Impact of Architectural Smells on Software Performance: an Exploratory Study

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
arcelli@disco.unimib.it

Matteo Camilli
Politecnico di Milano
Milano, Italy
matteo.camilli@polimi.it

Davide Rendina
University of Milano-Bicocca
Milano, Italy
d.rendina2@campus.unimib.it

Andrei Gabriel Taraboi
University of Milano-Bicocca
Milano, Italy
a.taraboi@campus.unimib.it

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

ABSTRACT

Architectural smells have been studied in the literature looking at several aspects, such as their impact on maintainability as a source of architectural debt, their correlations with code smells, and their evolution in the history of complex projects. The goal of this paper is to extend the study of architectural smells from a different perspective. We focus our attention on software performance, and we aim to quantify the impact of architectural smells as support to explain the root causes of system performance hindrances. Our method consists of a study design matching the occurrence of architectural smells with performance metrics. We exploit state-of-the-art tools for architectural smell detection, software performance profiling, and testing the systems under analysis. The removal of architectural smells generates new versions of systems from which we derive some observations on design changes improving/worsening performance metrics. Our experimentation considers two complex open-source projects, and results show that the detection and removal of two common types of architectural smells yield lower response time (up to 47%) with a large effect size, i.e., for 50%-90% of the hotspot methods. The median memory consumption is also lower (up to 20%) with a large effect size for all the services.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software design techniques**.

KEYWORDS

Software Architecture, Architectural Smells, Software Performance

1 INTRODUCTION

Software performance has been recently recognized as the *new correctness* [17] since poor performance (e.g., large battery consumption, long execution time, low throughput, high utilization of resources, etc.) prevents the usage of applications and implies the failure of large and costly projects [9, 22]. Explaining the performance characteristics of software systems is challenging, it is not trivial to understand which constituent elements (and their associated interactions) are responsible for performance issues [6, 21, 42, 43]. An opportunity in this direction is represented by Architectural Smells [3, 14, 15, 28] (AS) due to their key characteristics of being indicators of poor design choices, later impacting the quality of software systems. In the literature, the presence of AS has been recognized to have a strong influence on a variegate set of system qualities, such as understandability, changeability, usability, extensibility, and reusability [13, 41].

The investigation of AS as contributors to (poor) software performance represents the subject of this paper, and we found that this aspect is much less investigated in the literature. Few attempts can be found in [24, 48], and these approaches mainly target microservice systems. There are some efforts devoted to the identification of the design elements as root causes of software performance problems [20, 25, 29, 40]. However, to the best of our knowledge, there is no exploratory study on the correlation between AS and software performance characteristics. The goal of this paper is to fill this gap by investigating if AS have an impact on system performance, in particular, we are interested to study the correlation between the performance variations of a system and the absence/presence of AS as indicators of better or worse system performance.

We select, from a catalog of twelve AS [3], three different types of AS that we found more relevant to our goal of studying the impact on software performance. Specifically, we focus on *God Class* (GC), *Cyclic Dependency* (CD), and *Hub-Like Dependency* (HL). God Class smell is acknowledged as a problem of not well balancing the load among the available system resources. The problem is that GC typically violates the modularity principle [26], i.e., most of the application business is managed by a single or few components that inevitably become the bottleneck in software performance. This smell is called in different ways, according to the detection level, i.e., if the smell is detected at the class or package level. For example, God Class can be seen as an alias for Insufficient Modularization, as discussed by Suryanarayana et al. [39], and it is an indicator of an architectural refactoring opportunity as outlined by Sousa et al. [10]. Cyclic Dependency and Hub-Like Dependency smells are instead both symptoms of dependency issues. Dependencies are of great importance in software performance since the software components that are highly coupled and with a high number of dependencies are typically more critical since they have larger interactions [3]. Besides, CD has been recognized in the literature as one of the most common smells in the developers' opinion [27].

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The detection of the AS is performed with state-of-the-art tools, i.e., Arcan [12] and Designite [36]. Performance analysis of applications is performed with well-known profiling tools, i.e., JProfiler [30] and Kieker [31]. The contribution of this paper is the subsequent analysis devoted to quantifying the impact of AS on the system performance metrics (i.e., CPU utilization and memory consumption). We evaluate the software performance of the original system (including instances of the architectural smell types mentioned above) and we compare it with modified versions of the system from which we remove some smell instances.

We perform our analysis on two open-source and real-world projects: `OpenMRS` [32] and `TeaStore` [32]. Since the detection of Hub-Like Dependency smell found very few instances of this kind of smell, we focus our refactoring and analysis on the other two smells. Our results demonstrate that GC and CD have a substantial impact on the considered performance metrics. According to our experimental campaign, we obtained the following main findings:

- Refactoring GC and CD smell instances leads to a decrease of the execution time at the method level up to 42% and 47%, respectively.
- The median execution time measured in the original system version is generally higher compared to the versions obtained after refactoring. When refactoring CD instances, we observed large effect size for 50%-80% methods. When refactoring GC instances, we observed large effect size for 70%-90% methods.
- Refactoring of GC and CD smell instances leads to a decrease of the memory consumption at the service level up to 16% and 20%, respectively, with a large effect size for all services.

To summarize, we can assess the architectural smells as promising means to support software engineers in the task of understanding which design choices may contribute to good or poor performance characteristics of software systems.

The sequel of the paper is organized as follows. Section 2 introduces related work. Section 3 presents an overview of the study design including research questions, while Section 4 delves into the relevant details of our case studies and tools used in our study design pipeline. Architectural smells detection and their subsequent refactoring are presented in Section 5. The design of our experiments, major results, lessons learned, and threats to validity are discussed in Section 6. Final remarks and future work are reported in Section 7. Replication package is publicly available at https://doi.org/10.5281/zenodo.7552885.

2 RELATED WORK

Our work mainly relates to two streams of research, i.e., AS and software performance that are briefly reviewed in the following. This paper advances the state-of-the-art in the attempt of adopting AS for understanding their impact on the system performance. We make use of automated static analysis and performance monitoring tools, hence we also discuss related work adopting such tools.

Architectural smells. The AS are surface manifestations of strategic decisions concerning software architecture that negatively impact the quality of a software system. Different (commercial and open-source) tools are currently available for architectural smell detection. Recently researchers studied AS according to different perspectives. Sas et al. [35] analyze how AS evolve in industrial embedded systems and provide insights on the effects of AS on the long-term maintainability and evolvability of the systems. Sas et al. [34] investigate the relation between source code changes and AS. In particular, they study whether the frequency and size of changes are correlated with the presence of a selected set of AS. They found that change frequency increases after the introduction of a smell and the size of changes are also larger in smelly artifacts. Arcelli et al. [11] analyze whether AS are independent or can be derived from a code smell or from one category of them. They observe that AS are independent from code smells, and therefore deserve special attention by researchers interested in investigating their actual harmfulness. Sharma et al. [37] investigate correlation, collocation, and causation relationships between architecture and design smells. They found that AS are highly correlated with design smells. Overall, researchers have studied various aspects of AS, but to the best of our knowledge, the impact of AS on performance issues has not been explored yet.

