The Impact of Auto-Refactoring Code Smells on the Resource Utilization of Cloud Software

Asif Imran* and Tevfik Kosar†
Department of Computer Science and Engineering, University at Buffalo
Amherst, New York 14260-1660, USA
Email: *asifimra@buffalo.edu, †tkosar@buffalo.edu

***Abstract*—Cloud-based software-as-a-service (SaaS) have gained popularity due to their low cost and elasticity. However, like other software, SaaS applications suffer from code smells, which can drastically affect functionality and resource usage. Code smell is any design in the source code that indicates a deeper problem. The software community deploys automated refactoring to eliminate smells which can improve performance and also decrease the usage of critical resources. However, studies that analyze the impact of automatic refactoring smells in SaaS on resources such as CPU and memory have been conducted to a limited extent. Here, we aim to fill that gap and study the impact on resource usage of SaaS applications due to automatic refactoring of seven classic code smells: god class, feature envy, type checking, cyclic dependency, shotgun surgery, god method, and spaghetti code. We specified six real-life SaaS applications from Github called Zimbra, OneDataShare, GraphHopper, Hadoop, JENA, and JAMES which ran on Openstack cloud. Results show that refactoring smells by tools like JDeodrant and JSparrow have widely varying impacts on the CPU and memory consumption of the tested applications based on the type of smell refactored. We present the resource utilization impact of each smell and also discuss the potential reasons leading to that effect.**

***Index Terms*—Code smells, automated refactoring, cloud resource utilization**

I. Introduction

Software as a Service (SaaS) running in cloud platforms has become increasingly popular, mainly due to their lower cost, high availability, and quality of service for the users. SaaS applications are becoming mainstream in the everyday lives of users who use them to conduct day-to-day business and personal software needs [1]. SaaS applications are rapidly capturing the market, and more service providers are migrating their software to cloud everyday [2]. The critical advantage of SaaS is its capability to serve millions of users all around the world, harnessing the elasticity, reliability, and scalability of the underlying cloud platform.

Developing SaaS applications differ from desktop applications since those are required to perform on the cloud backbone where multiple software are competing for the compute cloud resources, mainly CPU and memory [3]. Furthermore, the shorter time-to-market affects the development of those open-source software, which are designed to serve multiple users. Those applications are designed by many contributors who work together to solve common issues and regularly add new features. When open source SaaS is incorrectly coded, they can drain cloud resources like CPU and memory, thereby resulting in wastage of critical resources that are billed by cloud service providers [3]. This results in technical debt and increases the cost of hosting the software in the cloud. We call the erroneous designs as code smells of SaaS.

Resource consuming code smells can degrade the performance of the SaaS application and increase the cost. Users will be demotivated to pay the increased cost and can move to less expensive services provided by the competitors. As a result, fixing those smells can improve the cost-efficiency of critical computing resources without sacrificing the quality of service. Hence, selectively refactoring these code smells would benefit the SaaS service providers as well as the customers of those services. Previous research has focused on exploring the impact of code smells on energy consumption, such as battery usage in smartphones [4]. Also, the importance of detecting and eliminating smells that affect speedup during migration has been addressed [5]. However, there is a lack of empirical study which focuses on analyzing the impact of automatically refactoring smells in SaaS on CPU and memory consumption.

This paper aims to fill the void by analyzing the effect of automatic refactoring of smells on resource utilization of SaaS applications running in the cloud. We conducted an initial search that included 84 repositories and filtered those to match the required pre-specific criteria of this research. We selected seven code smells, which we detected in the selected software using two popular tools, JDeodrant [6] and JSparrow [7], which have capabilities of detecting classic code smells and applying automatic refactoring on those. Out of the seven smells, JDeodrant detected and refactored *god class, feature envy,* and *type checking* whereas JSparrow could detect and refactor four smells namely *cyclic dependency, shotgun surgery, god method,* and *spaghetti code*. We strongly believe our research efforts will help to identify the critical importance of refactoring specific code smells in cloud-based software and their impact on the utilization of precious cloud resources.

