Constructing a Shared Infrastructure for Software Architecture Analysis and Maintenance

Joshua Garcia†, Mehdi Mirakhorli‡, Lu Xiao§, Yutong Zhao∥, Ibrahim Mujhid†, Khoi Pham‡, Rick Kazman§, Yuanfang Cai∥, and Nenad Medvidovič§

†University of California Irvine, {joshug4,malek}@uci.edu
‡Rochester Institute of Technology, {mxmvse,ijm9654,axoec}@rit.edu
§Stevens Institute of Technology, {lxiao6,yzhao102}@stevens.edu
∥University of Southern California, {khoipam,meno}@usc.edu
§University of Hawaii, kazman@hawaii.edu
^Drexel University, yfcai@cs.drexel.edu

Abstract—Over the past three decades software engineering researchers have produced a wide range of techniques and tools for understanding the architectures of large, complex systems. However, these have tended to be one-off research projects, and their idiosyncratic natures have hampered research collaboration, extension and combination of the tools, and technology transfer. The area of software architecture is rich with disjoint research and development infrastructures, and datasets that are either proprietary or captured in proprietary formats. This paper describes a concerted effort to reverse these trends. We have designed and implemented a flexible and extensible infrastructure (SAIN) with the goal of sharing, replicating, and advancing software architecture research. We have demonstrated that SAIN is capable of incorporating the constituent tools extracted from three independently developed, large, long-lived software architecture research environments. We discuss SAIN’s ambitious goals, the challenges we have faced in achieving those goals, the key decisions made in SAIN’s design and implementation, the lessons learned from our experience to date, and our ongoing and future work.

Index Terms—architecture analysis, maintenance, interoperability, reproducibility, reusability

I. INTRODUCTION

A software system’s architecture comprises the principal design decisions employed in the system’s construction and evolution [1]–[3]. Architecture is a key determinant of the system’s properties. While it is possible, for example, to make low-level design decisions for a system (e.g., the choice of a specific data structure), to implement the system carefully, and to test it thoroughly, none of those activities can mitigate inadequate architectural choices. Put simply, software systems “live and die” [4] by their architectures.

Despite this critical importance, the architectures of many systems are not explicitly documented. Instead, those architectures are reflected—actually, hidden—in the myriad system implementation details, posing significant challenges to the development, maintenance, and evolution of long-lived systems. In particular, the effort and cost of software maintenance dominate activities in a software system’s lifecycle [5]–[8]. Understanding and updating a system’s architecture is a critical facet of maintenance. The engineers of such a system must regularly [1] analyze the system to understand it, its architecture, and the implications of their planned changes; to do so, the engineers must somehow [2] recover the architecture from the system’s implementation in order enable the analysis, and determine how to best [3] represent the obtained architectural knowledge. Software engineering practice has shown this to be an exceptionally challenging problem, and engineers are forced to guess—and they very often actually ignore—the architectural implications of their choices and decisions.

To overcome this problem, for over the past two decades, software architecture research has yielded many different tools and techniques [9]. However, empirical studies and technology transfer are impeded by disjoint research and development environments, lack of a shared infrastructure, high initial costs associated with developing and/or integrating robust tools, and a dearth of datasets. The resulting one-off solutions inhibit further advances in software architecture research, delaying or preventing systematic synthesis and empirical validation of new or existing techniques. As a result, researchers and practitioners in need of cutting-edge tools tend to re-invent, re-implement research infrastructure, or ignore particular research avenues altogether. In doing so, they repeat each other’s efforts as well as mistakes, so that opportunities for potential breakthroughs are often missed and the field is replete with solutions that do not work as advertised and/or are not interoperable.

To address these challenges, we propose Software Architecture INstrument (SAIN), a first-of-its-kind framework for assembling tools in support of architecture-based software maintenance. SAIN’s capabilities have been motivated by directly engaging the software researcher and practitioner communities, in the form of three workshops as well as a survey conducted by the authors. SAIN is delivered as a web-based platform consisting of three principal components: [1] a catalogued library of cutting-edge tools for reverse engineering and analyzing software systems’ architectures; these tools are either provided by their original authors or reproduced from literature; [2] a plug-and-play instrument for integrating the tools and techniques to facilitate empirical studies of software architectures; and [3] reproducibility wizards to set up experiment templates, produce replication packages, and release them in easy-to-run and modify formats.

SAIN aims to facilitate empirical studies as well as development of new architecture analysis and maintenance solutions. By providing an extensible repository of architectural artifacts for non-trivial software systems, a major goal of SAIN is to enable researchers to establish a shared understanding of the relative accuracy of different techniques, to identify the gaps and sources of inaccuracy, and to develop new solutions to continually improve results. SAIN provides researchers with commonly needed data structures to represent architectural artifacts and
algorithms for conducting a wide range of analyses, thereby enabling our community to build on each others work and to reduce the re-development of commonly needed capabilities. SAIN also has the potential to impact the practice. Over time, it will provide practitioners with an authoritative source where they can obtain and try out various tools, provide feedback, contribute to the repository of architectural artifacts, and influence the research in this area. Similarly, the benchmark results, made available through SAIN’s portal, will help the practitioners determine which tools are suitable for obtaining architectural information for their systems.

The key contributions of this paper are as follows:

- We introduce a SAIN, a framework that comprises a library of cutting-edge tools for architecture recovery and analysis, a plug-and-play instrument for integrating tools, and reproducibility wizards to support replication of architecture-based research studies.

- We discuss our experience and our users’ experiences of SAIN in terms of the three tool suites currently contributed to SAIN: 13 architecture recovery components, 8 components for computing architectural metrics or analyses, 2 fact extractors, and 9 utility components from those tool suites; one compact case study of SAIN run on a game engine project called Mage and another detailed case study of SAIN run on Hadoop 2.5.0; and the empirical results of the detailed case study, which analyzes the relationships between architectural smells, architectural tactics, and error-proneness.

