Towards Sustainable Cloud Software Systems through Energy-Aware Code Smell Refactoring

Asif Imran[1], Tevfik Kosar[2], Jaroslaw Zola[2], M. Fatih Bulut[3]
[1]Department of Computer Science, California State University, San Marcos, California, USA
[2]Department of Computer Science & Engineering, University at Buffalo, Buffalo, New York, USA
[3]IBM TJ Watson Research Center, Yorktown Heights, New York, USA
Email: aimran@csusm.edu, {tkosar,jzola}@buffalo.edu, mfbulut@us.ibm.com

***Abstract*—Software applications and workloads, especially within the domains of Cloud computing and large-scale AI model training, exert considerable demand on computing resources, thus contributing significantly to the overall energy footprint of the IT industry. In this paper, we present an in-depth analysis of certain software coding practices that can play a substantial role in increasing the application's overall energy consumption, primarily stemming from the suboptimal utilization of computing resources. Our study encompasses a thorough investigation of 16 distinct code smells and other coding malpractices across 31 real-world open-source applications written in Java and Python. Through our research, we provide compelling evidence that various common refactoring techniques, typically employed to rectify specific code smells, can unintentionally escalate the application's energy consumption. We illustrate that a discerning and strategic approach to code smell refactoring can yield substantial energy savings. For selective refactorings, this yields a reduction of up to 13.1% of energy consumption and 5.1% of carbon emissions per workload on average. These findings underscore the potential of selective and intelligent refactoring to substantially increase energy efficiency of Cloud software systems.**

***Index Terms*—Cloud software, code smells, software batch refactoring, energy consumption, carbon footprint.**

I. Introduction

Software applications are considerable consumers of computing resources, leading to a significant energy consumption and carbon footprint. Recent estimates suggest that Information Technology accounts for approximately 11% of global energy consumption [1]. Projections indicate that by 2030, data centers alone could account for 3–13% of global electricity use, a stark increase from the 1% recorded in 2010, propelled by escalating demands and emerging trends like large-scale AI workloads [2]. The software industry is responsible for about three percent of global carbon emissions, which is very close to that associated with the aviation industry [3]. Certain software coding practices, if left unattended, can trigger a substantial surge in energy consumption, primarily stemming from the suboptimal utilization of computing resources. These coding practices are referred to as "code smells" (or sometimes as "energy smells") which are defined as *characteristics in software source code indicating a deeper, underlying issue* [4].

Software engineers constantly strive to restructure their code to eliminate the code smells, a mechanism popularly known as refactoring [5]. While current code refactoring practices prioritize the elimination of smells that impact software correctness [6], maintainability [7], and scalability [8], there has been a significant oversight regarding the analysis of the impact of such refactorings on energy consumption and carbon emission, a critical issue related to code smells. Research has demonstrated that certain code smells contribute to excessive resource consumption in software [9]. However, the potential effects of code smell refactoring practices on application energy consumption and carbon emission have only been sparsely studied. This knowledge gap and the disparity between theory and practice serve as the driving motivation for this study.

Certain categories of code smells can substantially increase energy consumption and carbon emission, particularly due to inefficient utilization of computing resources. However, it is crucial to note that not all code smell refactorings inevitably lead to enhanced resource utilization. Intriguingly, certain refactoring techniques and tools can inadvertently introduce new complexities. For instance, refactoring the *god class* smell involves decomposing a large class into smaller ones. This segmentation necessitates increased interclass communication, burdening the CPU with additional tasks, which in turn escalates CPU usage. Similar implications can occur when refactoring *god method* and *long parameter* smells, thereby exacerbating the application's CPU and memory consumption.

Previous research has predominantly focused on the impact of code smell refactoring on resource consumption within the context of smartphone applications [10]. However, these prior investigations have often been narrowly focused, considering only the isolated impact of a select few code smells and disregarding the potential cumulative effects of refactoring a broader array of smells [11]. Furthermore, these studies were limited in scope, examining only a small pool of applications, thus limiting their ability to form broad-based conclusions or predictions regarding their influence on energy consumption.

This paper fills a void in this area by providing a comprehensive analysis on the impact of batch refactoring 16 different code smell types on the resource consumption of 31 real-life Java and Python applications. We find that batch refactoring of code smells has a significant impact on the application's energy usage. Depending on the goal of the application developers, this study enables intelligent selection of which smells should be refactored together and which ones not be refactored. If software engineers are concerned about the resource consumption of their applications, this study will

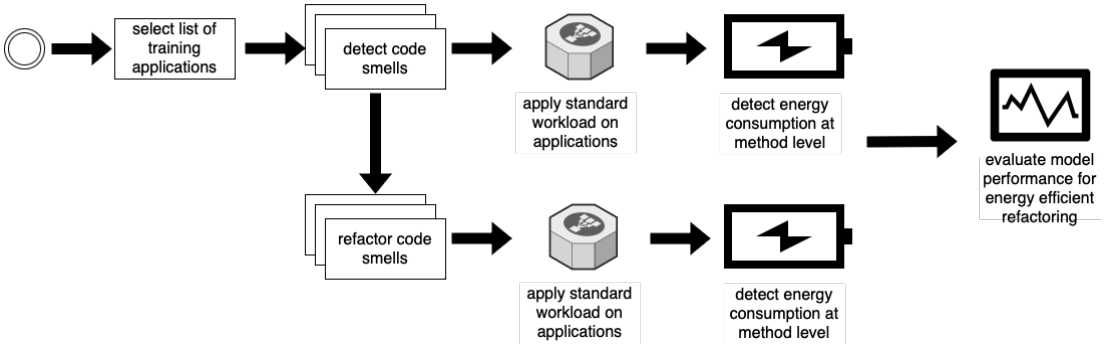


Fig. 1: Approach to selectively and intelligently refactor code smells.

help the developers decide which smells to refactor jointly to minimize the resource consumption. Also, we study if refactoring code smells impact the energy consumption of the software. We use the model presented in Section 2.3 to estimate the power and subsequent energy consumption before and after refactoring smells for a specific workload. Experiments on the 31 software show the impact of code smells on energy consumption.

In this study, we use 3 different automated refactoring tools, *Jdeodorant* [12] and *JSparrow* [13] for Java and *pycharm* for Python [14] applications to detect and refactor the code smells. We establish a benchmark where individual types of code smells are detected and refactored in each software, followed by an analysis of CPU and memory consumption and energy utilization impact. Afterward, we conduct a batch refactoring of smells and analyze their collective impact on energy usage. We provide an approach to selectively refactor code smells, which is illustrated in Figure 1.

The major contributions of this paper include the following:

- A detailed impact analysis of refactoring 16 different code smell types on the energy consumption of 31 real-world open-source Java and Python applications.
- An empirical evaluation of the change in energy utilization and carbon emission after auto-refactoring specific code smells in isolation as well as batch refactoring.
- A set of guiding principles to select the code smells that will improve energy usage when refactored collectively.

The rest of the paper is organized as follows: Section II explains the code smell types, the selected applications, workloads, automated refactoring tools, and energy estimation techniques used for this study. Section III presents the results of the experimental study and a summary of our findings. Section IV discusses the related work in this area, and Section V concludes the paper.