Software performance. It is a pervasive quality of software systems, and it is difficult to understand which components contribute to (poor) performance since many aspects are involved: the design, the implementation code, and the execution environment [23]. The focus of this paper is on the analysis of design choices, in fact, we make use of AS that recognize bad practices impacting software performance. There are some approaches in the literature that aim to identify the architectural elements representing the root causes of performance problems. For instance, Ibidunmoye et al. [20] survey the works targeting the performance of computing systems through anomaly detection and bottleneck identification. More recently, Liu et al. [24] present an empirical study on the correlations between runtime performance deficiencies and bad smells, however, the selected smells are specific to microservice systems and performance metrics relate to Kubernetes nodes only. Zhong et al. [48] investigate the possible impacts, causes, and solutions of AS, however, the smells relate to microservice systems also in this study, and architects in the industry acknowledge the difficulty of recognizing performance-related problems in the early stages of software development. Avritzer et al. [2] exploit software performance antipatterns to support continuous integration, delivery, and deployment pipelines, but it is the load testing that mainly drives the occurrence of performance issues. To summarize, to the best of our knowledge, there is no comprehensive study on AS quantifying their impact on software performance, as we do in this paper.

Automated static analysis tools. We use Arcan [12] and Designite [36] tools to detect architectural smells. Herold [19] also uses Arcan, however, the goal is to investigate the association between three types of AS and the existence of architecture-violating dependencies as manifestations of architectural degradation. Sharma et al. [8] exploit Designite to detect AS, but they empirically explore the degree of adherence of AS occurrences to the Pareto principle. They observe that a software development team optimizes the refactoring efforts focusing on a few components and attaining the maximum benefits.

Performance monitoring. To monitor software performance, we use JProfiler [30] and Kieker [18] tools. Velez et al. [44] adopt JProfiler to measure performance at the method level, but their goal is to parameterize performance models that are built to evaluate

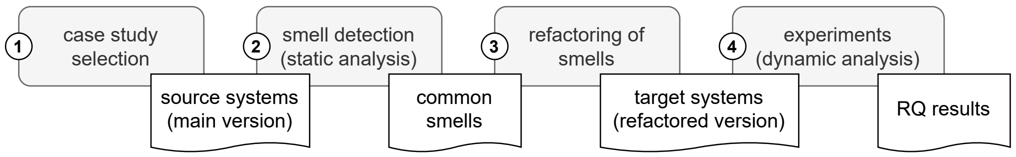

1
case study selection
2
smell detection (static analysis)
3
refactoring of smells
4
experiments (dynamic analysis)
source systems (main version)
common smells
target systems (refactored version)
RQ results

Figure 1: Study design pipeline

configurable systems. Reichelt et al. [33] focus on performance analysis at the code level, and they refer to KIEKER as a monitoring framework that is able to detect small performance changes.

3 STUDY DESIGN

3.1 Research questions

The major goal of our research is to understand the extent to which common AS affect the system performance characteristics. Thus, we investigate the following research questions.

RQ1: *What is the quantitative impact of common architectural smells on execution time?*
The motivation behind this question is that execution time is considered fundamental in software systems [5, 7]. There are several examples of project disposal and financial loss due to responsiveness problems [38].

RQ2: *What is the quantitative impact of common architectural smells on memory consumption?*
The rationale behind this question is that memory is a symptom of resources usage, hence, as the available memory may fluctuate meanwhile the system is running, it is indeed relevant to assist practitioners in examining memory usage and identifying optimization opportunities [4].

3.2 Pipeline overview

As illustrated in Fig. 1, the study design adopts a pipeline composed of four steps: ① case study selection; ② architectural smell detection (i.e., static analysis); ③ refactoring of smells; and then ④ experiments (i.e., dynamic analysis).

The workflow is as follows. Starting from the scope defined by our research questions RQ1-RQ2, we first ① selected existing open source representative benchmarks: a monolithic system following a service-oriented architecture called `OpenMRS` [32], and a well-known microservices system benchmark called `TeaStore` [45]. After collecting the latest versions of these two benchmarks, henceforth referred to as original (or main) versions, we ② statically analyzed the codebase to detect the presence of possible AS. In this stage, we leverage existing off-the-shelf detection tools: ARCAN [12] and DESIGNITE [36]. We identified GC and CD as the smells associated with the highest detection rate. The rate associated with the smell HL follows GC and CD, however, the total number of actual instances in this latter category is rather small. Since the static analysis identified GC and CD as the most common smells, we focused on these two smells in the next stages of the pipeline. After the detection, we ③ manually inspected the source code of the two systems to refactor some of the found smell instances. In particular, we took into account all GC and CD smells, and, for each occurrence, we analyzed the source code to check the actual existence. For true positive occurrences, we applied refactoring actions to the source code, thus removing the corresponding instances. This specific activity has been carried out following the best practices defined in [46]. Interestingly, we found that the removal of one smell may produce the nice effect of solving further detected smells. This means that solving n smells can produce the solution of m (with $m > n$) AS. As future work, we are interested in further investigating those refactorings that solve multiple AS. The outcome of the third stage is a set of target systems. Each target system has been obtained from the original one by removing as many architectural smell occurrences as possible in each category (i.e., GC and CD). Both original and target versions of the two selected case studies have been analyzed dynamically in our ④ experimental campaign, that is, the final stage of our pipeline. We designed and conducted a number of experiments to analyze the usage of resources (both CPU and memory) in the original and target versions of our case studies. The raw measurements were collected to carry out a pairwise comparison between the original and target software systems versions, with the ultimate goal of answering the research questions.

Table 1: Summary of selected case studies

Project	Version	#LoC	#Packages	#Classes
`OpenMRS`[1]	2.5.3	196, 573	90	1, 168
`TeaStore`[2]	1.4.1	11, 243	38	115

4 CASE STUDIES AND TOOLS

4.1 Case studies

We select two different well-known open-source service-based systems commonly adopted as benchmarks in the software performance engineering research community [32, 45]. Table 1 lists the two case studies along with the information about the specific versions we adopted, the size of the projects in terms of the number of Lines of Code (LoC), packages, and classes. The first case study is a medical record system, namely `OpenMRS`, which consists of a service-based monolith architecture. The second analyzed project is called `TeaStore`, and it is an open-source e-commerce system showing a microservices architecture.