- We discuss experimental results from our detailed case study that are summarized into 5 major findings that can aid architects with maintainability by focusing on a small set of architectural elements that involve error-prone modules, architectural tactics, and architectural smells.

- We make SAIN publicly available for researchers and practitioners at [10].

Section [II] covers SAIN’s foundational concepts. Section [III] discusses the requirements elicitation process for SAIN and the key challenges it aims to overcome. Section [IV] discusses SAIN’s key design principles and alternatives considered. Section [V] details our experience to date; Section [VI] summarizes our lessons learned; and then our paper concludes.

II. BACKGROUND AND FOUNDATION

To set the stage for subsequent discussion, we introduce key concepts framing software architecture, recovery, and analysis.

A. Architectural Decay

As software evolves, a major challenge impeding its successful maintenance is architectural decay [2], [11], where changes made to a system in the course of maintenance and evolution actually violate the system’s intended architecture. The effects of decay include increased time and effort required to perform maintenance tasks and introduction of architectural defects (e.g., a system unable to interface with outside agents due to conflicting assumptions about network protocols).

As an example of architectural decay, consider Bash [12], a widely used Unix shell. Bash’s conceptual architecture is depicted in Figure [13]. Its as-implemented architecture [14], shown in Figure [15], shows noticeable decay: not only do the components differ, but there are many dependencies that are unaccounted for in the conceptual architecture.

Decay has been reported in the architectures of a number of widely-used software systems [14], [15]. Recent studies have increasingly showcased the urgent need to address architectural decay. A study surveying over 1,800 software engineers and architects found architectural decay to be the greatest source of technical debt [16], and to be highly correlated with bugs and additional maintenance effort [17], [18].

B. Software Architecture Recovery

Reverse-engineering an architecture from implementation artifacts is referred to as architecture recovery [19]–[21]. Multiple architectural views [2], [22] of a system may be desirable, depending on the objective of recovery. For instance, a runtime view may be appropriate for reasoning about a system’s security, performance, and availability [23], [24], while different structural and/or behavioral views, obtained either automatically [20], [21], [22], [23], [25]–[33], [33], [34], [34], [34]–[36] or with the aid of analysis tools [37]–[48], may allow reasoning about the implications of a range of system changes. Thus, different recovery techniques may be needed for different architectural analyses [49]–[51]. Having ready access to multiple recovery techniques directly motivated SAIN.

For illustration, consider the four architectural views of Bash in Figure [2]. Figure [2a] is the as-implemented architecture from Figure [1b] redrawn in a circular layout. The other three views were each obtained from a different automated recovery technique. Figure [2b] uses information retrieval to create a semantic architectural view, while Figures [2c] and [2d] depict different structural views. Each of the four views may be useful for different architectural tasks. For example, a structural view may be more effective when considering system reconfiguration; a semantic view may be better suited for understanding the system concerns. Prior work has suggested a way of integrating multiple architectural views [52].

C. Architectural Analyses

Once an architecture is recovered from code-level artifacts, a variety of analyses and subsequent activities are made possible: identifying or predicting instances of architectural decay; repairing the architecture to eliminate decay; optimizing it to achieve desired quality attributes; and so on. We highlight a body of analyses that has inspired SAIN most directly.

A prominent activity for tracking architectural decay in software systems is architectural smell detection [53], [54]. An architectural “smell” is a design decision that negatively impacts a system’s maintenance and evolution. Potential adoption of existing techniques for detecting [17], [55]–[57] and, subsequently, repairing [58]–[62] architectural smells is hampered by the lack of readily reusable recovery techniques and architectural benchmarks (e.g., architectural models that can serve as “ground truths”) on which their efficacy can be evaluated and subsequent improvements measured.

Recently, evolutionary architectural analyses have been performed across multiple versions of existing software systems. These studies include assessing the nature and extent of architectural change [63] and decay [64], identifying the

Fig. 1: Architectures of Bash. The architectures are depicted at this magnification only as a way of visually comparing them; the reader is not expected to understand their details.
correlation between co-occurring changes across architectural modules and implementation defects \[65\], and attempting to predict architectural decay \[66\]. However, as before, the lack of available architectural benchmarks and “out of the box” recovery techniques restricts the scope and architectural phenomena studied, and renders each study one-of-a-kind.

III. SAIN’S REQUIREMENTS AND CHALLENGES

SAIN is motivated by challenges that are faced by the software engineering research community, and frequently discussed at conferences, workshops, and in the research literature. More specifically, the requirements for SAIN were directly elicited from the software architecture and software engineering research communities.

To elicit requirements for SAIN, we utilized two complementary techniques: requirements elicitation at brainstorming workshops, and an online survey. We organized three invitation-only, focused workshops that involved around 50 researchers and practitioners from the software architecture and empirical software engineering areas. The workshop attendees were guided through discussions of opportunities, challenges, and community needs for the area of software architecture. Furthermore, the participants brainstormed features and use cases of SAIN as well as ways to address the challenges and community needs. Through these workshops, we solicited and specified 17 requirements for SAIN.

We further asked the community to help us prioritize these requirements through an online survey—which was filled out by 60 members of the research community. These requirements involved creating a repository of benchmarks and datasets (e.g., machine-readable architectural models) and tools (e.g., tools that extract implementation-level information or architectural metrics), and the kinds of user interfaces and utilities SAIN would provide to the research community (e.g., reusable experiment templates or visualization capabilities). Ultimately, these various requirements involved re-occurring and time-consuming research prototyping challenges that can be potentially automated or outsourced as engineering tasks. Satisfying these requirements would facilitate and speed up research breakthroughs and productivity for many research groups working in the areas of software architecture, maintenance, and empirical software engineering.