The rest of the paper is organized as follows: Section II describes the methodology of our study, Section III presents the results of the experiments and summary of our findings, Section IV discusses the related work in this area, and Section V concludes the paper.

II. Experimental Process

This section describes the goals of this study and the research questions answered, the selection of code smells, and

DOI reference number: 10.18293/SEKE2020-138

Smell	Primary Causes	Impact on Software	Refactoring technique
cyclic dependency	-Violation of acyclic modularization [8] -Misplaced elements -Two packages are dependent on each other -Lack of encapsulation	-Difficult to maintain -Single method called for multiple tasks -Has ripple effect on other abstractions [8]	-Encapsulate all packages in a cycle and assign to single team
god method	-Many activities in a single method [9] -Tangled code -Long methods with multiple responsibilities	-Memory leak -Multiple calls to same function [10] -Difficult co-ordination of packages	-Extract method -Replace method with method object
type checking	-Using complex variation of an algorithm requiring execution based on the value of an attribute [11] -Objects in class come from different workers	-Abuse of type casting -Redundant code in a method or class -Less flexible code	-Replace "instanceof" from code -Strategy pattern
spaghetti code	-Convoluted code -Continuous addition of new code and no removal of obsolete ones [12] -Procedural code design	-Difficult to understand code [12] -Lack of well-articulated code	-Replace procedural code segments with object oriented design
feature envy	-Accessing data of another object often [6] -Occurs when fields are moved to data class -Data defined in class A, however operations defined in class B	-High volume of requests to access a class and its objects -Numerous read and write to remote object	-Move method -Extract method
shotgun surgery	-Single behavior defined across multiple classes [9]	-Requires multiple changes in different locations (e.g., multiple files) of the code in order to make a single modification [9] -Existing behavior in multiple classes	Inline class
god class	-Class aims to do many activities [9] -Large number of instance variables declared in one class	-Difficult to manage multiple functionalities -Difficult to understand code	-Extract class -Extract Interface

TABLE I: Characteristics and applied refactoring properties of software smells

the methodology of experimental and evaluation processes.

A. Identified Research Questions

The main goal is to determine whether automatic refactoring of smells in cloud-based software impact the resource consumption in terms of CPU and memory. The motivation is derived from the fact that cloud resources are in high demand, and any unnecessary resource usage will incur unnecessary costs. More specifically, we want to determine the refactoring of which smells will result in an increase or reduction of CPU and Memory consumption in the cloud. Based on this goal, we determine the following research questions:

- **RQ1.** How does auto-refactoring of code smells in cloud-based software affect CPU utilization?
- **RQ2.** How does auto-refactoring of code smells in cloud-based software affect memory utilization?

B. Selected Code Smells

A code smell is a software behavior that is indicative of more profound quality issues [9]. We selected the aforementioned seven smells mainly because of the following reasons. Primarily, those smells are studied popularly by software community, and they are considered as classic smells [9]. Secondly, several tools can detect the code smells; however, few tools are available, which can automatically refactor those. Our selected smells could be automatically detected by the refactoring tools we used in this study. Table I provides a summary of the smells which includes the causes, impact on software, and the refactoring policy applied by the tools.

C. Study Methodology

To eliminate biases in our study, we executed each of the apps 24 times and took the mean of the data. To analyze the effect of a given smell, first, we ran the smelly code 3 times. Then we refactored a smell and ran the software 3 times more, hence for one software, we had (7*3=21)+3=24 runs. Running each software 24 times resulted in a total of 144 runs. This required a significant time frame.

We conducted the experiments in $OpenStack$ cloud servers, which had 32 GB RAM, 8 core processors, and 2 TB persistent storage [13]. We set up the cloud and ran the six software in VM instances, which were created using *kernel virtual machine (kvm)*. Every time we refactored, we executed the software in a new instance and deleted all existing data to minimize the impact of caching. We did not run any other cloud VM instance with other services to minimize the effect of external entities. Following this mechanism allowed us to erase all existing data before the software was run. As a result, it helped us achieve the same initial condition for each experimental run. Next, we followed a methodology for refactoring the selected software, which is shown in Figure 1 and discussed here.