II. Methodology and Experimental Setup

Our analysis includes 16 different code smell types, and to the best of our knowledge, this is the most comprehensive study in this area so far. All selected smells can be detected and refactored using off-the-shelf automated refactoring tools. Table I summarizes the 16 code smells, including their properties, refactoring techniques, and their impact on application resource utilization. For automated smell detection and refactoring, we used *jdeodrant* [12] and *jsparrow* [13] for Java and *pycharm* for Python [14] applications to detect and refactor the code smells.

For each application, first, we compile and run the application without refactoring. In the process, we gather data regarding CPU and memory utilization and energy consumption. Second, we refactor them in two phases: in phase 1, we refactor all occurrences of one particular type of smell. In phase 2, we refactor multiple types of smells together to analyze the batch effect. Once all data is collected, we find the difference in CPU and memory utilization and energy consumption before and after refactoring. Next, we normalize the differences in resource usage by the instance of each type of smell that was detected. This gives us the per-smell impact of a specific type for each application. Next, we use the power model presented in Section 3.2 to estimate the power and energy consumption of the applications. Afterward, we use the carbon footprint calculator [21] to calculate the carbon emissions based on resource utilization. The carbon footprint estimation model considers several factors, which include hardware specifications, the number of cores in the CPU, and the total capacity of the memory. Additionally, it requires the time for which workloads are executed. Using runtime and data center location, empirical estimates are provided using parameter tuning and trial-and-errors [21].

For method-level data collection, we use a tool called *hprof* [22]. We record the execution path of the code and note the CPU and memory usage where the code smells are refactored. This allows us to collect information on resource usage precisely of the refactored method. As a result, we can relate the change in resource usage to refactoring. This is achieved by tracking resource usage via *method_id*, which is unique to a method and assigned by the $hprof$ tool. Using hprof, we collect resource usage data every 10 ms. For Python, we load the source codes in the *pycharm* and compile the code. Afterward, we apply specific workloads to test the resource usage before refactoring. When the applications run, we execute the workload and collect the resource usage using *logpid*. Next, we refactor the code smells in the same procedure discussed earlier and re-collect the data using the same workload. The workloads and experiments are detailed in the next section.

Next, we explain the applications and workloads that are used in our experiments.

Smell Type	Property	Refactoring Technique	Impact on Resource Utilization
cyclic dependency	Violates acyclic properties and results in misplaced elements [15]	Encapsulating all packages in a cycle and assign to single team	Refactoring prevents the enhanced loops from repeating, thus prevents resource wastage
god method	Many activities in a single method [4]	Divide the god method into multiple smaller methods	Multiple processes in a single method cause less inter-method communication, hence preserves resource usage
spaghetti code	Addition of new code without removing obsolete ones [16]	Replace procedural code segments with object-oriented design	Unrefactored code contains length() and size() can have a time complexity of O(n), refactoring results in using *isEmpty()* instead of *length()* and *size()* which has a complexity of *O(1)*
shotgun surgery	Single behavior defined across multiple classes[4]	Use Move Method and Move Field to move repetitive class behaviors into a single class	Refactoring removes the resource-consuming code blocks which were applied in multiple locations
god class	One class aims to do activities of many classes [4]	Divide the large class into smaller classes	Refactoring causes greater inter-class communication, thus increasing resource consumption
lazy class	The class does not do enough activity and can be easily replaced [4]	Use diamond operators to remove re-implementation of interface	Refactoring *lazy class* prevents the consumption of excess resource due to context switching from this class to the other classes
refused bequest	When the child classes of a parent are not related in any way, caused mainly by forceful inheritance [17]	Replace inheritance with delegation	Restructuring of code due to refactoring removes forceful inheritance, thereby preventing excess resource consumption
temporary field	When an instance variable is set only for certain cases [4]	Remove unnecessary throws and unused parameters	Refactoring results in removing temporary variables which act as additional fields ad consume CPU and memory in addition to the other variables
speculative generality	Codes which are placed by programmers for anticipated future events [18]	Eliminate the smell by boxing objects to use static strings.	As described above boxing the scalars to use the toString method is a waste of memory and CPU cycles.
dead code	Obsolete code which was not removed [19]	Parse through the methods to remove redundant code.	Code which is no longer needed keeps calling the methods and allocates spaces in memory and consumes CPU cycles. Refactoring removes this redundant code, thus stopping resource wastage
duplicate code	A code block which was copied in multiple classes rather than called through an object [4]	Apply proper inheritance	Removal of duplicate lines of code in multiple places will prevent CPU and memory from wastage
long parameter	When a method takes more than 5 parameters it is generally called to have a long parameter [18]	Simplify the lambda using method reference	Refactoring excessive number of parameters in a method will prevent caching at the beginning and this would remove the extra load on the CPU, however, memory usage will be increased
long statement	A statement in a code, e.g. a switch statement containing too many cases [20]	Divide the long statement into smaller statements and establish proper communication between those.	Refactoring prevents loading a long statement into memory which would otherwise consume excess memory resources.
primitive obsession	The undesirable practice of using primitive types when representing an object. [20]	Use $StringBuilder$ which ensures that no locking and syncing is done, resulting in faster operation.	Removes the use of obsolete string manipulation techniques like *StringBuffer* which allows locking but no synchronization, thereby saving CPU and memory resource
orphan variable	Variables that should be owned by another member class [20]	Extract all the variables to a class that should own them.	Refactoring ensures that a variable is transferred to a class to which it should belong, thus preventing wastage of CPU cycles and memory spaces while doing this communication
middleman	When a class is delegating almost all of its functionality to other classes [20]	Transfer functionality placed to the classes that they were mediating	Presence of such a delegation-centered class will create extra overhead in terms of resource consumption which is prevented by refactoring

TABLE I: Characteristics and refactoring techniques of the analyzed software smells.

A. Java: Applications and Workloads

For Java applications, in order to understand the impact of refactoring on resource utilization, the following workloads are run (clustered by application categories):

Email clients. The applications analyzed under this category are *emf [23]* and *columba [24]*. Predefined emails of size 70 bytes are sent using SMTP server [25]. The emails are sent to 2920 users who were identified as mail readers. The average time to deliver an email is 3083.03 milliseconds with a median of 2847.3 milliseconds.

Testing software. Eclipse bug dataset [26] is used as workload, which contains data about six applications that were analyzed, namely *jmeter [27], findbugs [28], cobertura [29], emma [30], jstock [31],* and *pmd [32]*. We merge all classes and files into one large dataset, which results in 24,642 LOC [33]. The workload consists of web services in Java, which consists of Java Server Pages (JSP), servlets, Enterprise Java Bean, and a database. The applications in the corpus are responsible for testing every conditional and loop statement.

Editors. We study seven applications here, which are *jedit [34], jhotdraw [35], antlr [36], aoi, galleon, batik [31],* and *jruby [37]*. Multiple bots conduct activities in the editor, such as typing, loading saved pictures, drawing simple shapes, and using various editor properties. The workload of each bot is 9.9 MB and a total of 109 virtual bots are used [38], [39]. The total time for the workload of all bots is 180 seconds.