The resulting requirements obtained from these workshops and the survey fall under five key challenges faced by the software engineering community, when conducting architecture-oriented research centered on software maintenance and evolution. In this paper, we focus on the three challenges that we have prioritized for the current version of SAIN.

C1 – Research Tool Accessibility and Reusability. Implementations of research techniques are often unavailable, defective, not easily accessible, or no longer supported by their original creators. For tools that do work, it is common for them to not operate as advertised, requiring major effort to adapt these tools for further research.

C2 – Interoperability of Tools. Software architecture research and technology transfer is hampered by dispersed research environments and stove-piped solutions emerging from different research groups. This, in turn, inhibits research advances, makes it difficult to synthesize techniques and tools in novel ways, and complicates comparisons of research solutions. Researchers and practitioners in need of cutting-edge architectural analyses must often recreate tools or their major elements, including basic code analysis, reverse-engineering functions, and frameworks. Furthermore, different assumptions that these tools make (e.g., about the execution environments, formats used, implementation languages, etc.) prevent their combined use, further inhibiting breakthroughs.

C3 – Reproducibility of Experiments and Analyses. Due to inaccessible, non-reusable, or defective tools, datasets, and case studies, and incompatible underlying tool assumptions, it is difficult to reproduce the results of many previous software architecture-oriented research studies \[67\], \[68\]. For software architecture-oriented research, it is often necessary to construct previous tools and datasets entirely from scratch if end implementations of research techniques are often unavailable, defective, not easily accessible, or no longer supported by their original creators. For tools that do work, it is common for them to not operate as advertised, requiring major effort to adapt these tools for further research.

A. SAIN’s Design Principles

Various design principles were considered and architectural alternatives analyzed to identify a design that could adequately address the needs and challenges identified through SAIN’s requirements elicitation effort. SAIN’s core design principle is based on a plug-and-play architecture to enable tool accessibility and reusability (C1), interoperability of tools...
(C2), and reproducibility of experimental templates (C3). Specifically, components added to SAIN that respect a standard interface can easily interoperate with other SAIN components for novel experiments; entire experiment workflows can be saved, modified, and shared; and SAIN allows for search and navigation of tools of interest for researchers who wish to reuse or access tools or their constituent components.

To facilitate ease of composing a new experimental pipeline using existing SAIN tools or their constituent components (C3), SAIN incorporates a plug-and-play solution based on components that respect a standard interface expected by SAIN and provision of wrappers or converters to address disparate languages or data formats. This solution allows users to upload an executable format of an existing tool or its constituent components into SAIN and have it ready for integration with other tools or components which, in turn, addresses C2. Upon importing a tool or component, an SAIN user needs to use SAIN to specify the tool’s or component’s interface, parameters, and specific execution commands. By adhering to such a standard, SAIN can execute the tool or component. This integration solution relies on interface compatibility, however, since each tool or component may be developed by different researchers using different languages and formats, especially for novel research prototypes, SAIN allows users to upload and incorporate components that act as wrappers or converters, enabling integration of novel tools and components with existing SAIN components (i.e., C2).

SAIN experimental pipelines utilize a pipe-and-filter architectural style that helps combine components in some experiments that involve sequential processing of the information. Furthermore, SAIN uses a blackboard architectural style in cases where a sequential order cannot be defined. This architectural style allows components to communicate through a shared data model. The use of these two styles enable flexible experiment workflows to be designed, saved, reused, and shared—which aids in addressing C3.

SAIN’s design also enables a drag-and-drop mechanism that users can leverage to easily compose new experiment pipelines by dragging a component from SAIN’s component catalog and dropping it onto the integration environment’s canvas. This simplicity of use and access directly supports overcoming C1. The SAIN UI relies on a graphical programming language that allows creation of workflows which can be used by SAIN to compose components and generate a fully executable pipeline in the back end, which helps to address C3. This graphical programming language-based UI is depicted in Figure 3.

B. Prototyped SAIN Design Alternatives

To evaluate various alternatives brainstormed by the team during joint application design sessions, we implemented a prototype of the architecture to examine five design alternatives early on and assess the risks. The first two solutions we assessed but did not adopt are based on Google’s Blockly [75].

For the visual programming language-based solution, each tool contributed to SAIN is represented as a graphical block or node in SAIN’s front end. On the back end, each tool is represented as a Node.js API service. The prototype of this solution was successful at addressing requirements related to all three major challenges C1, C2, and C3. The visual aspect of the approach, which is similar to end-user programming solutions, simplified quick experimentation with ease of tool reuse and access (C1), while still allowing complex component integration (i.e., C2) and sophisticated experiments (i.e., C3). Therefore, this design alternative was chosen and SAIN is delivered as a web-based platform that can be used for quick experimentation by even novice users and new researchers.

The prototype of the microservice-based design included using typical microservice solutions, i.e., containers and exposure of tool interfaces using HTTP. This solution enables standalone reuse of tools and their constituent components, allowing researchers to easily run each tool or component within a Docker container on their local machines. To enable integration of contributed tools in SAIN as microservices, our visual programming language is used to allow users to compose experiments without dealing with technical difficulties.

The final alternative we considered was a hybrid pipe-and-filter and blackboard architectural style compared to a simpler publish-subscribe style. Although the publish-subscribe style would enable a highly flexible architecture for experiment templates, the highly general interfaces of such a style were unsuitable for the more specific and controlled interfaces needed to contribute tools to SAIN. Additionally, a pipe-and-filter style more naturally modeled the kinds of pipeline-like workflows used in empirical software engineering-oriented experiments. On the other hand, the blackboard style enabled a user to have flexible integration of partial solutions to form an experiment in which components could communicate or independently act by reading and writing data in a global shared store. This design is particularly suitable for SAIN as complex experiments may not necessarily have a deterministic pipeline and might be composed of various experimental fragments. The hybrid pipe-and-filter and blackboard style is well-suited for enabling various forms of interactions needed to create complex experiments in which tool integration can be process-centric or data-centric.