- **Smell detection and provenance:** As a first step, we applied *JDeodrant* to detect the three code smells (*god*

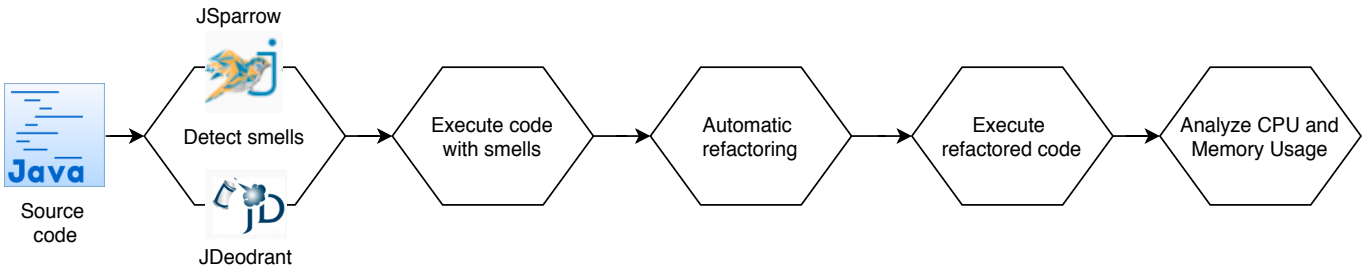

Fig. 1: Automatic detection and refactoring of code smells.

class, feature envy, and *type checking*), followed by *JSparrow* to detect four remaining smells. Since both *JDeodrant* and *JSparrow* have plugins for *Eclipse IDE*, we imported the source codes in to *Eclipse* and built those. After detection of the smells, we preserved provenance of the packages, classes, and methods, which were affected by the smells. Overall, *JDeodrant* and *JSparrow* detected 744 instances of the seven analyzed smells in the six software. The reason for using two tools is to include significant number of smells in the study. None of the tools could single-handedly detect and refactor all the smells considered here.

- **Executing smelly code:** Next, we executed the codes in the cloud. We ran the code three times, each time with the same workload, and collected the CPU and memory utilization during each clock cycle using scripts in $OpenStack$ [3]. This allowed us to eliminate biases.
- **Automatic refactoring of smells:** Afterwards, we refactored the code from the smells. For each smell, we took a new copy of the smelly code and refactored to eliminate the effect of another refactoring. As a result, the projects were imported newly for each smell, refactored and executed. After refactoring, we repeat step 2 and record the CPU and memory consumption for the same workload and following the same method.
- **Logging the data:** We proceeded to log all the CPU and memory consumption in each experimental run. We tried to eliminate any external impact during each execution. We recorded the CPU and memory consumption during each CPU cycle and stored the data persistently.
- **Statistical operation:** Finally, we obtained the arithmetic mean of the CPU and memory usage. Afterward, we tabulated the results and compared whether the resource usage improved or deteriorated after refactoring the smells.

The next section highlights results obtained by following these prescribed steps.

III. Results

The obtained results during experimentation have been provided here. The number of detected smells is showcased which is followed by quantitative analysis of the data achieved during experimentation.

System	Activity	LoC	Commits	NoC
Zimbra	08/05-03/19	24,698	15,052	1,871
JGraph	09/09-02/19	22,758	1,360	187
OnedataShare	11/18-01/20	23,002	1,644	390
Hadoop	12/11-03/20	14,2790	23,611	2,069
JENA	06/10-05/19	70,948	8,287	1,392
JAMES	06/04-08/19	27,003	9,314	340