Project management. The applications in this category include *ganttproject [40], xerces [41], javacc [42], nekohtml [31], log4j [43],* and *sablecc [42]*. Multiple bots conduct project management activities [44]. Three sample projects are

chosen: automated tender and procurement management, college management system, and resource monitoring system for ready-made garments. For each of the projects, bots check whether the project management tool is available all the time.

Parsers. The applications considered in this category are *ant [31], jparse [45],* and *xalan [41]*. The workload contains a set of incorrect and correct inputs [46], [47]. The incorrect input is fed into the parser and ensured that the correct error code is returned by the parser. For the correct input, the expected Abstract Syntax Trees (AST) are described in a format that can be correctly parsed. The AST of the correct input is verified by a third-party XML-based parser considered to be bug-free.

B. Python: Applications and Workloads

For Python applications, in order to evaluate the impact of refactoring on resource utilization, the following applications and workloads are run:

OpenStack. OpenStack is a Cloud platform providing users with virtual machine instances that can be used for Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS) [48]. It is the most popular open-source Cloud platform in both academia and industry [49]. To test the OpenStack source code, we compile it from the source code and launch VM instances. The OpenStack processes, including nova-compute, are allocated to a single node, and the resource consumption of that core is monitored.

Sentry. Sentry is a tool that reports and documents exceptions thrown by Python code running at the back-end servers. Sentry runs in the background and acts as a central hub to monitor and report errors. Test case workloads provided by $pytest$ are used for the experiments.

Tensorflow. To test *Tensorflow* we use the $tf.distribute.Strategy$ to run the "2017 US Internal Migration" dataset and train the system. The database contains 80 years of data and it is used to train the *Tensorflow* model to predict the internal migration trend for the next two years. The size of the dataset is 3.2 GB large, which contains detailed information regarding migration population, age, gender, occupation, and economic conditions.

Tornado. Tornado is a scalable framework with an asynchronous networking library primarily used for long-lived network applications. For Tornado, the inbuilt test suite is used to generate the data, which is used as a workload. The framework is synchronous, so the test results are completed when the method which is being tested returns.

Rebound. Rebound is a popular tool used by software engineers. It is a command-line tool written in Python that fetches all the solutions from stack overflow related to a problem. As a workload, we call the *sim.integrate(100.)* function, which presents 100 pre-specified erroneous code blocks in rebound and relies on it to fetch the solutions from stack overflow.

Kivy. Kivy is a Python library that is built over *OpenGL ES 2* that allows rapid development of multi-touch applications. The workload for Kivy is generated using its own module called *recorder*, which allows to replay of keyboard events in sample applications with Kivy running at the backend. A demo

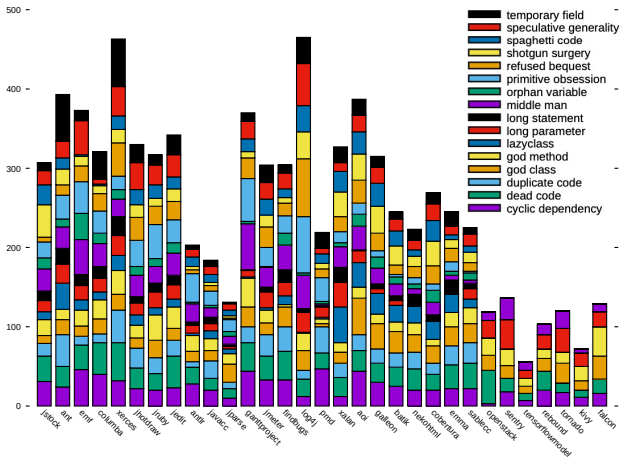


Fig. 2: Code smell distribution across the 31 applications analyzed in this study.

login page is launched with Kivy which simulates clicking on a login button. the *is_click* option in *recorderkivy.py* is set to true, and the screen coordinates for the click are specified. Next, the recorder is set to execute one click per second, and this is repeated for 2 hours. This workload ensures that the critical code segments of Kivy are called, a number of which also contain code smells.

Falcon. It is a WSGI library for building web APIs. The workload mainly includes simulating requests to a WSGI client through *class falcon.testing.TestClient(app, headers=None)[source]* class which is a contextual wrapper for *simulate_*()* function. This class simulates the entire app lifecycle in a single call, starting from the lifespan and disconnecting process. This workload is repeated by passing the number of repetitions to the *simulate_request()* function. It is repeated 300 times, and the CPU and memory usage are recorded. The process is repeated after refactoring the Falcon source code.

C. Energy Model

We use a power model to detect power consumption in watts based on the CPU utilization. We use average CPU utilization over time *t* and calculate average power consumption over that period. The time value shows the length of time for which the system was running to complete the workload. The following equations are used to calculate the power and energy estimates.

$$P_t = (C_{cpu} * U_{cpu}) * PBP \tag{1}$$

$$E_t = P_t * T \tag{2}$$

where,

- P_t = refers to the power consumption
- E_t = refers to the estimated energy consumption
- C_{cpu}:CPU co-efficient for single and multi-core processing
- U_{cpu}: Average CPU utilization to complete the task
- PBP_{ser}: Processor base power
- T = refers to the time in hours for execution of workload