C. SAIN’s Library of Architecture Recovery and Analysis Tools

Through the three workshops and online survey discussed in Section III, SAIN’s requirements focused on four different types of tools: it must support to enable tool reuse and accessibility (C1) and tool interoperability (C2); tools for architecture recovery; architectural analysis and metrics; fact extractors; and utilities. These types of tool are selectable in the SAIN visual programming language-based UI depicted in Figure 3. Specifically, the pane on the left side of Figure 3 shows four groups of tools selectable by a user that can be dragged-and-dropped onto the canvas of the window to produce experiment
workflows. We discuss each of these tools and their importance further in the following paragraphs.

Architecture recovery tools obtain architectural abstractions of a system based on implementation-level entities. Given that such tools aim to directly determine an architecture to overcome the pervasive problem of architectural decay, having such tools were critical for SAIN in addressing C1.

Tools for computing metrics related to architecture recovery and analysis were deemed highly important and discussed extensively in SAIN workshops and the online survey. Participants of the workshops and survey pointed to the need to use standard metrics and easily reuse tools to measure architectures (e.g., compute metrics about architectural smells) and compare architectures of implemented systems from various domains (e.g., metrics for comparing a recovered architecture against a ground-truth architecture [28]).

Fact extractors are used to obtain raw facts about a software system. Examples of such raw facts include dependencies between software modules, system and package dependency graphs, change requests from issue-tracking repositories, architectural metrics, etc. There was extensive discussion in workshops about how simply having fact extractors that are accessible and reusable would facilitate and speed up empirical research in software architecture on its own—especially since many fact extractors often need to be re-implemented to serve as raw materials for creating novel experiments.

Utilities are tools that provide “helper” functionality, such as data-format conversion and statistical analysis that may not be architectural in nature on their own but are critical for interoperability of tools, i.e., C2. For example, different architecture recovery techniques can sometimes use different data formats as input for representing raw facts. In SAIN, an example utility is a tool for uploading projects from different sources (e.g., a GitHub repo or a program directory) or a generic data-mining tool like Mallet [76] that might be reused in some studies.

D. SAIN features: Reproducibility Wizards

To address the key challenge of realizing reproducible experiments and analysis (C3), three key reproducibility wizard features have been implemented in SAIN: experimental workflow composition, reusable experimental templates, and easy assembly of replication packages.

To enable novel and reproducible experiment templates in SAIN, we provided features for construction of workflows involving SAIN artifacts and datasets. Combining artifacts into workflows facilitates running new experiments or reproducing previous ones. For example, Figure 3 shows the workflow of the seven components from three tool suites Titan, Archie, and ARCADE: Sdsm, Hdsm, Bug Space, Tactic Detection, ACDCWithSmellDetection, Tactic RootCover, and Smell RootCover. This new experiment template enables the integration of architectural roots of error-proneness, architectural tactic implementation, and architectural smell analysis, leading to new and valuable findings which otherwise are not available. This template intuitively illustrates the workflow of using the seven components to identify the architectural roots of error-proneness and their association with architectural tactics in a software project. The experiment rationale and details will be introduced in Section V. The point here is that, following the flow in this template, an analyst can easily reproduce this experiment, by first executing Sdsm, TacticDetection, and ACDCWithSmellDetection. The intermediate output of Sdsm, Bug Space, Hdsm, and Tactic Detection are used as input to Tactic RootCover; and the intermediate output of ACDCWithSmellDetection and Tactic RootCover are used as input to Smell RootCover.

Beside the ability to specify new experiment workflows, we have included a number of predefined, commonly employed workflows to serve as templates. Users can easily reuse and
revise these experimental pipelines. Currently, we have released six templates focusing on reverse engineering different architectural views, detecting architectural smells, and investigating the relationships between smells and software quality issues.

One of the key features of the instrument is to allow researchers to easily assemble their experiment setup using SAIN and a save menu, export it as a self-contained replication package (available on the top-left File menu in Figure 3). Storage of experiment templates or workflows using this feature allows for sharing the exact experiment structure used by a researcher—enabling researchers to easily understand and modify existing experiments to produce novel experiments to achieve new breakthroughs in software architecture research.

V. EXPERIENCE WITH SAIN

To convey our experience of constructing SAIN, we discuss the tool suites and components it currently contains, the experience of the initial users of SAIN, and some SAIN experiments conducted so far. We further present a compact case study of architectural smell detection using SAIN on the game engine project, Mage [77], and a detailed case study of SAIN on Hadoop 2.5.0 [78], a large and widely used framework for distributed processing of large datasets across cluster computers. The latter case study combines various components from three different tool suites incorporated into SAIN, the benefits and challenges provided by SAIN in that context, and novel empirical results obtained from it.

A. Current SAIN Tool Suites

SAIN has been populated with components from three separate tool suites that support architecture recovery and analysis, have been used in a variety of empirical studies, and are publicly available: Titan, a tool suite that extracts representations called Design Rule Spaces (DRSpaces) that bridges the gap between architecture and defect prediction [52], [57], [79], [80]; Archie, a tool suite that automates the creation and maintenance of architecturally-relevant trace links between code, architectural decisions, architectural tactics, and related requirements [23], [81]–[84]; and Architecture Recovery, Change, And Decay Evaluator (ARCADE), a tool suite that employs a collection of architecture-recovery techniques and a set of metrics for measuring different aspects of architectural change [28], [55], [63], [85].