4.1.1 OpenMRS. This case study is a customizable electronic Medical Record System (MRS). The mission of `OpenMRS` is to improve healthcare delivery in resource-constrained environments by coordinating a global community that creates a robust, scalable, user-driven, open-source medical record system platform. To achieve this

[1]https://github.com/openmrs/openmrs-core
[2]https://github.com/DescartesResearch/TeaStore

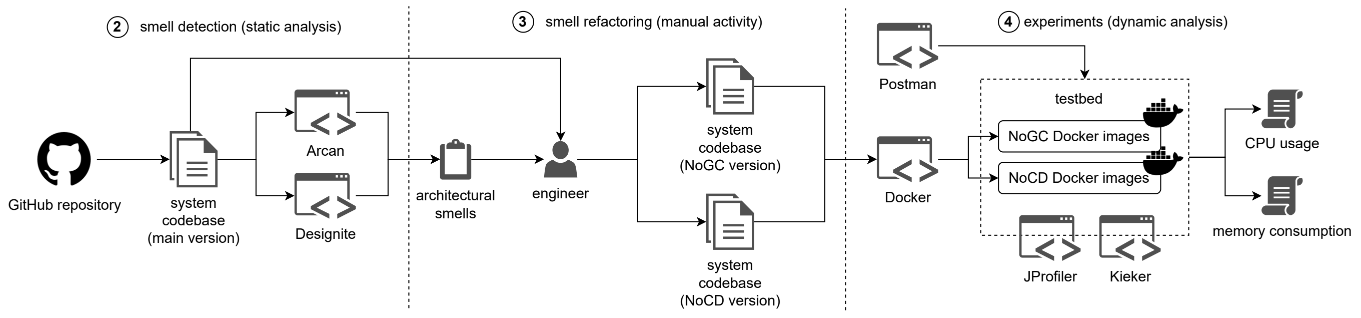

2 smell detection (static analysis)
3 smell refactoring (manual activity)
4 experiments (dynamic analysis)
GitHub repository
system codebase (main version)
Arcan
Designite
architectural smells
engineer
system codebase (NoGC version)
system codebase (NoCD version)
Postman
Docker
testbed
NoGC Docker images
NoCD Docker images
JProfiler
Kieker
CPU usage
memory consumption

Figure 2: Pipeline zoom in (stages 2-4)

goal, the system adopts a modular architecture, where the additional components can be attached to the core one called `OpenMRSCore`. The core module implements the main business logic of the whole system using Apache Tomcat[3]. It stores/retrieves the data into/from a MySQL[4] relational database, using mainstream solutions to map project entities into tables. In addition to the core module, the REST module of the system exposes RESTful APIs that we used to carry out end-to-end tests of the major functions. The system is equipped with pre-loaded datasets describing dummy patients and their medical records.

4.1.2 *TeaStore.*

This case study adopts a microservices architectural style and implements six main microservices interacting via lightweight communication mechanisms using RESTful APIs and common technology stacks (e.g., Docker containers[5], Netflix Ribbon client-side load-balancer[6]). The main microservices are:

- *Image Provider Service.* It provides all the images related to the products (i.e., tea varieties).
- *WebUI Service.* It exposes the functions used by the web UI and acts as an API gateway for the external actors interacting with the application.
- *Authentication Service.* It implements authentication and access control mechanisms.
- *Recommender Service.* Given the history of orders, it recommends available products users may like.
- *Persistence Provider Service.* It is responsible for data persistence handling the CRUD (i.e., Create, Read, Update, and Delete) operations executed onto the database.
- *Registry Service.* It acts as a dynamic registry for all the other services to avoid thigh coupling between them.

4.2 Tools

Figure 2 illustrates a more detailed view of the main steps of our pipeline. In particular, it zooms into the following stages: ② smell detection, ③ smell refactoring, and ④ experiments. According to Fig. 2, we need different tools to build these stages. We categorize the tools into: *static analysis* (AS detection) and *dynamic analysis* (deployment, testing, and profiling tools). In the following, we list and briefly describe all these tools.

[3]https://tomcat.apache.org/
[4]https://www.mysql.com/
[5]https://docs.docker.com/
[6]https://github.com/Netflix/ribbon

4.2.1 *Static Analysis.*

We used Arcan [12] and Designite [36] tools to perform the static analysis of the codebases and detect AS.

The tool Arcan[7] has been used to detect CD and HL smells. Arcan supports different programming languages such as Java, Python and C#. We decided to use this tool because it supports code inspection and it also provides a smart visualization of the relationship between class objects, i.e., an intuitive and easy-to-understand graph representation. This feature indeed simplifies the identification of possible refactoring steps to remove the detected smell instances. The tool additionally allows the smells to be tracked over multiple software versions (i.e., the so-called feature *Number of Smells Over Time*). We took advantage of this latter feature, especially while analyzing and refactoring `OpenMRS` since for this case study we detected more than 500 smell instances. A comprehensive description of Arcan detection strategies can be found in [12, 34].

As shown in Fig. 2, we complement the results obtained through Arcan by using Designite[8] to detect the GC smell.

It is worth noting that the version we used of Designite for Java programs (i.e., to analyze the codebases of our systems) does not support the detection of GC smells directly, while it detects the smell *Insufficient Modularization.* According to the work presented in [39], the smell is defined as "*an abstraction that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both*". This statement is consistent with the original definition of GC, hence this AS can be seen as an alias for Insufficient Modularization, as outlined in [39]. In our study, we consider them as synonyms and we henceforth refer to both of them as GC smells.

4.2.2 *Dynamic analysis.*

According to Fig. 2, to conduct the experiments stage of the pipeline, we used Docker to build and run multiple software images (i.e., all the considered versions of the two selected systems) onto our testbed. We used both JProfiler [30] and Kieker [31] profiling tools to monitor and collect information during the experiments we executed by means of Postman[9].

Docker. Docker is an open-source project that automates the process of deploying applications within software containers, providing additional abstraction through virtualization at the operating system level. To build, deploy, and run all the software images of

[7]https://docs.arcan.tech/2.8.0/
[8]https://www.designite-tools.com/features/
[9]https://www.postman.com/

Table 2: Smell detection and refactoring statistics

Smells	GC		CD		HL	
	OpenMRS	TeaStore	OpenMRS	TeaStore	OpenMRS	TeaStore
Detected	133	2	541	11	2	2
Refactored	31	2	26	8	0	1
Tot. refactored	33		34		1	

our target systems (OpenMRS and TeaStore), we used the *docker-compose* configuration files that are available in the original codebase of the two systems.

JProfiler. JProfiler [30] is a profiling tool that has been selected due to several reasons. First, it collects all the information we need about CPU and memory. Concerning memory consumption, we collect metrics on *live and dead objects* allocated in the memory and cleaned by the garbage collector. JProfiler traces CPU hotspots at the method level and provides the user with an easy-to-use comparison function for the recorded snapshots. These features are of key relevance for measuring the extent to which changes introduced by smell refactoring affect system performance. JProfiler is well integrated with Docker, in fact, it can be automatically associated to running software images. We use JProfiler to record (i) live and dead objects during all the experiments executed with both case studies; and (ii) CPU hotspots during all the experiments executed with OpenMRS. To trace hotspots of TeaStore, instead, we use the Kieker profiling tool, hereafter more details are reported.