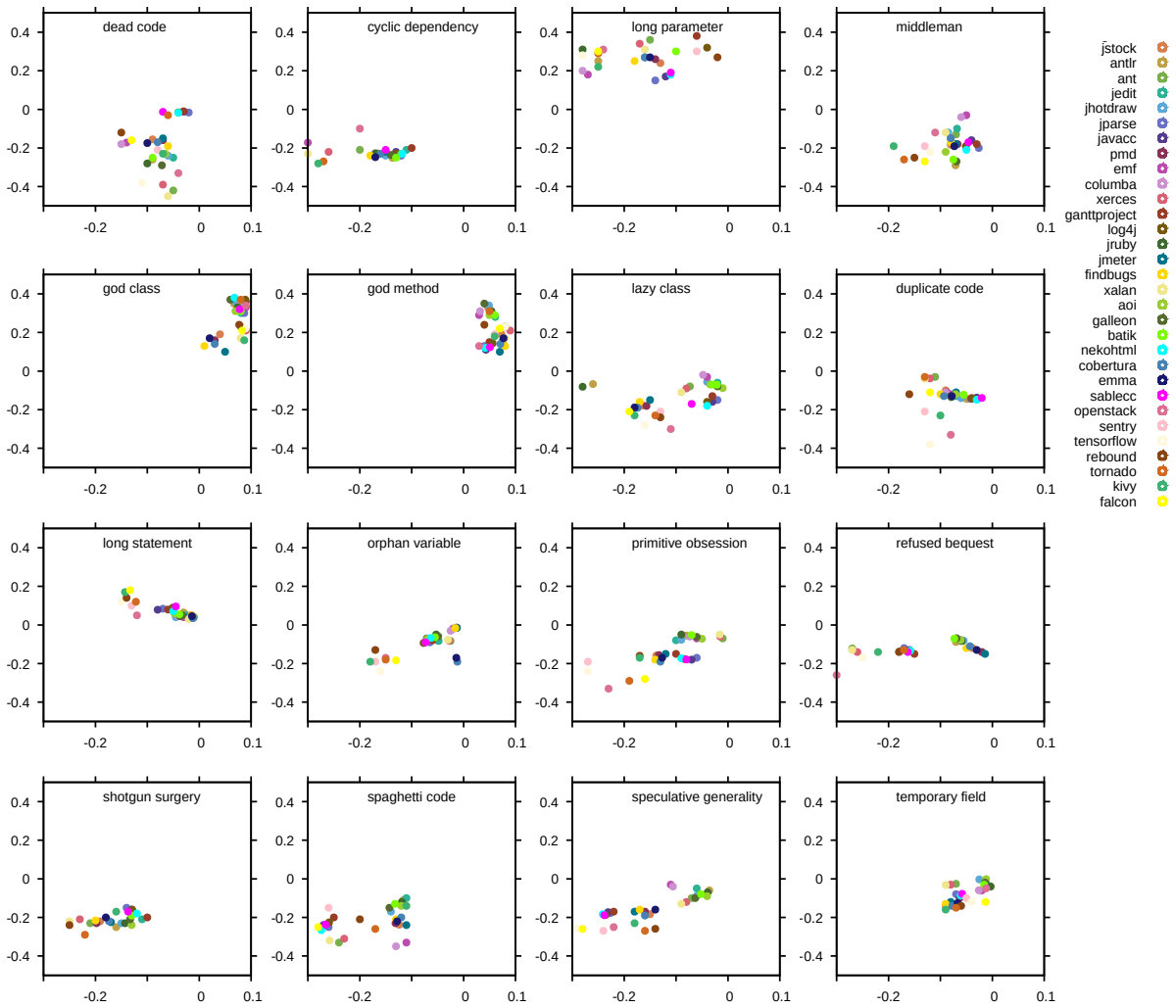


Fig. 3: Normalized plots of the impact of refactoring each code smell individually (per instance) on resource usage. X-axis: change in CPU usage (%); Y-axis: change in memory usage (%) of the application.

In server environments, accessing hardware information such as disk and network usage may not be possible. Also, users may not be able to connect power monitoring devices to the servers as those may be located in various regions compared to the users. Under current circumstances, it may be useful to estimate power consumption using CPU utilization. Earlier studies have shown a strong relationship between CPU and power. The correlation between CPU and power was found to be 87.81%. As a result, we adopt a CPU-based model that will give us considerably accurate results for our experiments, thus letting us measure the energy consumption of data transfer tools in a data center environment with high confidence.

We use the *Processor Base Power (PBP)* value to maximize the power consumed by the servers during the execution of the workload. *PBP* value is provided by CPU vendors, which indicates the the power drawn by a processor for a thermally significant period when it is at 100% peak utilization.

III. Experimental Results

In this section, we discuss the results of our experiments for both Java and Python applications. Figure 2 shows the frequency of 16 analyzed code smell types across all studied applications. The distribution shows that while the number of smells differs between applications, no single smell dominates. For example, *Cyclic Dependency* code smell is prevalent in the highest numbers in Java source codes as we detect 725 instances of this smell as seen in the Figure. On the other hand, 259 instances of *Orphan Variable* are detected. Given the assessment of smell distribution, we perform individual and batch refactoring of all applications, and we record the CPU and memory usage and the energy consumption before and after the refactoring. Figure 3 shows the relative change in resource usage we observed. Here, we define relative change as the difference in CPU and memory usage between before and after refactoring. The dataset for generating Figure 3 is provided [1] for reproducibility. We note that in the case of

[1] https://github.com/asif33/batchrefactoring/tree/scatter

Python applications, our tools can detect and refactor the following smells: *dead code, cyclic dependency, long parameter, middleman, god method*, and *god class*. Below, we summarize our findings for each smell type.

dead code: We know that *dead code* is a code that is either redundant because the results are never used or is never executed. Since the results are never used, but the code is getting executed, it is common to expect that it leads to CPU and memory waste. In cases when the code is not executed, it can still have adverse effects due to adding code bloat. Our results confirm that CPU usage can be improved by removing *dead code* smells. *Dead code* makes the runtime footprint larger than it needs to be, thereby consuming excess resources in terms of CPU and memory, which can be critical for large-scale data center applications like *OpenStack*.

cyclic dependency: This smell can cause a domino effect on the code when a small change in one module quickly spreads to other mutually recursive modules. The smell causes infinite recursion in 134 instances where it is found. In 63 instances, it results in memory leaks in Java by preventing garbage collectors from deallocating memory. The extent of the impact of this smell is also dependent on the type of software, as a similar type of application is found to behave similarly. Analysis of the refactored code shows that the refactoring eliminates the enhanced loops in most parts of the software, thereby improving resource usage. The enhanced loop traverses each loop one by one, thereby requiring increased CPU utilization even when traversal of the entire array may not be required. The refactoring tools address this and remove the enhanced loops.

Removing unwanted loops results in loop unrolling, which is observed in the refactored code of both Java and Python datasets. The loop unrolling reduces CPU and memory consumption by removing loop overhead. At the same time, loop control instructions and loop test instructions are eliminated, so the resources required to conduct those activities are freed. The total number of iterations is reduced to improve resource efficiency. As seen in the figure, in all cases of Python and Java, the removal of *cyclic dependency* code smell improves resource utilization performance.

long parameter: For *long parameter* code smell, it is seen that out of the 31 applications, all are showing positive memory change and negative CPU change, meaning memory usage degraded after refactoring the software smell. When we look towards refactoring, for example, in $jhotdraw$, the tool used "Introduce Parameter Object" refactoring.

If we consider an example of a *long parameter* smell found in $openstack$, we notice that a method with many parameters is refactored where the parameters are distributed to three methods, preserving the functionality. Although the above segregation is a better way to provide useful and reusable classes, it is causing the unboxing of the parameters from one to 3 methods. If one method contained all parameters, then all those parameters could have been cached at the beginning, and it does not require loading into memory multiple times. However, this would provide an extra load on the CPU as the parameters that are not required at the initial stage of polygon formation would still be called. Refactoring it in the mechanism described above breaks the concatenation, hence preventing caching.