TABLE II: List of selected systems for the study

A. Selected Software for Analysis

For this study, we identified and selected six open-source software which serve the users in the cloud. Among those, we analyzed 3,11,199 Lines of Code (LoC) and 6,249 class files. *Zimbra* [14] is an OS-independent emailing, and file sharing tool running in the cloud, popularly used by Dell, Rackspace, and Mozilla. *OneDataShare* [14] is a cloud-based data transfer tool that incorporates multiple transfer protocols, including DropBox and GoogleDrive. *GraphHopper* is a GIS application provided as SaaS that incorporates spatial rules in hybrid mode. *Hadoop* is a software framework which was first proposed by Google to facilitate the analysis of large volume of data in the cloud [15]. *Java Apache Mail Enterprise Server (JAMES)* consists of a modular architecture based on state of the art components which provides secure, stable, and end-to-end mail servers running on the JVM [14]. *JENA* is a platform provided by Apache for designing and building linked data applications, giving access to a variety of APIs for serialization, processing, storage, and transfer [16]. Table II shows details related to the software selected for this study. Following is a description of the mechanism followed here to to obtain those software.

To analyze the impact of automatic refactoring of smells on resource usage, we decided to use open-source software that runs in the cloud as Software as a Service (SaaS) for multiple reasons. First, there are many software which are developed as SaaS for cloud since cloud computing has increased in popularity. So the availability of software was satisfied. Second, due to the high demand for the cloud, those software are heavily deployed by the industry. As shown in the previous paragraph, our selected software are used widely by top companies. Hence we believe it is important to analyze the impact of

automatic smell refactoring on resource consumption since those companies will benefit from our research. Third, the authors of this paper felt that the software running in the cloud will contain a significant number of smells since those have complex code blocks that are written to ensure real-time user interactions and serving a large number of users from remote cloud data centers where they are hosted. Finally, one of the main challenges of SaaS is to optimize resource utilization, since provisioning extra resources in the cloud will require additional cost. Hence optimized use of cloud resources will result in cost reduction for the cloud service provider.

We obtained the source code of the software from Github during November 2019. The software source code obtained based on search in Github using the following keywords: *cloud computing software, Software as a Service, pervasive computing,* and *cloud data transfer*. From there, we selected the software sorted by popularity. To ensure that our selection of software is unbiased, we implement a set of checks which software shall satisfy to be selected for analysis. Those checks are described as follows:-

- ***Check 1 - Initial list of software:*** Initially, we conducted a preliminary search in Github to identify the realistic chance of obtaining software that runs in the cloud. This search was manual and it involved looking out for cloud computing SaaS. It helped us to identify the keywords for searching in Github and also laid the foundation for the following checks.
- ***Check 2 - Script to automatically download the software:*** We developed a script that will parse through Github projects and automatically download the source codes from the master branch based on the keywords. It resulted in downloading 84 repositories from Github. We required to use automated scripts because cloning the 84 repositories manually would be time-consuming.
- ***Check 3 - Refining the search to match tool requirements:*** Finally, we refined our search to be in line with the requirements of our selected automatic code smell detection and refactoring tools. Prior to that, we eliminated any source code not written in Java, which left us with a list of 21 repositories. Next, we decided to include source codes that can be compiled in Eclipse as *JDeodrant* and *JSparrow* both have plugins, which can be run using Eclipse. After this, we were left with 11 repositories. Finally, we considered repositories with at least 10,000 lines of code, since otherwise we could run into the risk of considering a prototype software which we aimed to avoid.

B. Analysis of Results

We start with highlighting our findings related to CPU utilization. More specifically, we identify which code smells from the list will impact the CPU consumption of the analyzed Java software. Hence we address the research question *"How does auto-refactoring of code smells in cloud-based software affect CPU utilization?"* Table III identifies the change in CPU utilization after each smell is refactored. Each column highlights the percent change in CPU utilization after refactoring the smell specified in column heading. The negative sign in front of the numbers show decrease in resource use, whereas the positive sign show increase in resource usage.

After presenting the analysis of obtained results for CPU consumption, we analyze the results for change in memory usage after refactoring the smells. Hence, we aim to answer our second research question, *"How does auto-refactoring of code smells in cloud-based software affect memory utilization?"* Table IV identifies the percent change in memory utilization after refactoring each of the smells. The values of memory usage were measured in MB. Following is an analysis of the obtained results.