Archie: Tactic Detection. Archie [23], [86] is a reverse engineering method that detects architectural tactics. It detects security tactics, such as audit, authenticate, HMAC, Secure Session Management, and RBAC; reliability tactics, such as heartbeat and CheckPoint; and performance tactics, such as Resource Pooling, Resource Scheduling, and Asynchronous Invocation [84]. Archie leverages machine learning and structural analysis techniques to identify tactics and map them to code snippets, classes, or source files.

Titan is a tool suite for bridging the gap between software architecture and maintenance quality [79], built upon the design rule theory proposed by Baldwin and Clark [87]. It captures the architecture of a software system as multiple, overlapping design spaces, called Design Rule Spaces (DRSpaces). Each DRSpace is composed of a leading file, which is the design rule of the space, and a set of member files that structurally depend on the leading file, directly or indirectly. In addition, Titan also models the history coupling between source files—how frequently they change together in revision commits—as an additional layer of architectural connections. Titan can identify and rank the DRSpaces in a project which aggregate the error-prone files—thus these DRSpaces are called the Architectural Roots (ArchRoots), which deserve attention from practitioners interested in addressing the long-term maintenance quality of a project.

ARCADE: Smell Detection. ARCADE’s smell-detection component can identify architectural smells that contribute to maintenance difficulties in a project [55], [64], [88]. The definition of a subset of those architectural smells, which are focused later in this section, and their potential negative impacts are described below: 1) Dependency Cycle occurs when a set of components (e.g. classes or source files) whose links form a circular chain, causing changes to propagate from one component to another on the chain. Such high coupling between components violates design principles for modularity. 2) Link Overload manifests when a component has interfaces involved in an excessive number of links (e.g. procedure-call dependencies), affecting the system’s separation of concerns and isolation of change. 3) Concern Overload occurs when a component implements an excessive number of concerns, violating the principle of separation of concerns, potentially increasing the size of a component and reducing its maintainability.

B. Current SAIN Components

SAIN’s components are divided into the four types described in Section IV-C: architecture recovery, architectural analysis and metrics, fact extractors, and utilities. These components may be part of the web-based integration capability or reproducibility wizards of SAIN, available in the form of microservices, or as their original source or binaries. As of the writing of this paper, SAIN contains 13 components for architecture recovery, 5 components for architectural analysis and metrics, 2 fact extraction components, and 4 utility components available as part of SAIN’s web-based integration environment—from three different tool suites. 6 architecture recovery techniques, 8 architectural analyses and metrics, 2 fact extractors, and 9 utility components are available as microservices. These components allow for recovery of other components, architectural tactics, and DRSpaces; architectural analysis and measurement of architectural tactics, architectural smells, defects, change-proneness, etc.; fact extractors for structural dependencies, natural language processing-based information, issue repository extraction, etc.; and utilities for data-format manipulation, visualization, etc.

SAIN includes extensive documentation describing the purpose of each individual component from a tool suite, its inputs, outputs, and links to publications describing the tool suite further. Our users have so far found the documents ease the burden of understanding each individual component, as opposed to the tool suite as a whole, or even standalone tools of each tool suite. SAIN’s design that forces tool authors to utilize a standardized form of documentation requiring descriptions of inputs and outputs eased user understanding of SAIN components.

The variety of components available in the form of visual integration mechanisms, microservices, or individually downloadable component source or binaries has also allowed multiple students to use tools from outside their research group with greater ease and a shorter learning curve. More specifically, several research groups have re-used SAIN components that
originate outside of their own research group for novel experiments. We elaborate on two of these experiments in Sections V-C and V-D.

C. An Experiment to Identify Architectural Smells

To conduct research to identify architectural smells in open-source projects that use or implement AI/ML, we leveraged the tool integration module of SAIN. Figure 4 shows the SAIN experiment template for this study. We leveraged components that were already deployed in SAIN to design an experiment which takes a project jar file as input and produces a CSV file that lists architectural smells in each module of the project. First, we used the Dependency Builder component that is part of ARCADE to extract the dependencies in a given project. Then, we used ACDC [29], a widely used architecture recovery technique available in SAIN, to discover clusters that follow patterns commonly observed in decompositions of software systems and recover module views of a software system's architecture. We included the Smell Detection component of ARCADE, which takes the outputs of the Dependency Builder and ACDC as input and generates an XML file that lists the identified architectural smells. Finally, the Smell Analyzer component is used to deserialize the output of the smell detector and generate a CSV file that lists class- and component-level smells. These aforementioned four components were connected using the simple and user-friendly drag-and-drop tool integration interface of SAIN and the designed experiment was saved as a JSON file for later use.

The tool integration module of SAIN allows users to export and import JSON files created for experiments. The smell detection experiment was imported to identify the architectural smells in Mage [77], a game engine project. Mage was downloaded as a zip file and provided as input to the first component in the experiment. It took us around two minutes to extract various types of component- and class-level architectural smells in the Mage project. The output of the components at each step of the experiment was visualized in the component panel and generated log files were accessible through a back-end terminal output window to trace and debug any issues throughout the experiment process—which is accessible through the “>” UI element in the upper-right region of Figures 3 and 2. Using the import and export features of SAIN, it was possible to (1) repeat a previous experiment with the same input or re-run it for additional projects and (2) share the experiment JSON file with other members of the research team to reproduce our experiment results at any time. The rich and user-friendly experimentation environment of SAIN helps make the results of scientific experiments reproducible and supports research transparency.

D. Integrating SAIN Components

To further showcase our experience with SAIN, we elaborate on a preliminary case study accomplished by integrating SAIN components, which were originally produced by different research teams. This case study shows how SAIN can help researchers achieve a result whose sum is greater than its parts in software architecture analysis with flexible and versatile functions while obtaining interesting, novel research findings.