Kieker. Kieker [18] is a Java-based performance monitoring and dynamic software analysis framework which enables performance monitoring at the method level for the CPU hotspots. We used Kieker to monitor all the versions of TeaStore since the original codebase is equipped with Kieker instrumentation that is already implemented. During the deployment of TeaStore, we selectively activated the performance monitoring only on the microservices in which AS instances were found. In this case, we used JProfiler to monitor the memory consumption (live and dead objects), and Kieker to monitor the CPU hotspots of interest.

Postman. Postman[10] is a platform that can be adopted to build and generate calls to REST services. As described above, the two selected service-based systems expose RESTful APIs to external actors. Thus, we used Postman to generate valid API calls that have been recorded into Python scripts. We then run the scripts to launch multiple end-to-end (e2e) tests to the API gateway running onto our testbed. The e2e tests have been executed locally to avoid latency issues and reduce the risk of obtaining unreliable results. The result of the e2e tests is aimed to compare the different project versions, the goal is to analyze performance fluctuations introduced by refactoring the AS.

5 SMELL DETECTION AND REFACTORING

5.1 Architectural smells detection

As described in Sec. 3.2, to answer our research questions, we carried out a pairwise performance comparison between the original version and the refactored versions of the two case studies.

[10]https://www.postman.com/api-documentation-tool/

Table 2 summarizes the number of detected smells by using Arcan and Designite per category and each case study. We can observe that the number of smells detected in the codebase of OpenMRS is generally higher compared to TeaStore. The difference is mainly due to: (1) the size of the projects, and (2) the aging factors. Indeed, TeaStore is a relatively smaller project compared to OpenMRS which is instead bigger and more legacy. The data in Table 2 shows that the highest number of detected instances in OpenMRS belongs to the GC category. The highest number of detected instances in TeaStore is instead found in the CD category. The HL smell yields a low number of instances. As anticipated in Sec. 3.2, based on these results, we decide to focus our study on GC and CD since they are the most common smells. As opposite, the number of HL instances prevents a follow-up study on this AS since our two selected case studies include few instances and there is not a wide margin of refactoring actions.

5.2 Architectural smells refactoring

According to Fig. 2, the smell refactoring stage is a manual activity carried out by the authors based on common best practices and guidelines described below. The number of detected smells suggests that the complexity of refactoring the two systems is indeed different since OpenMRS includes a much larger number of instances compared to TeaStore. Table 2 shows the number of smell instances that have been removed after applying refactoring actions on the original codebase of the two systems. Concerning OpenMRS we remove 31 smell instances for GC and 26 occurrences of CD that represent 23% (31/133) and roughly 5% (26/541) of the detected AS types, respectively. Concerning TeaStore both Arcan and Designate detect a smaller number of smells, and the refactoring applies to all instances for GC and 72% (8/11) for CD. Overall, we consider a total number of refactorings that is very similar, i.e., 33 for GC and 34 for CD. It is worth noting that in TeaStore, the AS have been detected in a subset of microservices, i.e., the ImageProvider and PersistenceProvider services. In this case, we profiled and monitored only these two components during the experiments.

The refactoring activity has been conducted following the guidelines of Suryanarayana et al. [39], consisting of a refactoring catalog for many AS including both GC and CD.

Refactoring of GC smells. Concerning GC smells, we mainly applied two different strategies according to the following conditions. **Extract Class**. Some of the classes in the two projects show a large number of attributes. This often affects the number of implemented methods as well as the size of the class itself. In this case, we extract one or more new classes by moving related attributes and methods to obtain possibly better separation of concerns. In OpenMRS this represents the hardest strategy since refactoring actions in this category are often disruptive. Indeed, modifying "god" classes (e.g., Hibernate configuration classes in OpenMRS) causes a lot of additional changes in all the other dependent classes. This means that the modification is not isolated, there are several classes involved in this type of refactoring. We applied the *extract class* strategy also in the case of large utility classes to get smaller classes and achieve possibly better separation of concerns. By applying this strategy, we removed 9 GC smell instances in total, i.e., 27% (9/33) of the implemented refactorings.

Split Service. In some cases, the classes implementing the business logic of (micro)services serve a lot of different client types, hence limiting the separation of concerns. To avoid this scenario, we split one service into multiple services directly modifying the existing DAO and Hibernate configuration files for accessing the database. This means that requests of different types are re-directed to specific (micro)services, instead of being managed by a unique resource that inevitably becomes the system bottleneck. Applying this technique, we removed 24 GC smell instances in total, i.e., 73% (24/33) of the implemented refactorings.

Refactoring of CD smells. Concerning CD smells, we mainly considered three different strategies as follows.
Encapsulation. In some cases, the projects show classes used as data containers only. An example in `OpenMRS` is represented by the class `AllergyReaction` used only by the class `Allergy` as a data structure to maintain the reactions to a specific allergy. The refactoring strategy adopted in this case is the *encapsulation*, and it leads to removing 7 CD smell instances. Indeed, the principle of encapsulation advocates the separation of concerns and information hiding. Hiding implementation details and variations are two techniques that enable the effective application of this refactoring.
Remove Middle Man. The projects show classes with a number of methods that simply delegate the tasks to other components. We handled this situation by using the *Remove Middle Man* refactoring strategy, which means removing these methods and forcing the client to call directly the methods that are responsible for the execution of certain tasks. Typical refactoring steps of this strategy are (i) the creation of a *getter* method for accessing the delegate-class object from the main-class object and (ii) the replacement of all the calls to delegating methods in the main-class with direct calls to methods in the delegate-class. This technique leads to removing 3 CD smells in total.
Move Method. Most of the smell instances have been removed by applying this refactoring strategy. It consists of moving methods from the original class to a target one, and we remove 20 CD smell instances in total. In all cases, we move the original method into the class having the highest call frequency. We then refactor all the calls to the original method that shall instead call the new method.

It is worth noting that when refactoring CD smells we implemented 30 design changes in total (i.e., 7 encapsulation, 3 remove middle man, 20 move method). However, according to Table 2, the effect of these changes is the solution of 34 smell instances. As anticipated in Section 3.2, we found that a refactoring action does not necessarily map one-to-one with a smell removal. We aim at investigating this latter point as part of our future research.

6 EXPERIMENTS

6.1 Design of the experiments

To answer the research questions, we carried out a number of experiments on the two selected case studies introduced in Sec. 4.1. For each system we consider three different versions:

- original (or main) version (`OpenMRS`, `TeaStore`);
- after GC refactoring (`OpenMRS_NoGC`, `TeaStore_NoGC`);
- after CD refactoring (`OpenMRS_NoCD`, `TeaStore_NoCD`).