As seen in Table III, the refactoring of *cyclic dependency* smells improves CPU utilization for all six software tested. The CPU utilization was reduced by 16.52%, 17.22%, 50.81%, 4.67%, 31.96%, and 24.53% respectively for *Zimbra, GraphHopper, OneDataShare, Hadoop, JENA, and JAMES*. Cyclic dependencies result in direct and indirect dependencies between abstractions [17]. The abstractions were tightly coupled between a large number of direct and indirect cyclic dependencies, so they resulted in a tangled design of the code. Further analysis showed that since the elements were not placed in the correct package, it resulted in cyclic dependencies between packages as well. Co-ordination between the packages became difficult which resulted in multiple calls to the same package. Hence, the existence of this smell resulted in higher CPU and memory usage. To eliminate the smell, JSparrow encapsulated all the packages involved in the chain and assigned it to a single team, which required less processing and could be loaded once into memory, thus reducing resource consumption in terms of both CPU and memory.

A common practice to eliminate the *god method* smells by refactoring tools is to detect and remove those by implementing *extract method* design [18]. This refactoring procedure will improve the quality and maintainability of the software while preserving correctness. However, our results in Tables III and IV show that refactoring *god method* leads to an increase in resource utilization. The extra resource usage we found came from the higher number of message traffic obtained from architecture modification of this smell. Refactoring of this smell enables us to obtain a modular architecture of code where the elements have higher cohesion and lower coupling. Despite the benefits, this refactoring mechanism may not be ideal for cloud software as it might create harmful side effects in terms of sustainability [18]. Refactoring of *god method* smells yielded an increase in CPU utilization of *3.50%, 14.59%, 35.79%, 10.45%, 14.36%, and 3.69%* for *Zimbra, GraphHopper, OneDataShare, Hadoop, JENA, and JAMES*.

Removal of *shotgun surgery* smells contributed to the reduction of resource usage by the software. As we know *shotgun surgery* smells occur when we try to modify a class which in turn makes multiple modifications to several different classes. For example, in *OneDataShare*, removal of all instances of this smell resulted in 52.30% of CPU and

TABLE III: The impact of smell refactoring on percentage of CPU utilization

Table	Affect on CPU utilization after refactoring						
	cyclic dependency	*god method*	*shotgun surgery*	*spaghetti code*	*feature envy*	*type checking*	*god class*
Zimbra	-16.5%	+3.5%	-42.6%	-27.6%	+11.0%	-33.7%	+14.8%
GraphHopper	-17.2%	+14.6%	-30.7%	-0.7%	+27.6%	-10.4%	+66.2%
OneDataShare	-50.8%	+35.8%	-52.3%	-27.4%	+66.8%	-53.6%	+48.8%
Hadoop	-4.7%	+10.5%	-14.4%	-8.2%	+20.1%	-6.8%	+17.4%
JENA	-32.0%	+14.4%	-35.5%	-10.4%	+57.0%	-2.8%	+78.5%
JAMES	-24.5%	+3.7%	-20.7%	-2.7%	+35.0%	-3.3%	+20.6%

TABLE IV: The impact of smell refactoring on percentage of Memory utilization

Table	Affect on Memory utilization after refactoring						
	cyclic dependency	*god method*	*shotgun surgery*	*spaghetti code*	*feature envy*	*type checking*	*god class*
Zimbra	-60.8%	+14.2%	-71.2%	-60.5%	+12.6%	-53.9%	+6.2%
GraphHopper	-55.6%	+18.9%	-57.1%	-4.7%	+66.7%	-22.1%	+54.8%
OneDataShare	-33.2%	+38.2%	-31.4%	-4.8%8	+81.9%	-13.6%	+88.1%
Hadoop	-7.5%	+7.4%	-12.8%	-20.6%	+5.4%	-7.7%	+17.3%
JENA	-7.5%	+48.6%	-1.7%	-14.2%	+28.7%	-2.9%	+42.3%
JAMES	-13.2%	+0.1%	-21.9%	-17.8%	+14.7%	-5.0%	+35.3%

31.43% of memory utilization. We consulted the $diff$ in the commits of $OneDataShare$ and agreed that the smell was introduced due to the practice of overzealous and sudden changing of multiple components in different classes during development to incorporate new requirements without proper documentation. The refactoring tool used *Inline Class* to transfer the diversified behaviors of one operation to one class. This was done for 39 occurrences of this smell. As a result, we obtained the improvement of resource utilization.