Although the modularization of code improves readability, it worsens memory usage as more instructions need to be loaded into memory. Similar behavior applies to the remaining 3 applications which show these traits. For $openstack$ we notice that the CPU utilization reduces by 7.9%, which is significant compared to others. It must be stated that the number of smells of the *long parameter* in $openstack$ is found to be 40, significantly higher than the same smell being found in other apps. This large number of smells may have contributed to improving CPU usage.

middleman: Elimination of $middleman$ smells contributes to the improvement of CPU and memory usage. The most improvement is seen in $ganttproject$ with CPU and memory usage reductions of *0.61%* and *0.29%* respectively. There are *58* instances of $middleman$ code smell in the *ganttproject*, resulting in a significant performance improvement. Also, the *ganttproject* is a CPU-intensive project occupying a significant percentage of CPU when running, thus yielding greater change in CPU than memory. For the Python dataset, $sentry$ had the maximum number of $middleman$ code smells detected. When the 27 code smells in $sentry$ are refactored, per smell improvement in CPU and memory is 0.44% and 0.13%, respectively, for every smell refactored.

god class: In the list of applications that are refactored, it is seen that the extract class refactoring mechanism caused the resource usage to worsen [50]. Refactoring this smell involves a large class being separated into multiple smaller classes, each with lesser responsibilities, hence extra time and resources are required for inter-class communication. As a result, CPU and memory usage increases. Further analysis shows that inter-class communication increases as large classes are extracted into multiple small classes. We inspect all the new methods that are created and find the average lines of code in those. In most cases, we see that usually 16.14 lines of code trigger and complete operations on a variable or object, based on slicing, a new method needs to be made with those. From a software engineering perspective, such large volumes of extraction are desirable. However, from the standpoint of resource usage, such granular segregation may cause a huge volume of context switching and inter-method communications, which may add a high volume of overhead.

god method: The behavior of the graph for the *god method* is similar to that of the *god class*. All values are positive, which shows that refactoring the *god method* code smells increases resource usage. Besides, the normalized increase is quite high for the god method compared to other kinds of code smells. To refactor the *god method*, the extract method mechanism is used. So, a large method is broken down into multiple smaller methods, which increases inter-method communication.

lazy class: The same behavior is seen for refactoring *lazy class* smell where each category of applications is showing similar behavior. One exception is that JRuby is located very close to the group of document editors. The reason is that

the number of *lazy class* smells of JRuby is only 9. Similar resource consumption changes are seen for the group of editors, where the number of smells ranges from 9-13 for all the applications. Hence, for $lazy\ class$, the number of smells is proportional to the impact on resource usage.

duplicate code: Similar behavior is seen for refactoring *duplicate code* smell. It is seen that the apps belonging to the same category are behaving similarly, emphasizing the fact that similar types of apps have the same impact when the code smell is refactored. Ant, $xalan$, $maven$, and $xerces$ are found to show significant improvement in CPU resources after refactoring. Analysis of the code in $xerces$ shows that it parses the XML documents and places the variables in those in a list, reiterating through it multiple times.

orphan variable: Similar categories of applications are seen to behave similarly in terms of change in resource usage when the *orphan variable* is refactored. As a result, it can be stated that the category of applications can be used to group the impact of refactoring the code smell. The email clients, namely emf and $columba$, are seen to have the maximum impact of refactoring this code smell.

primitive obsession: After refactoring the *primitive obsession* code smell, it is seen that for *primitive obsession*, the change in resources data can be used to group the applications by category. One of the rules used by the refactoring tool is to replace $StringBuffer$ with $StringBuilder$. It is recommended to use $StringBuilder$ because no locking and syncing is done. Hence, it is faster. When running programs in a single thread, which is generally the case, $StringBuilder$ offers performance benefits over $StringBuffer$.

refused bequest: We see that the impact of automated refactoring is higher for the group of code analyzer apps than the others. This is because the testing apps loaded the source code in memory to run the tests. The presence of unused methods and variables in the code which is loaded into memory results in excessive resource usage by the applications. On average, after refactoring 0.284% of CPU and 0.147% of memory are reduced for each *refused bequest* smell refactored.

shotgun surgery: For log4j, it is seen that refactoring the *shotgun surgery* significantly contributes to improving memory resources by 7%. Refactoring this code smell also improves the unpredictability and efficiency of the generated random values. Simplification of the data structures occurred in 21 of the cases of refactoring, a high percentage of 61.76% where this refactoring is done, thus simplifying the code significantly. As most of the loops are used to read and load the logs in memory, simplifying it meant that less memory is required for loading.

spaghetti code: The $jruby$ application has the highest impact due to refactoring the *spaghetti code* smell. The number of *spaghetti codes* detected in this application is 57, which is higher than any application in the list. This results in more loc being refactored and greater change in resource usage before and after refactoring. One of the rules of refactoring *spaghetti code* replaces the $concat()$ method on Strings with the + operator. It has slight performance benefits if the size of the $concat()$ is large. Another rules replaces $length()$ or $size()$ with $isEmpty()$. This rule provides performance advantages since $isEmpty()$ time complexity is $O(1)$ whereas $length()$ and $size()$ can have a time complexity of $O(n)$.

speculative generality: It is seen that the code parser category showed the highest change in CPU and memory utilization for *speculative generality*. This category of applications has 76 cases of *speculative generality* and non-normalized CPU usage improves by 4.63% and memory usage improves by 1.47% due to refactoring of the smells. This increase is mainly due to the removal of excess code that was added but not called in the system. These codes keep using heap memory and CPU for basic, non-required computations.

temporary variable: Given the lower number of smells detected for this smell type in the applications, the grouping of applications in the plots based on category implies that the smell is having an impact on the resource change. Also, the refactoring does not keep temporary fields, thus making those final, leading to improvement in resource consumption.

A. Impact of Batch Refactoring

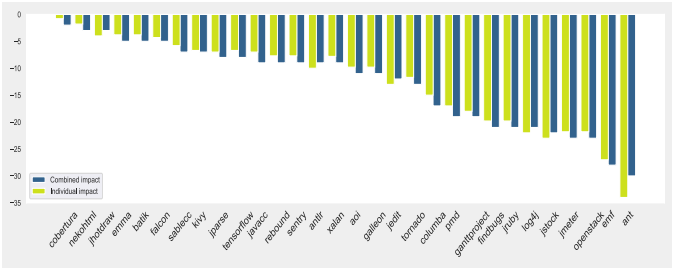
In addition to analyzing the individual impact of code smells on resource usage (as presented in the previous section), we also study the combined impact of code smells and their refactoring on resource usage. In this study, we investigate whether the combined impact is equal (or similar) to the summation of the individual impacts of refactoring different smell types. With this requirement in mind, we refactor the smells in batches and present the results in Figures 4, 5, and 6.

1) Refactoring All Code Smells Together:

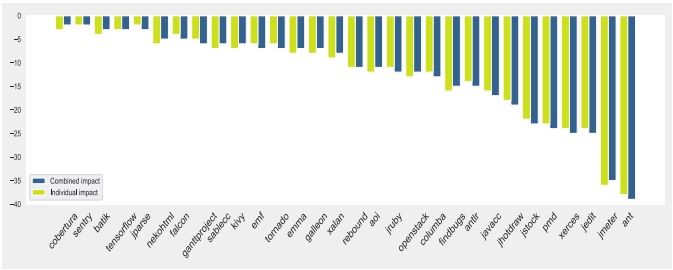
First, we refactor and analyze the combined impact of all 16 code smells. In this section, we provide the findings in terms of CPU and memory utilization (%), energy consumption (Joules), and carbon emission (gCO2e). Figure 4 shows the combined impact of refactoring all smells together.

Impact on CPU: Combined refactoring of all the smells, irrespective of impact, can provide useful information as to whether those smells improve resource usage or worsen those. It is seen that although the individual impacts of performance degrading smells are significant, refactoring all 16 smells in 31 applications together resulted in an improvement of the resource usage overall since the type of resource usage improving smells are larger compared to those which worsen performance. From the CPU perspective, it is seen that the total CPU usage of ant improved by 30.01% which is significant and desirable. At the same time, the least percentage improvement was seen in $Javacc$ which is 8.10%. It is seen that the presence of the numbers of various types of smells greatly influences the percentage improvement of CPU.