Table 3: Mapping between RQs and measurement data

RQ	subject	systems	data	statistical tests
RQ1	impact on execution time	`OpenMRS` `TeaStore` `OpenMRS_NoGC` `TeaStore_NoGC` `OpenMRS_NoCD` `TeaStore_NoCD`	CPU hotspots	Mann-Whitney U test Vargha-Delaney $\hat{A}_{AB}$
RQ2	impact on memory consumption	`OpenMRS` `TeaStore` `OpenMRS_NoGC` `TeaStore_NoGC` `OpenMRS_NoCD` `TeaStore_NoCD`	live/dead objects	Mann-Whitney U test Vargha-Delaney $\hat{A}_{AB}$

Table 3 maps each research question to the analyzed systems' versions, the measurement data carried out during the experiments, and the conducted statistical tests [1].

We repeat e2e tests 20 times for each system version. Each e2e test has been created using Postman according to the documentation of the system, and it repeatedly generates synthetic users that simulate realistic interactions through the exposed RESTful APIs. Each test issues a large sample of requests ($\sim$ 5800 for OpenMRS, and $\sim$ 7400 for TeaStore) to reduce the risk of obtaining biased results. During each e2e test we measure the CPU hotspots (milliseconds) and the live/dead objects (bytes) that represent the measurements to understand how AS impact on execution time and memory consumption, respectively.

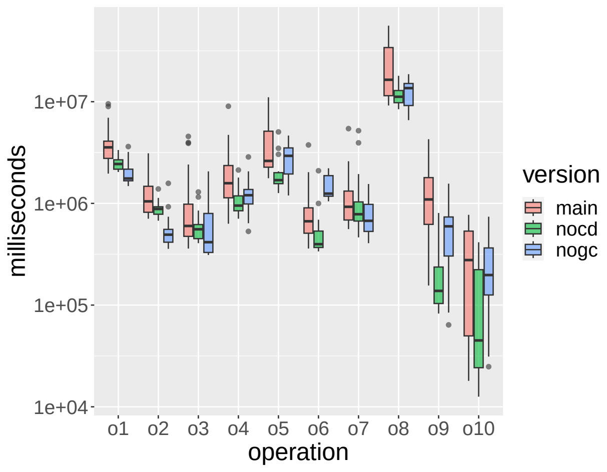
We additionally perform a pairwise comparison of the datasets (both cpu hotspots and live/dead objects) collected for all system versions. As reported in Table 3, we detect statistical significance using the Mann-Whitney U-test [47] (significance level $\alpha = 0.05$) and we calculate the non-parametric Vargha and Delaney's $\hat{A}_{AB}$ [16] to capture the effect size of the difference between A and B, typically characterized as follows: small ($\hat{A}_{AB} \geq 0.56$), medium ($\hat{A}_{AB} \geq 0.64$), and large ($\hat{A}_{AB} \geq 0.71$).

All the e2e tests have been executed on Dockerized system images deployed onto our testbed, i.e., a commodity hardware machine equipped with 2.3 GHz Dual-Core Intel Core i5 CPU, and 8 GB 2133 MHz LPDDR3 RAM.

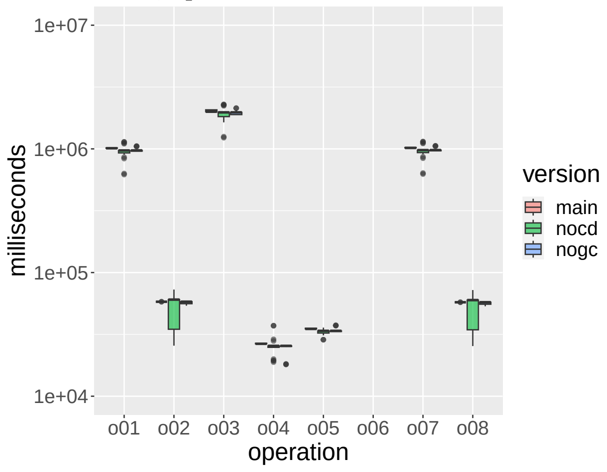
6.2 Results

6.2.1 RQ1 (impact on execution time). As anticipated above, for each case study and system version, we run 20 e2e tests and we measure the CPU hotspots at the method level. For each test, we collect the total execution time for each method, and then we rank all the methods in descending order to identify the most expensive methods executed by the services of the target system. Note that we select ten methods (o_1-o_{10}) for OpenMRS and the TeaStore persistence service, whereas the TeaStore image service includes eight methods in total, hence we rank those ones (o_1-o_8).

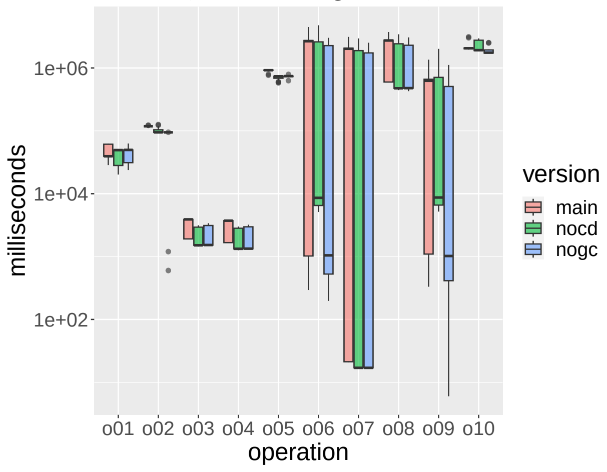
Figure 3 shows the box plots of the CPU hotspots for each analyzed system service: OpenMRS (monolithic service in Fig. 3a), TeaStore (image service in Fig. 3b), and TeaStore (persistence service in Fig. 3c). Each box plot illustrates, for all versions, the total execution time (expressed in milliseconds) of the most expensive methods over all e2e tests. We can observe that the order of magnitude varies between: 10^4 and 10^7 milliseconds for methods executed



(a) OpenMRS (monolithic service)



(b) TeaStore (image service)



(c) TeaStore (persistence service)

Figure 3: CPU hotspot data collected from the e2e tests

Table 4: CPU hotspot $\hat{A}_{AB}$ effect size

	o1	o2	o3	o4	o5	o6	o7	o8	o9	o10
A					*B* = OpenMRS (monolithic service)					
OpenMRS_NoCD	0.88	0.94	0.72	0.68	0.58	0.12	0.70	0.72	0.74	0.54
OpenMRS_NoGC	0.81	0.7	0.59	0.74	0.88	0.79	0.56	0.8	0.94	0.73
A					*B* = TeaStore (image service)					
TeaStore_NoCD	0.85	0.36	0.85	0.85	0.85	0.89	0.85	0.36	-	-
TeaStore_NoGC	0.86	0.38	0.88	1.00	0.86	0.87	0.86	0.38	-	-
A					*B* = TeaStore (persistence service)					
TeaStore_NoCD	0.55	0.88	0.89	0.89	1.00	0.59	0.82	0.82	0.46	0.71
TeaStore_NoGC	0.52	1.00	0.86	0.86	1.00	0.74	0.79	0.81	0.78	0.78