OneDataShare was found to have improved CPU utilization by 27.35% when the *spaghetti code* smells were refactored. On the other hand, *Zimbra's* memory utilization improved by 60.47% when this smell was removed from it. Once again taking $OneDataShare$ as a case study, upon analysis of its source code using the refactoring tools, it was seen that the code used a large volume of $GOTO$ statements. Excessive use of $GOTO$ instead of well-articulated code design resulted in software that is convoluted and unmanageable [19]. At the same time, it causes the program to have methods scattered across many classes, which required more memory for file read and write operations.

Eliminating *feature envy* causes increased resource utilization which is a threat to the sustainability of the software, as it would result in provisioning more resources for the same task, thus incurring more cost [20]. More specifically, CPU and memory consumption increased by *66.84%* and *81.87%* respectively when this smells was refactored in *OneDataShare*. Analysis of files in $OneDataShare$ showed that the method calls required access to specific classes each time it requested for certain operations, hence increasing inter-class communication, thus deteriorating memory and CPU usage.

Refactoring the *type checking* smell yielded CPU utilization improvement of 53.6% of *OneDataShare*. Also, notable improvement of memory utilization of 53.9% was observed for *Zimbra*. This is because type checking results in a large function which should be broken down into multiple smaller functions. Calling the large function numerous times will cause all its functionality to be executed even when it is not necessary, yielding high CPU and memory utilization.

The tested software started to consume an alarming quantity of memory after refactoring *god class* smells. In general, the reason for a similar pattern of memory usage can be summarized to two main causes. The first is despite having a large volume of LoC and effective session cache management, the refactoring divided large classes into multiple sub-classes [18]. Hence, implementing *extract method* on *god classes* caused an increased number of calls the program has to make to perform its tasks, which increased the data volume stored in memory. Secondly, all the software have multiple developers contributing to it. So, from the perspective of community smells, there is a possible reason between these smells and developer viewpoint, which requires further research.

In summary, it is seen that all the software considered here suffered from increased CPU utilization after smells called *god method, feature envy,* and *god class* were refactored. The increase is *3.5%, 11.00%, and 14.8%* respectively for the three smells. Although the increase in CPU utilization may seem low, however, it is important to remember that when the software runs in the cloud, every CPU cycle is charged to the customer, hence increase in CPU utilization due to refactoring of software smells will lead to the cloud client incurring unnecessary extra cost. On the other hand, it is seen that refactoring the smells called *cyclic dependency, shotgun surgery, spaghetti code,* and *type checking* contributes to reduced CPU usage.

As seen in Table IV, memory consumption is seen to drastically increase after refactoring *god class*. For *Zimbra, GraphHopper, and OneDataShare* this increase in memory consumption is found to be *6.2%, 54.8%, and 88.1%*. Overall, similar observations are made for *feature envy, and god method*

smells as well for all six software. Like CPU, memory resource is also paid in the cloud, hence refactoring these smells will lead to additional cost as extra memory needs to be provisioned. We determine that the refactoring techniques adopted by *JDeodrant* for *god class,* and *feature envy* smells, together with *JSparrow* for *god method* are not useful for the cloud-based applications since all the six software resulted in an increase of CPU utilization and memory utilization.

IV. Related Work

Platform-specific code smells in High-Performance Computing applications were determined by Wang et al. [5]. AST based matching was used to determine smells present in an HPC software by comparing it with a dictionary of smells. The authors claimed that the removal of such smells would increase the speedup of the software when migrated to a new platform. The assumption was that certain code blocks perform well in terms of speedup in a given platform. However, the results show that certain smell detection and refactoring reduced the speedup, thus challenging the claims and showing the importance of further research in this area.