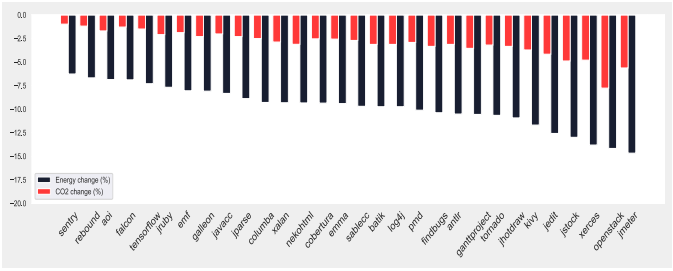
Impact on memory: A similar pattern is seen for memory consumption, where the usage improves after refactoring the 16 smells together, where ant shows the highest improvement of 39.70%. The lowest change in CPU usage is seen for $jparse$ with a 3.50% improvement. Again the increase can be credited to the total instances of the various types of smells found in the un-refactored code.



(a) impact on CPU utilization(%)



(b) impact on memory utilization (%)



(c) impact on energy consumption (%) and carbon emission (%)

Fig. 4: Combined impact of refactoring all code smells considered in this study on the application's CPU and memory utilization (%), energy consumption (%), and carbon emission (%).

Impact on energy consumption and carbon emission: Figure 4c shows the energy savings by refactoring all 16 code smells together. It is seen that energy savings in $OpenStack$, ant, and $Jmeter$ range between 76-139 joules. Average energy savings account to 6.01% (88.46 joules) for the 31 software considered here. The blue bars in the plot are used to show energy consumption (in Joules), and the green bars show carbon emission estimations (in grams of carbon dioxide equivalent - gCO2e) for each of the 31 applications. Negative values show an improvement (decrease) in energy consumption, whereas positive values show an increase in energy consumption. It is seen that 15.98 grams of carbon emission can be saved on average by refactoring the 16 code smells in the 31 software.

2) Refactoring Smells That Decrease Resource Usage:

This section states the combined impact of refactoring the smells that positively impact resource usage. Similar to the last section, we highlight the impact on CPU and memory utilization (%), energy consumption, and carbon emission (gCO2e) separately and analyze the results as shown in Figure 5.

Impact on CPU: We analyze refactoring of smells that consistently improve the resource utilization for our application set and found that *cyclic dependency, duplicate code, dead code, primitive obsession, speculative generality, shotgun surgery, long parameter, middle man, refused bequest, orphan variables, long statements*, and *temporary fields* meet our condition. We proceed to refactor the aforementioned smells altogether and determine the total change in resource utilization when they are refactored together. Out of the 31 applications, $columba$, $log4j$, and $jruby$ give errors when the smells which improved performance individually are refactored together. As shown in Figure 3 for CPU, it is seen that the range of percentage improvement of CPU stretches from 7.6% for $jparse$ till 37.70% for ant. We proceed to sum up the impact of refactoring those smells individually for comparison purposes. It is seen that combining impact stretches from 7.86% to 38.87%. Upon calculation of the differences, it is seen that combining all the smells whose refactoring improves performance shows consistent behavior to refactoring them individually and adding the values. The difference in the values ranged from 0.26% to 1.46% which are seen for $jparse$ and emf, respectively. The mean deviation is 0.61%.

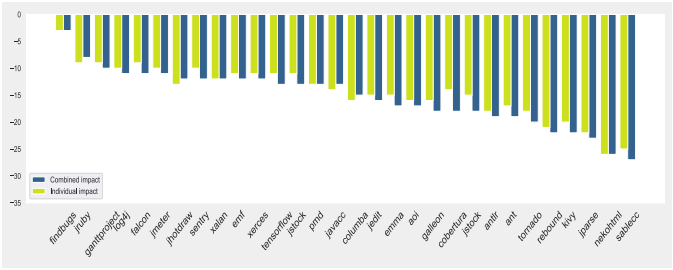
Impact on memory: Memory usage is impacted significantly, with a range of 25.47% and 47.77% for the apps and an average improvement of 28.63%. It is seen that $jmeter$ shows the maximum improvement, whereas emf shows the minimal effect on memory. Further analysis shows that when the smells are refactored in $jmeter$, the spatial locality volume increases due to the rearrangement done to it. Also, compression is conducted by dissolving longer parameters, which results in smaller and smarter formats. Finally, temporal localities are increased by refactoring smells like a *refused bequest, shotgun surgery,* and *speculative generality*, which cache trashing, hence reducing memory usage. Reduce and reuse refer to techniques that minimize memory operations with the temporal locality that reduce cache fetches. This is accomplished by reusing data still in the cache by merging loops that use the same data with a mean deviation of 0.64%.

Impact on energy consumption and CO2 emission: Figure 5c shows the energy consumption and carbon emission impact of refactoring the code smells which improve resource usage. A positive correlation is observed between energy consumption and improvements in the CPU and memory utilization. It is seen that energy consumption decreases by 8.71% (128.22 joules) on average. $Falcon$, Aoi, $Jedit$, and $Findbugs$ show the greatest improvement in energy consumption when the code smells are removed. The carbon footprint improves by 4.31% (25.51 grams) on average for the 31 applications. Hence, it is seen that selectively refactoring the code smells improves energy consumption and carbon emissions.

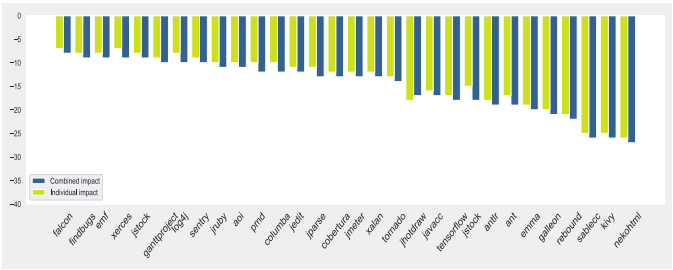
3) Refactoring Smells That Increase Resource Usage:

In this section, we present the combined impact of refactoring the smells that negatively impact resource usage.