Our case study subject is an open-source project, Hadoop, which is actively and widely used and maintained. We analyze Hadoop by integrating the insights from the three different tool suites, (i.e. ARCADE, Archie, and Titan) currently available in SAIN. We investigate Hadoop version 2.5.0, since it is a major release, and has been previously analyzed by all three tool suites. Figure 5 depicts this experiment as realized in SAIN.

1) Integrating ArchRoots and Architectural Tactics: We began our experiment by integrating Titan’s architectural root detection and Archie’s architectural tactic detection (TacticDetection in Figure 3) to enable more advance analyses.

Integration Motivation and Rationale: We aim to integrate the insights of ArchRoots and architectural tactics. To this end, we can achieve a multi-perspective view of ArchRoots from their architecture design structure, error-proneness, and involvement in tactic implementation. This view helps us answer questions such as the following: How much are the architectural tactics associated with error-proneness? How are tactic files and error-prone files architecturally connected to each other?

To answer these questions, we aim to identify DRSpaces that are led by tactic files using the ArchRoots detection component. This component exhaustively searches for all the DRSpaces that are led by each and every source file in a system as the leading file. This tends to identify large spaces that have the largest coverage on error-prone files. However, these spaces do not necessarily have a focus on tactics. While in this integration case study, Titan only searches for the DRSpaces that are led by tactic files. Therefore, the key aim of our study guides us as to how and to what extent the architectural tactics are associated with error-proneness by focusing on source files that are impacted by the tactic files as their “design rules”. We refer to the ArchRoots associated with tactics from the integration study as Tactic-ArchRoots, extracted by the Tactic RootCover component in Figure 3.

Our study’s results are illustrated in Figure 5. The x-axis shows the ranking of the top x ArchRoots detected by Titan. The y-axis shows the coverage of the top x ArchRoots to the Error space (i.e. the set of error-prone files with at least 5 bug fixes in the revision history for Hadoop-2.5.0). The rectangular data points represent the original ArchRoots; while the diamond-shaped data points represent the Tactic-ArchRoots. We make the following observations from this result:

Finding 1: The tactic implementation is non-trivially associated with the error-proneness in Hadoop-2.5.0—38% of the error-prone files are aggregated in design spaces that are led by the tactic files. Therefore, it is important for practitioners to investigate the error-proneness of a project from the perspective of tactic implementation.

The top five ArchRoots, considering DRSpaces led by any source file, can cover 80% of files in Error. The implication is that error-prone files are significantly linked to each other through their architectural connections. This is consistent with previous findings [57]. While the maximal coverage of the Error space by the Tactic-ArchRoots reaches up to 38% with a total of 30 Tactic-ArchRoots. This large coverage of error-prone files by Tactic-ArchRoots indicates that error-proneness of Hadoop-2.5.0 is non-trivially associated with the tactic files. Practitioners should examine this association throughout their architectural tactics. This view helps us answer questions such as the following: How much are the architectural tactics associated with error-proneness? How are tactic files and error-prone files architecturally connected to each other?
implementation. In Table I we list the characteristics of the top
five Tactic-ArchRoots. The first column shows the tactic leading
files of the identified Tactic-ArchRoots. The second and the
third columns show the frequency and ranking of each tactic
leading file for fixing bugs. The last two columns describe the
characteristics of the Tactic-ArchRoots. In terms of BSC and
DSB measurements. BSC is the percentage of files in Error
that are covered in a root; DSB is the percentage of files in a
root that are from Error. Table I shows that the top five tactic
leading files could be very error-prone: UserGroupInformation,
CommonConfigurationKeysPublic, and FileContext are highly
ranked for their bug fixing frequency. These Tactic-ArchRoots
have a high concentration of error-prone files in Hadoop-2.5.0.
For example, the DSB of the ArchRoot led by FileContext
reaches up to 54%, indicating every one in two files in this
root contain more than five bug fixes. Such insights would not
be available without the integrated analysis of Titan and Archie.

<table>
<thead>
<tr>
<th>Leading Tactic File</th>
<th>Root Info</th>
<th>Tactic File</th>
<th>B. Freq</th>
<th>B. Rank</th>
<th>BSC</th>
<th>DSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserGroupInformation</td>
<td>31</td>
<td>3</td>
<td>15%</td>
<td>55%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommonConfigurationKeysPublic</td>
<td>14</td>
<td>12</td>
<td>11%</td>
<td>45%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MiniDFSCluster</td>
<td>2</td>
<td>128</td>
<td>7%</td>
<td>19%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FileContext</td>
<td>13</td>
<td>15</td>
<td>6%</td>
<td>54%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token</td>
<td>2</td>
<td>208</td>
<td>8%</td>
<td>32%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE I: Top Five Tactic ArchRoots

Finding 2: The top five Tactic-ArchRoots deserves special
attention from the developers, since they strongly relate
to the error-prone files—i.e., 19% to 54% of files in
each Tactic-ArchRoot is error-prone as measured using
DSB. The top five Tactic-ArchRoots can provide a useful
perspective for examining the association between tactic
implementation and error-proneness in a project.

2) Integrating Tactic-ArchRoots and ARCADE Smell De-
tection: For the next step of our case study, we integrated
ARCADE’s architectural smell detection with Tactic-ArchRoots.
In other words, we aim to investigate whether and to what
extent the most error-prone Tactic-ArchRoots from the above
integration also suffer from architectural smells. This integra-
tion of architectural smells and Tactic-ArchRoots is realized
by Smell RootCover, which takes ACDCTWithSmellDetection,
the smell detector of ARCADE based on ACDC, and Tactic
RootCover as input—all of which are depicted in Figure 2.
In Hadoop-2.5.0, we identified three types of smells: Dependency
Cycle (1 instance), Link Overload (5 instances), and Concern
Overload (1 instance). The architectural smells detected by
ARCADE can have negative impacts on the maintainability
and software quality, which in turn can increase the error-rate
of the components involved in the smells. For instance, if one
of the components involved in a Dependency Cycle contains
an error, a change fixing the error can propagate changes to
other components in the smell.