Oliveira et al. conducted an empirical study to evaluate nine context-aware Android apps to analyze the impact of automated refactoring of code smells on resource consumption [21]. They studied three code smells, namely *god class, god method, and feature envy*. They found that for the three smells, resource utilization increases when they are refactored. Although their findings are useful, it is limited to the analysis of three code smells only. At the same time, the importance of analyzing the impact of automated code smell refactoring on cloud computing SaaS applications were not considered.

V. Conclusion

In this paper, we evaluated the impact of automatically refactoring seven code smells on resource usage of software running in the cloud platform. Obtained results highlight that the refactoring techniques adopted by *JDeodrant* for *god class,* and *feature envy*, together with *JSparrow* for *god method* resulted in more message traffic which adversely affected CPU and memory usage. More specifically, cumulative increase of CPU and memory consumption for refactoring these three smells significantly higher as shown in Table III and Table IV. Hence, there exists scope of further research to improve the automatic refactoring mechanisms of existing tools. Also, determining the correlation between refactoring multiple smells and resource consumption needs to be explored. Additionally, impact on resource usage after refactoring smells specific to the cloud should be studied.

Acknowledgment

This project is in part sponsored by the National Science Foundation (NSF) under award numbers OAC-1724898 and OAC-1842054.

References

[1] V. V. H. Pham, X. Liu, X. Zheng, M. Fu, S. V. Deshpande, W. Xia, R. Zhou, and M. Abdelrazek, "Paas-black or white: an investigation into software development model for building retail industry saas," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 285–287.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] A. Imran, A. U. Gias, R. Rahman, A. Seal, T. Rahman, F. Ishraque, and K. Sakib, "Cloud-niagara: A high availability and low overhead fault tolerance middleware for the cloud," in *16th Int'l Conf. Computer and Information Technology*. IEEE, 2014, pp. 271–276.

[4] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical evaluation of the energy impact of refactoring code smells." in *ICT4S*, 2018, pp. 365–383.

[5] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi, "A platform-specific code smell alert system for high performance computing applications," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 652–661.

[6] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1037–1039.

[7] S. IT-Consulting. (2020) Jsparrow.

[8] S. Sarkar, G. M. Rama, N. N. Siddaramappa, A. C. Kak, and S. Ramachandran, "Measuring quality of software modularization," Mar. 27 2012, US Patent 8,146,058.

[9] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[10] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[11] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 329–331.

[12] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 181–190.

[13] A. Imran, S. Aljawarneh, and K. Sakib, "Web data amalgamation for security engineering: Digital forensic investigation of open source cloud." *J. UCS*, vol. 22, no. 4, pp. 494–520, 2016.

[14] A. Imran, "Design smell detection and analysis for open source java software," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 644–648.

[15] M. R. Ghazi and D. Gangodkar, "Hadoop, mapreduce and hdfs: a developers perspective," *Procedia Computer Science*, vol. 48, no. C, pp. 45–50, 2015.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

[17] G. Samarthyam, G. Suryanarayana, and T. Sharma, "Refactoring for software architecture smells," in *Proceedings of the 1st International Workshop on Software Refactoring*, 2016, pp. 1–4.

[18] R. Pérez-Castillo and M. Piattini, "Analyzing the harmful effect of god class refactoring on power consumption," *IEEE software*, vol. 31, no. 3, pp. 48–54, 2014.

[19] M. Ceccato, P. Tonella, and C. Matteotti, "Goto elimination strategies in the migration of legacy code to java," in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 53–62.

[20] K. Nongpong, "Feature envy factor: A metric for automatic feature envy detection," in *2015 7th International Conference on Knowledge and Smart Technology (KST)*. IEEE, 2015, pp. 7–12.

[21] J. Oliveira, M. Viggiato, M. F. Santos, E. Figueiredo, and H. Marques-Neto, "An empirical study on the impact of android code smells on resource usage." in *SEKE*, 2018, pp. 314–313.