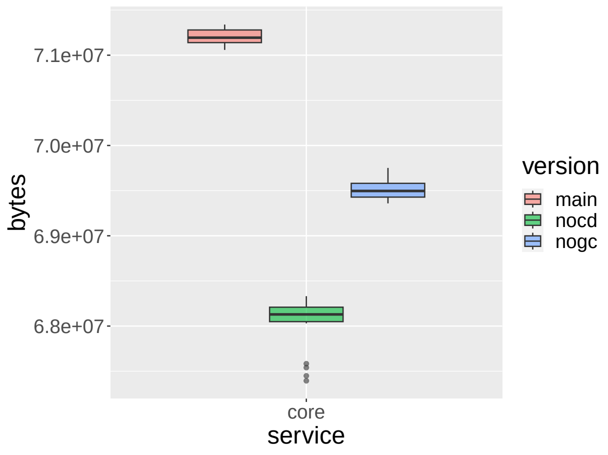
Table 5: CPU hotspot p-value

	o1	o2	o3	o4	o5	o6	o7	o8	o9	o10
A					*B* = OpenMRS (monolithic service)					
OpenMRS_NoCD	0.000	0.035	0.365	0.007	0.000	0.001	0.547	0.001	0.000	0.012
OpenMRS_NoGC	0.000	0.000	0.020	0.056	0.365	0.000	0.029	0.019	0.001	0.643
A					*B* = TeaStore (image service)					
TeaStore_NoCD	0.000	0.289	0.000	0.000	0.000	0.000	0.000	0.289	-	-
TeaStore_NoGC	0.000	0.210	0.000	0.000	0.000	0.000	0.000	0.210	-	-
A					*B* = TeaStore (persistence service)					
TeaStore_NoCD	0.576	0.000	0.000	0.000	0.000	0.305	0.000	0.000	0.653	0.009
TeaStore_NoGC	0.839	0.000	0.000	0.000	0.008	0.001	0.000	0.001	0.78	0.027

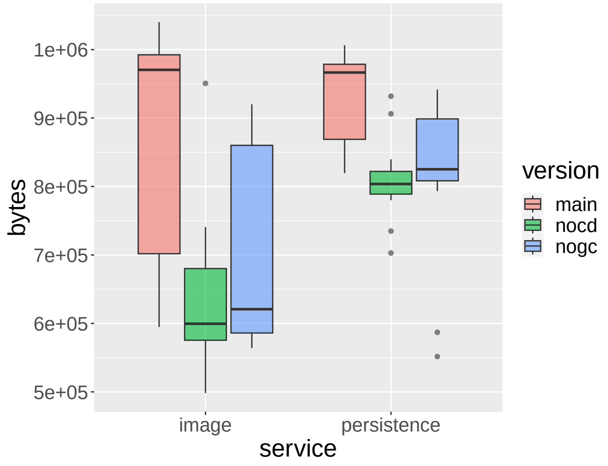
by the OpenMRS monolithic service and the TeaStore image service; and 10^2 and 10^6 milliseconds for methods executed by the TeaStore persistence service. The skewness also largely varies across the methods, and it is generally higher for those methods involving persistent data (e.g., o_9, o_{10} in Fig. 3a and o_6, o_7, o_9 in Fig. 3c). The median execution time of the original (main) version is generally higher compared to the corresponding NoCD and NoGC versions.

After refactoring GC and CD smell instances in OpenMRS, the execution time at the method level decreases up to 42% and 47%, respectively. According to the $\hat{A}_{AB}$ effect size in Table 4, we can observe that OpenMRS_NoCD is better than OpenMRS for 9 out of 10 methods. In particular, we have a large (green cells) and medium (white cells) effect size for 50% and 40% of the methods, respectively. The version OpenMRS_NoGC is always better than OpenMRS with either large (70% methods) or medium (30% methods) effect size. According to Table 5, the Mann-Whitney U-test detected statistically significant results in 80% of the methods for both OpenMRS_NoCD and OpenMRS_NoGC (i.e., p-value less than 0.05).

After refactoring GC and CD smell instances in TeaStore, we can observe that the execution time at the method level decreases up to 40% and 38%, respectively. Considering the image service, we can observe that both TeaStore_NoCD and TeaStore_NoGC are better than TeaStore for 6 out of 8 methods with large effect size. For the remaining 2 methods (o_2 and o_8 in both cases) the effect size is less than 0.5 but, according to Table 5, results are not statistically significant. Considering instead the persistence service, the TeaStore_NoCD is better for 90% of the methods. For 80% of them, we observe a large effect size and no statistically significant results for the remaining 2 methods. The TeaStore_NoGC is better for 90% of the methods with large effect size. The remaining method (o_1) has no statistical significance.



(a) OpenMRS (monolithic service)



(b) TeaStore (image and persistence services)

Figure 4: Memory usage datasets collected in the e2e tests

> **RQ1 summary.** After refactoring GC and CD smell instances, the execution time at the method level decreases up to 42% and 47%, respectively. The median execution time measured in the original version is generally higher compared to the corresponding NoCD and NoGC versions for both systems (OpenMRS and TeaStore). Considering the NoCD versions, we observe a large effect size for 50%-80% methods. Considering the NoGC versions, we also observe a large effect size for 70%-90% methods.

6.2.2 RQ2 (impact on memory consumption). For all e2e tests executed on each case study and specific versions, we measure the total memory consumption (bytes) for each service in terms of live and dead objects. Figure 4 shows two box plots of the memory consumption measured for the two systems: OpenMRS (monolithic service in Fig. 4a) and TeaStore (image and persistence services in Fig. 4b). The order of magnitude is 10^7 bytes for OpenMRS and it varies between 10^5 and 10^6 bytes for TeaStore.

About memory consumption measurements (see Figure 4), we can observe the following trend. The median memory consumption of the NoCD version is the lowest; the NoGC version has a higher median than NoCD; and the main version shows the highest value. The trend holds for both services of TeaStore as well as for OpenMRS. After refactoring GC and CD smell instances in OpenMRS, the memory consumption at the service level decreases up to 2% and 4%, respectively. After TeaStore refactoring, the memory consumption decreases up to 16% and 20%, respectively. This is a key indicator of AS showing an impact on software performance.