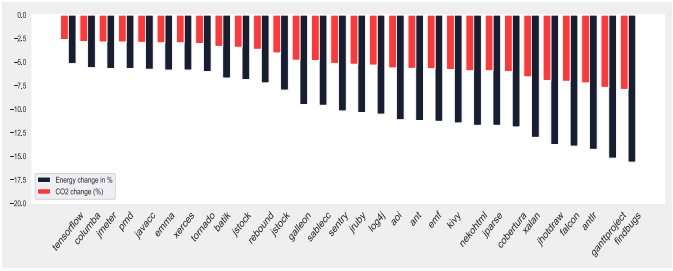
Impact on CPU: It is seen that refactoring *god class, god method*, and *feature envy* negatively impacts performance when refactored. Upon analysis of the normalized graph for *god class* and *god method*, it is seen that per smell impact of *god class* is found to be around 0.22%-0.50%, whereas for *god class* it is 0.20%-0.22%, indicating that a software



(a) impact on CPU utilization (%)

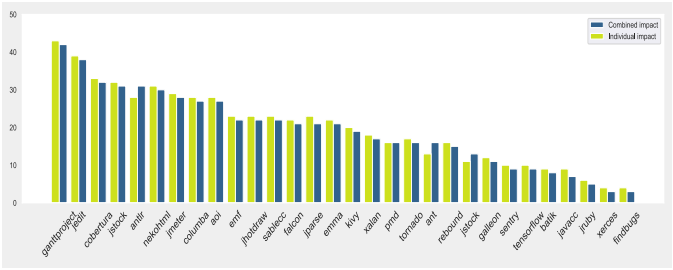


(b) impact on memory utilization (%)

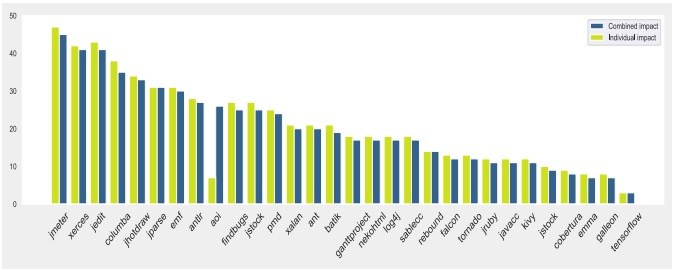


(c) impact on energy consumption (%) and carbon emission (%)

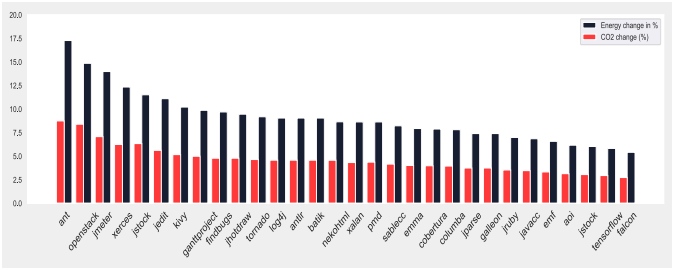
Fig. 5: Combined impact of refactoring code smells that improve resource usage on the application's CPU and memory utilization (%), energy consumption (%), and carbon emission (%).



(a) impact on CPU utilization (%)



(b) impact on memory utilization (%)



(c) impact on energy consumption (%) and carbon emission (%)

Fig. 6: Combined impact of refactoring code smells that worsen resource usage on the application's CPU and memory utilization (%), energy consumption (%), and carbon emission (%).

engineer who is focusing on refactoring and has optimizing resource usage and energy consumption in mind should avoid refactoring *god classes* and *god methods*.

At the same time, it is seen that *ganttproject* suffers from the largest percentage increase in CPU usage, which is undesirable. In total *ganttproject* has 61 occurrences of smells of *god class* and *god method*, refactoring which greatly impacts the resource usage, with a degradation of 16.30%. On average, the total degradation of CPU usage after refactoring the smells for 31 applications is found to be 7.79%. Upon refactoring the individual smells and adding the total change in CPU usage, we get similar values to refactoring them altogether.

Impact on memory: Refactoring *god class* and *god method* also worsen memory usage for all 31 applications. *Log4j* has the highest degradation of memory consumption, which is 19.50% when the concerned smells are refactored altogether. Also, refactoring those individually and adding up the values results in a total memory consumption of 20.01%, which is only 0.51% greater than combined refactoring. This ensures that the results add up to individual impact, and hence, are consistent. The large number of smells of *god classes* and *god methods*, which sum up to 100 instances of smells being present in the code, leads to a large volume of refactoring done in the code by implementing extract class and extract method refactoring procedures. This contributes to the large distortion of memory usage before and after refactoring. Overall, we see that individual refactoring impacts add up when the refactoring is done in a combined procedure. The mean deviation in CPU utilization is 0.64% and the mean deviation in memory utilization is 1.47%.

Impact on energy consumption and CO2 emission: Figure 6c shows the impact of refactoring the code smells which increase resource usage on energy consumption and carbon emission. A positive correlation is observed between energy consumption and resource usage increase when certain code smells are refactored. It is seen that energy consumption increased from 23-88 joules across the 31 applications when specific code smells are refactored. Average energy wastage accounts for 10.01% for the 31 software considered here. It is seen that *Ant*, *Jemeter*, and *OpenStack* emit the result in the highest carbon emissions in this case. The carbon emission worsens by 3.16% on average for the 31 applications. This illustrates the importance of selective refactoring code smells when building energy-intensive applications.

IV. Related Work

Automated batch refactoring techniques are known to significantly improve overall software quality and maintainability, but their impact on energy utilization and carbon emission is not well studied in the literature. 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 [51] of Android applications. 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 batch refactoring code smells on software resource usage was not considered.

To understand the relationship between Android code smells and nonfunctional factors like energy consumption and performance, Palomba et al. [17] conducted a study with nine Android-specific smells and 60 Android applications. Their results showed that some smell types cause much higher energy consumption compared to others and refactoring those smells improved energy consumption in all cases. Although the results are consistent with our findings, the authors only addressed the individual impact of nine code smells, and the analyzed smells were specific to Android applications.

The impact of multiple refactoring on code maintainability, also known as batch refactoring, was explored by Bibiano et al. [52]. They argue that removing an individual code smell in a code block increases the tendency to introduce new smells by 60%. Therefore, the importance of analyzing the combined and complex impact of refactoring code smells in a batch rather than individual smells is proposed. Şanlıalp et al. [53] also emphasized studying the combination of multiple refactoring techniques and its impact on energy combination of software. Besides maintainability, it is also essential to study the effect of batch refactoring on the resource usage of the application.

Park et al. investigated whether existing refactoring techniques support energy-efficient software creation or not [9]. Since low-power software is critical in mobile environments, they focused their study on mobile applications. Results show that specific refactoring techniques like the *Extract Class* and *Extract Method* can worsen energy consumption because they did not consider power consumption in their refactoring process. The goal was to analyze the energy efficiency of the refactoring techniques themselves, and they stated the need for energy-efficient refactoring mechanisms for code smells.

Pérez-Castillo et al. stated that excessive message traffic derived from refactoring *god class* increases a system's power consumption [54]. It was observed that power consumption increased by 1.91% (message traffic = 5.26%) and 1.64% (message traffic = 22.27%), respectively, for the two applications they analyzed. The heavy message-passing traffic increased processor usage, which proved to be in line with the increase in power consumption during the execution of those two applications. The study was limited to only *god class* code smell. However, a detailed analysis is required to determine the impact of code smell refactoring on resource consumption.

An automatic refactoring tool that applied the *Extract Class* module to divide a *god class* into smaller cohesive classes was proposed in [12]. The tool aimed to improve code design by ensuring no classes are large enough, which is challenging to maintain and contains a lot of responsibilities. The tool refactored code by suggesting *Extract Class* modifications to the users through a User Interface. The tool was incorporated into the Eclipse IDE via a plugin. The authors consulted an expert in the software quality assessment field to give his expert opinion to identify the effectiveness of the tool. Results show that in 12 cases (75%), the evaluator confirmed that the classes suggested being extracted indeed described a separate concept. According to the expert, two of these classes could be extracted and used as utility or helper classes. However, the effect of such refactoring on resource usage of the software was considered to a limited extent.

