenance effort with function points. In *1997 Proceedings International Conference on Software Maintenance*, pages 32–39. IEEE, 1997.

[120] Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and Lu Xiao. Towards an architecture-centric approach to security analysis. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230. IEEE, 2016.



Lu Xiao is an Assistant Professor in the School of Systems and Enterprises at Stevens Institute of Technology. Her research focues on software architecture, software evolution and maintenance. In particular, she is interested in modeling and analyzing software architecture and its evolution for addressing quality problems, such as maintenance quality and performance. She earned her PhD in Computer Science at Drexel University in 2016, advised by Dr. Yuanfang Cai.



Yuanfang Cai is currently a Professor at Drexel University, USA. Dr. Cai's research focuses on software design, software architecture, software evolution, and software economics. Her recent work investigates architecture issues that are the root cause of software defects, and the quantification of architectural debt. Dr. Cai is currently serving on program committees and organizing committees for multiple top conferences and the editorial board of top journals in the area of software engineering. The tools and technologies from Dr Cai's research have been licensed and adopted by multiple multinational corporations.

Rick Kazman is a Professor at the University of Hawaii and a Visiting Scientist at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method) and the Dali and Titan tools. He is the author of over 200 publications, and co-author of several books, including Software Architecture in Practice, Designing Software Architectures: A Practical Approach, Evaluating Software Architectures: Methods and Case Studies, and Ultra-Large-Scale Systems: The Software Challenge of the Future.

Ran Mo is an Associate Professor in the School of Computer Science at Central China Normal University. His research focuses on analyzing the quality of software based on software architecture. More specifically, how to measure software architecture in terms of maintainability? what are the architecture problems which incur high maintenance costs? when and how to fix these problems to reduce maintenance effort? He received the Ph.D degree in Computer Science from Drexel University in 2018, advised by Dr. Yuanfang Cai.

Qiong Feng got her Ph.D. degree from the Computer Science Department at Drexel University in 2019. Her research mainly focuses on analyzing the evolution of software architecture. She is now a software engineer at Wayfair Inc.