Table 6: Memory consumption $\hat{A}_{AB}$ effect size

A	B = OpenMRS (monolithic service)
OpenMRS_NoCD	1.00
OpenMRS_NoGC	1.00
A	B = TeaStore (image service)
TeaStore_NoCD	0.88
TeaStore_NoGC	0.84
A	B = TeaStore (persistence service)
TeaStore_NoCD	0.94
TeaStore_NoGC	0.85

Table 7: Memory consumption p-value

A	B = OpenMRS (monolithic service)
OpenMRS_NoCD	0.000
OpenMRS_NoGC	0.000
A	B = TeaStore (image service)
TeaStore_NoCD	0.000
TeaStore_NoGC	0.000
A	B = TeaStore (persistence service)
TeaStore_NoCD	0.000
TeaStore_NoGC	0.000

Table 6 indicates that both OpenMRS_NoCD and OpenMRS_NoGC have large effect size equal to 1.0. Considering the image service of TeaStore, we always have a large effect size (0.88 and 0.84 for TeaStore_NoCD and TeaStore_NoGC, respectively). The same holds for the persistence service (0.94 and 0.85 for TeaStore_NoCD and TeaStore_NoGC, respectively). Moreover, Table 7 demonstrates that results are always statistically significant.

> **RQ2 summary.** After refactoring GC and CD smell instances, we observe a memory consumption reduction up to 16% and 20%, respectively. The median memory consumption of the NoCD version is the lowest; the NoGC version has a higher median than NoCD; and the main version has the highest median. Considering OpenMRS, both NoCD and NoGC versions have a large effect size equal to 1.0. Considering TeaStore, we also observe a large effect size between 0.88 and 0.94 for NoCD versions, and between 0.84 and 0.85 for NoGC versions.

6.3 Lessons learned

From the conducted experimentation, we can derive the following main lessons learned. First, the refactoring of architectural smells does have a large impact on software performance, both execution time and memory consumption are largely affected in the considered systems. Second, there are no straightforward relationships between methods implemented within a certain operation, in fact each method shows peculiar variations in the considered performance metrics. Third, the refactoring of smells might reveal an opposite impact across methods, i.e., some methods benefit from the refactoring of one smell type, whereas other methods worsen their performance characteristics when refactoring the very same type of smell. Fourth, we found that memory consumption shows a more regular trend of variation w.r.t. execution time, i.e., all services under evaluation are beneficially affected more by CD instead of GC. This can be related to the selected services, we rather do not generalize any conclusion from this observation. Last, it is worth remarking that our experimentation considers a coarse granularity for the refactoring; all architectural smell instances are jointly removed, hence the system versions are embedding several changes, and it is rather impossible to trace back which instance of smell was more/less beneficial from a performance-based perspective.

6.4 Threats to validity

External validity. Threats in this category may exist if the characteristics of our case studies are not a generalization of other systems. We mitigated these threats by considering existing (well-known) benchmark systems from the literature of software performance engineering. We considered two systems following two different mainstream architectural styles: a monolithic service-oriented architecture (`OpenMRS`) and a microservices architecture (`TeaStore`). However, we acknowledge that the generalization of our findings to other systems requires additional experiments.

Internal validity. Threats may be caused by bias in establishing cause-effect relationships in our experiments. To limit these threats, we controlled the factors of interest in our case studies as much as possible. We analyzed the original codebase of both systems to spot AS and then we refactored the source code to remove the most severe true positive instances (at least 5% of detected instances). Fine-grained access to insights automatically extracted from the source through Arcan and Designate have been crucial to produce the new versions of both systems and then comparing the results to find cause-effect relations. Direct manipulation of the source code to control the presence of AS increases internal validity compared to observations without manipulation. Smell refactoring is a manual activity carried out by the authors based on their domain knowledge and reference guidelines proposed by Suryanarayana et al. [39]. Refactoring conducted by other engineers may lead to different quantitative results. We tried to mitigate this threat by eliciting the specific refactoring strategies (e.g., Extract Class and Encapsulation) adopted for each case study. We also reported the removed number of smells in each category and related to each strategy.

Conclusion validity. We addressed these threats by reducing the possibility of obtaining biased results. In particular, for each e2e test we collected a large sample of requests ($\sim 6k$ for `OpenMRS`, and $\sim 7k$ for `TeaStore`). Furthermore, each e2e test has been repeated 20 times for all system versions. We followed well-known guidelines in the software engineering research community to assess the statistical significance of our experiments [1]. In particular, we conducted pairwise comparisons among the datasets collected for all system versions using the Mann-Whitney U-test to calculate the p-value. In addition to statistical differences, we used the Vargha and Delaney's $\hat{A}_{AB}$ non-parametric effect size measure.

Construct validity. Threats in this category may arise in case selected measurements do not reflect the properties of interest of our study subjects. We limited this threat by assessing their validity before using them in our experimental campaign. In particular, we measured the impact on execution time and memory consumption at the method level and the service level, respectively. We adopted standard metrics in the performance engineering research community: CPU hotspots (execution time) and amount of live/dead objects (memory consumption). We also adopted standard statistical tests to carry out a pairwise comparison between the datasets collected by testing all the system versions. According to the guideline presented in [47], we used the standard significance level $\alpha = 0.05$ for the Mann-Whitney U-test. We also followed the common categories (small, medium, large) and corresponding levels (0.56, 0.64, 0.71) for the Vargha and Delaney's $\hat{A}_{AB}$ non-parametric effect size measure as described in [16].

7 CONCLUSION AND FUTURE WORK

This paper presents an exploratory study on the impact of AS on software performance with the goal of understanding the extent to which common AS affect the execution time and memory consumption of software systems. To answer our research questions, i.e., RQ1 (impact on execution time) and RQ2 (impact on memory consumption), we define a study design pipeline composed of case study selection, smell detection, smell refactoring, and experiments. We consider two well-known open-source systems (OpenMRS and TeaStore) commonly adopted as benchmarks in the software performance engineering research community, and we focus on two common smells, i.e., God Class and Cyclic Dependency. Results are, for the most, statistically significant and they show substantial improvements through the removal of smells, with a large effect size. In fact, after refactoring the detected smell instances, the execution time at the method level decreases up to 42% and 47%, respectively, while the memory consumption at the service level decreases up to 16% and 20%, respectively.

Our future research agenda includes the following directions: (i) extend the investigation to other AS, possibly those whose detection and refactoring have been assessed; (ii) identify which refactorings are more promising than others in solving further instances of AS; (iii) consider other applications, additional programming languages, and possibly industrial domains.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We acknowledge the support of MUR PRIN project 2017TWRCNB SEDUCE, and the PNRR MUR project VITALITY (ECS00000041) Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering*. 1–10.
[2] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Matteo Camilli, Andrea Janes, Barbara Russo, André van Hoorn, Robert Heinrich, Martina Rapp, Jörg Henß, and Ram Kishan Chalawadi. 2022. Scalability testing automation using multivariate characterization and detection of software performance antipatterns. *J. Syst. Softw.* 193 (2022), 111446.

[3] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. 2019. Architectural smells detected by tools: a catalogue proposal. In *International Conference on Technical Debt*. 88–97.