Integration Motivation and Rationale: Practitioners can
gain valuable insights by viewing Tactic-ArchRoots and the
architectural smells in combination. In particular, for our case
study, we are interested in answering the following question:
How are the source files in the top five Tactic-ArchRoots
involved in the architectural smells? This can potentially help
developers to reveal the underlying architectural design flaws
that lead to high error-proneness of Tactic-ArchRoots.

The integration rationale is that, for Tactic-ArchRoots,
we investigate how each instance of an architectural smell,
detected by ARCADE, is contained in the roots. Note that
an architectural smell instance is usually composed of a group
of source files. For example, we identified a Dependency Cycle
formed by 78 source files in Hadoop-2.5.0. A Tactic-Root alone
may not contain all the files of a smell. Thus, we calculate
the percentage of files in each architectural smell instance that are
also contained in a Tactic-ArchRoots. In addition, we calculate
the percentage of files in each architectural smell instance that
are contained and aggregated in the top x Tactic-ArchRoots.

Table II presents the overview of the integration analysis of
combining Tactic-ArchRoots and architectural smells.

Finding 3: Overall, among a total of seven instances
of architectural smells in Hadoop-2.5.0, five instances
are involved in the Tactic-ArchRoots. This indicates that
reviewing the Tactic-ArchRoots is important to investigate
most (5/7) architectural smells in Hadoop-2.5.0.

The detailed analysis of each involved architectural instance
is shown as a column in Table II. When considering all the
TABLE II: Smells Identified by ARCADE in Tactic-ArchRoots

<table>
<thead>
<tr>
<th>Tactic Leading File</th>
<th>ARCADE Smell Instance (% Files)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dp. Cycle-1 (# 78)</td>
</tr>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td>UserGroupInformation</td>
<td>20%</td>
</tr>
<tr>
<td>CommonConfigurationKeysPublic</td>
<td>3%</td>
</tr>
<tr>
<td>MiniDFSCluster</td>
<td>0%</td>
</tr>
<tr>
<td>FileContext</td>
<td>1%</td>
</tr>
<tr>
<td>Token</td>
<td>32%</td>
</tr>
<tr>
<td>All Tactic-ArchRoots</td>
<td>38%</td>
</tr>
</tbody>
</table>

identified Tactic-ArchRoots (a total of 30 roots as shown in Figure 5), they contain 6% to 100% of the files in different architectural smell instances, as shown in the last row. This indicates that different architectural smells have different levels of associations with Tactic-ArchRoots. In addition, if we focus on the top five Tactic-ArchRoots, we notice that the maximal percentage (6% to 100%) of files have already been covered for each smell instance.

**Finding 4:** These results indicate that developers only need to focus on the top five Tactic-ArchRoots for understanding how the architectural smells overlap with the tactic implementation and their error-proneness.

To illustrate in greater depth the interesting results that can be obtained by using SAIN to integrate various architectural tools, we present a qualitative example and visualization to show how combining Tactic-ArchRoots and smell analysis helps developers understand the root causes of error-proneness of the top ranked Tactic-ArchRoots. Figure 6 is a part of the Design Structure Matrix (DSM) visualization of the top ranked Tactic-ArchRoot led by tactic file UserGroupInformation. Due to space limitations, we only illustrated part of the space that focuses on the tactic implementation for Authenticate, which ensures that a user or a remote system is who it claims to be.

A DSM is an $n \times n$ square matrix, which represents the relationship among source files in a system. As shown in Figure 6, the rows and the columns represent the source files from the Tactic-ArchRoot led by UserGroupInformation (this leading file is listed in row 1). The relationship among files is captured in the $n \times n$ square matrix—found in the outer rectangular box on the right of Figure 6. Titan captures two types of relationship between files in the DSM: (1) structural dependencies, including “ext” and “dp”—where “ext” indicates that the file on the row extends the file on the column, while “dp” represents all other general types of reference relationships, such as method call and variable declarations; (2) the historical coupling, captured as a numeric value, indicating the number of times the file on the row changes together with the file on the column in the same commits. For example, cell[3,2] says “dp,36”, indicating the file on row 3 ipc.Cline depends on the file on row 2 ipc.Server, and they change together 36 times in the same commits. This indicates strong coupling between ipc.Server and ipc.Client. To make reading Figure 6 more intuitive, we color-coded the cells based on the weight of the historical coupling between files, where darker shades of red indicate a larger number of co-occurring commits. The DSM visualization helps us to gain insights regarding both the structural and historical coupling among files in a system.

Using the basic DSM, we integrated three additional aspects of information for each involved source file in the space: (1) the involved tactic(s) (labelled as column “Tactic”); (2) the involved smell(s) (labelled as column “Smell”); and finally, (3) the error change frequency (labelled as column “E-Freq”), which is color-coded in a heat-map based on the value—darker shades of red indicate a higher error change frequency. This additional information helps us to investigate how different tactic files are coupled with each other both structurally and historically, and how they are involved in architectural smells which, in turn, provide insights regarding the root cause of the error-proneness of Tactic-ArchRoots.

For example, through this integrated DSM visualization in Figure 6 we have the following overall finding:

**Finding 5:** The complicated structural and historical coupling among the tactic files tend to contribute to the error-proneness of Tactic-ArchRoot-1 in Hadoop 2.5.0. There appears to be little-to-no relationship between error-proneness and architectural smells for Tactic-ArchRoot-1 in Hadoop 2.5.0.