V. Conclusion

In this paper, we evaluate the impact of batch refactoring 16 code smells on the resource usage of 31 open-source Java and Python applications. We provide a detailed empirical analysis of the change in the CPU and memory utilization, energy consumption, and carbon emission after auto-refactoring specific code smells in isolation as well as in combination with other smells. Obtained results highlight that the refactoring techniques adopted for code smells such as *god class* and *god method* adversely affect CPU and memory usage as well as energy consumption and carbon emission of the application. Refactoring *Long Parameters* smell results in improvement of CPU usage but worsens memory usage. Refactoring all other code smells improves resource usage for the same workload. We notice that applications belonging to the same category are impacted similarly by refactoring specific smells. Also, the impacts of smells on resource consumption for Java and Python applications are quite similar; hence, our results can be generalized. Combined refactoring of various code smells adds up to the impact of refactoring those smells individually.

Our study provides compelling evidence that various common refactoring techniques, typically employed to rectify specific code smells, can unintentionally escalate the application's energy consumption. We illustrate that a discerning and strategic approach to code smell refactoring can yield substantial energy savings. For selective refactoring, this yields a reduction of up to 13.08% of energy consumption per workload. We also obtain a carbon emission reduction of 5.10% for specific applications. These findings underscore the potential of selective and intelligent refactoring to substantially enhance the energy efficiency of software applications.

Based on these observations, we suggest a set of guiding principles for selecting the correct set of code smells to be refactored for the most efficient resource utilization.

Acknowledgements

This project is in part sponsored by NSF under award numbers 2343284 and 2343285.

REFERENCES

[1] L. Belkhir and A. Elmeligi, "Assessing ict global emissions footprint: Trends to 2040 & recommendations," *Journal of cleaner production*, vol. 177, pp. 448–463, 2018.

[2] A. S. Andrae and T. Edler, "On global electricity usage of communication technology: trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.

[3] R. Memon, "Calculating software carbon intensity," *URL https://www.thoughtworks.com/en-us/insights/blog/ethical-tech/calculating-software-carbon-intensity*, 2023.

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

[5] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 211–221.

[6] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 125–136.

[7] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 233–243.

[8] E. Murphy-Hill, "Scalable, expressive, and context-sensitive code smell display," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008, pp. 771–772.

[9] J. J. Park, J.-E. Hong, and S.-H. Lee, "Investigation for software power consumption of code refactoring techniques." in *SEKE*, 2014, pp. 717–722.

[10] 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.

[11] 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.

[12] 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.

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

[14] H. Gulabovska and Z. Porkoláb, "Survey on static analysis tools of python programs." in *SQAMIA*, 2019.

[15] 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.

[16] 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.

[17] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.

[18] 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.

[19] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[20] A. Shvets, "Java code smells, https://refactoring.guru/, 2021."

[21] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: quantifying the carbon footprint of computation," *Advanced science*, vol. 8, no. 12, p. 2100707, 2021.

[22] K. O'Hair, "Hprof: a heap/cpu profiling tool in j2se 5.0," *Sun Developer Network, Developer Technical Articles & Tips*, vol. 28, 2004.

[23] R. M. Santos, M. C. R. Junior, and M. G. de Mendonça Neto, "Self-admitted technical debt classification using lstm neural network," in *ITNG 2020*. Springer, 2020, pp. 679–685.

[24] F. Dietz and T. Stitch, "Columba email client project," *URL: http://columba.sourceforge.net/testing/index.php(visited on 2/11/2021)*, 2017.

[25] L. Riungu-Kalliosaari, O. Taipale, and K. Smolander, "Testing in the cloud: Exploring the practice," *IEEE software*, vol. 29, no. 2, pp. 46–51, 2011.

[26] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 9–9.

[27] E. H. Halili, *Apache JMeter*. Packt Publishing Birmingham, 2008.

[28] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 805–806.

[29] J. Aarniala, "Instrumenting java bytecode," in *Seminar work for the Compilerscourse, Department of Computer Science, University of Helsinki, Finland*, 2005.

[30] Y. Y. Liu, B. Hu, L. P. Rao, and L. Pan, "Java code coverage test technology based on emma," in *Advanced Materials Research*, vol. 1049. Trans Tech Publ, 2014, pp. 2069–2072.

[31] 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.

[32] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th International symposium on software reliability engineering*. IEEE, 2004, pp. 245–256.

[33] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 35–39.

[34] M. Wenzel, "Isabelle/jedit–a prover ide within the pide framework," in *International Conference on Intelligent Computer Mathematics*. Springer, 2012, pp. 468–471.

[35] J. Savolskyte, "Review of the jhotdraw framework," *Harlow, Information and Media Technologies*, 2004.

[36] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[37] C. Nutter, T. Enebo, O. Bini, N. Sieger *et al.*, "Jruby," *URL: http://jruby.org/(visited on 12/11/2013)*, 2014.

[38] F. Jacob, D. Hou, and P. Jablonski, "Actively comparing clones inside the code editor," in *Proceedings of the 4th International Workshop on Software Clones*, 2010, pp. 9–16.

[39] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Communications of the ACM*, vol. 21, no. 9, pp. 760–768, 1978.

[40] S. Cromar, "Ganttproject," in *From Techie to Boss*. Springer, 2013, pp. 225–229.

[41] T. W. Leung, *Professional XML Development with Apache Tools: Xerces, Xalan, FOP, Cocoon, Axis, Xindice*. John Wiley & Sons, 2004.

[42] A. J. Dos Reis, *Compiler Construction Using Java, JavaCC, and Yacc*. John Wiley & Sons, 2012.

[43] W. Z. Liu, Q. Y. Tao, Q. He, and L. J. Yu, "Application of log4j in e-commerce services," in *Applied Mechanics and Materials*, vol. 635. Trans Tech Publ, 2014, pp. 1517–1521.

[44] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A berkeley view of cloud computing," *UC Berkeley EECS Technical Report EECS-2009-28*, 2009.

[45] J. James, "Jparse: A java parser," *Retrieved (11 September 2004) from http://www. ittc. ku. edu/JParse*.

[46] L. Duong, H. Afshar, D. Estival, G. Pink, P. R. Cohen, and M. Johnson, "Multilingual semantic parsing and code-switching," in *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*, 2017, pp. 379–389.

[47] M. D. Ćirić and S. R. Rančić, "Parsing in different languages," *Facta universitatis-series: Electronics and Energetics*, vol. 18, no. 2, pp. 299–307, 2005.

[48] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[49] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[50] K. Alkharabsheh, Y. Crespo, M. Fernández-Delgado, J. R. Viqueira, and J. A. Taboada, "Exploratory study of the impact of project domain and size category on the detection of the god class design smell," *Software Quality Journal*, pp. 1–41, 2021.
[51] 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.
[52] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.
[53] İ. Şanlıalp, M. M. Öztürk, and T. Yiğit, "Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices," *Electronics*, vol. 11, no. 3, p. 442, 2022.
[54] 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.