[4] Alison Fernandez Blanco, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. 2022. Software visualizations to analyze memory consumption: A literature review. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–34.

[5] Matteo Camilli, Carmine Colarusso, Barbara Russo, and Eugenio Zimeo. 2023. Actor-Driven Decomposition of Microservices through Multi-Level Scalability Assessment. *ACM Trans. Softw. Eng. Methodol.* (feb 2023). https://doi.org/10.1145/3583563

[6] Matteo Camilli, Andrea Janes, and Barbara Russo. 2022. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software* 187 (2022), 111225. https://doi.org/10.1016/j.jss.2022.111225

[7] Matteo Camilli and Barbara Russo. 2022. Modeling Performance of Microservices Systems with Growth Theory. *Empirical Software Engineering* 27, 2 (11 Jan 2022), 39. https://doi.org/10.1007/s10664-021-10088-0

[8] Alexandra-Maria Chaniotaki and Tushar Sharma. 2021. Architecture Smells and Pareto Principle: A Preliminary Empirical Exploration. In *International Conference on Mining Software Repositories*. 190–194.

[9] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. 2021. Enhancing the analysis of software failures in cloud computing systems with deep learning. *Journal of Systems and Software* 181 (2021), 111043.

[10] Leonardo da Silva Sousa, Willian Nalepa Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities: A Study of 50 Software Projects. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 354–365. https://doi.org/10.1145/3387904.3389276

[11] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. 2019. Are architectural smells independent from code smells? An empirical study. *J. Syst. Softw.* 154 (2019), 139–156.

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

[13] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. 2016. Automatic detection of instability architectural smells. In *International Conference on Software Maintenance and Evolution*. 433–437.

[14] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying architectural bad smells. In *European Conference on Software Maintenance and Reengineering*. 255–258.

[15] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a catalogue of architectural bad smells. In *International conference on the quality of software architectures*. 146–162.

[16] R.J. Grissom and J.J. Kim. 2005. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates.

[17] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Working Conference on Source Code Analysis and Manipulation*. 1–23.

[18] Wilhelm Hasselbring and Andre van Hoorn. 2015. *Open-Source Software as Catalyzer for Technology Transfer: Kieker's Development and Lessons Learned*. Research Report. Department of Computer Science, Kiel University, Kiel, Germany. http://www.inf.uni-kiel.de/de/forschung/publikationen/technische-berichte/

[19] Sebastian Herold. 2020. An Initial Study on the Association Between Architectural Smells and Degradation. In *European Conference on Software Architecture*. 193–201.

[20] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 1–35.

[21] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *International Conference on Automated Software Engineering*. 497–508.

[22] Herb Krasner. 2021. The cost of poor software quality in the US: a 2020 report. *Proc. Consortium Inf. Softw. QualityTM* (2021).

[23] Henry H Liu. 2011. *Software performance and scalability: a quantitative approach*. John Wiley & Sons.

[24] Lei Liu, Zhiying Tu, Xiang He, Xiaofei Xu, and Zhongjie Wang. 2021. An Empirical Study on Underlying Correlations between Runtime Performance Deficiencies and "Bad Smells" of Microservice Systems. In *International Conference on Web Services*. 751–757.

[25] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. 2020. Diagnosing root causes of intermittent slow queries in cloud databases. *VLDB Endowment* 13, 8 (2020), 1176–1189.

[26] Robert Cecil Martin. 2003. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

[27] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. 2018. Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company. In *European Conference on Software Architecture*, Vol. 11048. 320–335.

[28] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2021. A systematic mapping study on architectural smells detection. *Journal of Systems and Software* 173 (2021), 110885.

[29] Riccardo Pinciroli, Connie U. Smith, and Catia Trubiani. 2021. QN-based Modeling and Analysis of Software Performance Antipatterns for Cyber-Physical Systems. In *International Conference on Performance Engineering*. 93–104.

[30] JProfiler Project. 2001. *JProfiler web site*. Retrieved Apr, 2023 from ttps://www.ej-technologies.com/products/jprofiler/overview.html

[31] Kieker Project. 2001. *Kieker web site*. Retrieved Apr, 2023 from http://kieker-monitoring.net/

[32] OpenMRS Project. 2001. *OpenMRS web site*. Retrieved Apr, 2023 from https://openmrs.org/

[33] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2022. Automated Identification of Performance Changes at Code Level. In *International Conference on Software Quality, Reliability and Security*. 916–925.

[34] Darius Sas, Paris Avgeriou, Ilaria Pigazzini, and Francesca Arcelli Fontana. 2022. On the relation between architectural smells and source code changes. *J. Softw. Evol. Process.* 34, 1 (2022).

[35] Darius Sas, Paris Avgeriou, and Umut Uyumaz. 2022. On the evolution and impact of architectural smells - an industrial case study. *Empirical Software Engineering* 27, 4 (2022), 86.

[36] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: A Software Design Quality Assessment Tool. In *International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*. 1–4.

[37] Tushar Sharma, Paramvir Singh, and Diomidis Spinellis. 2020. An empirical investigation on the relationship between design and architecture smells. *Empir. Softw. Eng.* 25, 5 (2020), 4020–4068.

[38] Connie U Smith. 2007. Introduction to software performance engineering: Origins and outstanding problems. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 395–428.

[39] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2015. Refactoring for Software Design Smells. https://www.sciencedirect.com/science/article/pii/B9780128013977120016.

[40] Antony Tang, Yan Jin, and Jun Han. 2007. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software* 80, 6 (2007), 918–934.

[41] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. 2019. How developers discuss architecture smells? an exploratory study on stack overflow. In *International conference on software architecture*. 91–100.

[42] Catia Trubiani, Aldeida Aleti, Sarah Goodwin, Pooyan Jamshidi, André van Hoorn, and Samuel Gratzl. 2020. VisArch: Visualisation of Performance-based Architectural Refactorings. In *European Conference on Software Architecture*. 182–190.

[43] Catia Trubiani, Pooyan Jamshidi, Jürgen Cito, Weiyi Shang, Zhen Ming Jiang, and Markus Borg. 2019. Performance Issues? Hey DevOps, Mind the Uncertainty. *IEEE Softw.* 36, 2 (2019), 110–117.

[44] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *International Conference on Software Engineering*. 1072–1084.

[45] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*.

[46] Hanzhang Wang, Ali Ouni, Marouane Kessentini, Bruce Maxim, and William I Grosky. 2016. Identification of web service refactoring opportunities as a multi-objective problem. In *International Conference on Web Services*. 586–593.

[47] Rand R Wilcox. 2001. *Fundamentals of modern statistical methods: Substantially improving power and accuracy*. Vol. 249. Springer.

[48] Chenxing Zhong, Huang Huang, He Zhang, and Shanshan Li. 2022. Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. *Software: Practice and Experience* 52, 12 (2022), 2574–2597.