VI. DISCUSSION AND LESSONS LEARNED

To realize the current version of SAIN, our team of several developers and research groups faced major development challenges. More specifically, we have a geographically distributed team across three different continents and 13 different time zones. We found that agile methods with two-weeks sprints, joint application design (JAD) session [89], and exploratory prototyping of design alternatives worked effectively to develop SAIN under these circumstances.

For many tools or components in SAIN, a variety of data types are used, from general types such as XML and JSON to specific types for sub-domains of architecture research, such as the Rigi Standard Format [29], [88], [90]–[92] for clustering-based architecture recovery. The long-standing problem of data conversion has required the construction of new utility components that act as adaptors or wrappers. Nevertheless, we have found that building these conversion tools has not been a major pain point for researchers using SAIN compared to the high variability of formats. Once these utility components or connectors are built, they can be easily contributed back to SAIN.

As a result, we aim to include support mechanisms to aid documentation of datasets, benchmarks, and their metadata—in a similar manner as we have done for tools and their components—which itself has already eased the burden of interoperability. We believe that this challenge further emphasizes the need for architecture researchers to address problems of disparate data types, possibly through flexible languages that can describe current architectural phenomena with mechanisms allowing for incorporation of future phenomena. To that end, extensible architectural languages such as ACME [93], [94] or xADL [95], [96] may be a promising starting point.

Each SAIN component can take a wide variety of input options or complex configuration files. Incorporating these components into our plug-and-play integration tool or creating microservices out of them aided in determining the best default options or the key options for components of a tool suite.
Fig. 6: Tactic-ArchRoot-1 led by UserGroupInformation with Smells. Numbers along the diagonal refer to the ID of each file.

The visual nature of our plug-and-play integration mechanism made it easy to identify the key input options that users must supply (e.g., a zipped directory), without even having to look extensively at existing documentation, which could be ampler for some of the tool suites. As a result, our experience encourages wider use of visual or block-based paradigms for creating novel experiments, re-using them, or sharing them.

Moreover, we found that students, developers, and researchers could easily try out and combine various components from tool suites they never tried before. Although this did not completely eliminate integration challenges (e.g., the need to create new data conversion components or modify existing components), SAIN made it easier to identify these issues for novel experiments that users wished to run.

The plug-and-play integration panel provided by SAIN allows a novice researcher to quickly become familiar with the workflow of different tool suites. For example, the integration experiment was driven by a third year Ph.D. student who had no prior experience with Archie or ARCADE, and only had very limited experience with Titan components. He was able to accomplish the integration case study in a time frame of two weeks. He finally ended up contributing two new analysis components, built upon existing components. This would not be possible without the support of SAIN.

Due to the experience described above with our plug-and-play mechanisms, composition of experiments was significantly eased. Other challenges remained, however. For instance, debugging an error in experiment can be more challenging due to a SAIN user being unable to set breakpoints and step through a program to diagnose or fix a bug. Running on a remote machine (e.g., SAIN server), as opposed to a local machine can create unexpected delays or slowdowns. Nevertheless, SAIN developers have managed to overcome many of these new issues by providing research prototype interoperability mechanisms, allowing various users across several research groups to more easily and quickly learn and use architecture-oriented tools and components from outside their respective groups.

The SAIN platform provided a comprehensive view of different architectural instruments that are available. It allows the researchers to think out-of-the-box about the potential connections and integration opportunities among different components that were initially developed by independent teams. These connections and opportunities only became explicit and available when different architectural instruments were organized and reviewed together.

VII. CONCLUSION

Over three decades of software engineering research aimed at tackling the problem of architectural decay has resulted in a plethora of techniques and tools to address the problem. Researchers attempting to address this long-standing architecture problem face enormous challenges behind tool reuse and accessibility, tool interoperability, and reproducibility of experiments and analyses using these tools. To address these three major challenges, we have constructed SAIN, a first-of-its-kind framework for assembling tools to support architecture-based software maintenance. SAIN comprises a library of cutting-edge tools for architecture recovery and analysis, a plug-and-play instrument for integrating tools, and reproducibility wizards to support replication of architecture-based research studies. We make SAIN publicly available for researchers and practitioners at

We have discussed our experience of SAIN and our users' experiences of SAIN in terms of the three tool suites: 13 architecture recovery components, 8 components for computing architectural metrics or analyses, 2 fact extractors, and 9 utility components; one compact case study and a detailed case study of our users running novel experiments using SAIN and how it eased the process for them; and the results of the detailed case study, which analyzes the relationships between architectural smells, architectural tactics, and error-proneness. This detailed study resulted in 5 major findings that can aid architects interested in improving maintainability of their systems by simply focusing on a small set of Tactic-ArchRoots.

It is an open challenge to determine how to provide mechanisms that (1) ease dataset and benchmark inclusion and integration into an experiment and (2) microservice creation for research prototypes or their components. We, therefore, aim to study mechanisms for specifying and integrating datasets and benchmarks into our plug-and-play mechanisms and reproducibility wizard. Although full automation of microservice or containerization is desirable, a highly valuable first step is to design interfaces and supporting software mechanisms that reduces the manual labor needed to create a container for a research prototype or one of its components. We aim for our future work to overcome this challenge.

VIII. ACKNOWLEDGEMENTS

This work was supported in part by awards CNS-1823262, CNS-1823246, CNS-1823074, CNS-1823354, CNS-1823214, CNS-1823177, CNS-1823074, CCF-1823177, OAC-1835292, CCF-1816594, and CCF-1717963 from the National Science Foundation and the U.S. Office of Naval Research under grant N00014-17-1-2896. We would also like to thank the anonymous reviewers for their valuable feedback, which helped us to improve this work.


