Build Code Needs Maintenance Too: A Study on Refactoring and Technical Debt in Build Systems

1st Anwar Ghammam
Oakland University
Rochester Hills, USA
anwarghammam@oakland.edu

2nd Dhia Elhaq Rzig
University of Michigan- Dearborn
Dearborn, USA
dhiarzig@umich.edu

3d Mohamed Almukhtar
University of Michigan- Flint
Flint, USA
almukhtr@umich.edu

4th Rania Khalsi
University of Michigan- Flint
Flint, USA
rkhalsi@umich.edu

5th Foyzul Hassan
University of Michigan- Dearborn
Dearborn, USA
foyzul@umich.edu

6th Marouane Kessentini
Grand Valley State University
Grand Valley, USA
kessentm@gvsu.edu

Abstract—In modern software engineering, build systems play the crucial role of facilitating the conversion of source code into software artifacts. Recent research has explored high-level causes of build failures, but has largely overlooked the structural properties of build files. Akin to source code, build systems face technical debt challenges that hinder maintenance and optimization. While refactoring is often seen as a key tool for addressing technical debt in source code, there is a significant research gap regarding the specific refactoring changes developers apply to build code and whether these refactorings effectively address technical debt.

In this paper, we address this gap by examining refactorings applied to build scripts in open-source projects, covering the widely used build systems of Gradle, Ant, and Maven. Additionally, we investigate whether these refactorings are used to tackle technical debts in build systems. Our analysis was conducted on 725 examined build-file-related commits. We identified 24 build-related refactorings, which we divided into 6 main categories. These refactorings are organized into the first empirically derived taxonomy of build system refactorings. Furthermore, we investigate how developers employ these refactoring types to address technical debts via a manual commit-analysis and a developer survey. In this context, we identified 5 technical debts addressed by these refactorings and discussed their correlation with the different refactorings. Finally, we introduce BuildRefMiner, an LLM-powered tool leveraging GPT-4o to automate the detection of refactorings within build systems. We evaluated its performance and found that it achieves an F1 score of 0.76 across all build systems.

This study will serve as a foundational building block for guiding future research and practice in the maintenance and optimization of build systems. BuildRefMiner and the replication package for this study are available at [1]

I. Introduction

Build systems are responsible for transforming source code into executable programs by coordinating the execution of various tools, ranging from compilers to code analyzers [2]–[4]. Build systems, such as Maven [5], Ant [6] and Gradle [7] are commonly employed in the development of large software projects to automate the process of compiling, packaging, and testing software products. However, with this wide range of capabilities and flexibility, a lot of complexity can emerge in build code. Indeed, prior research [8]–[11] has demonstrated

how configuring build systems can frequently lead to challenges in their maintenance and result in delays in software development projects. Seo et al. [10] demonstrated that up to 37% of builds conducted at Google experience failure, while Kumfert et al. [11] further estimate that build maintenance imposes a 12% overhead on the development process, distracting developers from their main tasks. This highlights the importance of a robust, clean, and well-maintained build system to facilitate seamless development tasks, thereby preventing them from becoming arduous and time-consuming, consequently affecting the overall efficiency of the software.

Refactoring, defined as the restructuring of existing code to improve its quality without altering its outward behavior [12], is frequently proposed as a method to achieve this objective. However, Refactoring build systems, despite their importance in software development, remains an area with limited understanding [13]–[15]. To the best of our knowledge, no empirically validated taxonomy of build refactorings currently exists. Moreover, no analysis has been conducted to connect build refactorings to the technical debts they may mitigate. Technical debt (TD) is a metaphor that describes the lower-quality code, which represents a trade-off between the short-term benefits of rapid delivery and the long-term value of software [16].

Currently, practitioners have limited access to specific guidance on how to apply build-related refactorings and organize their build code. Knowing the types of refactorings and TDs that may occur within build systems can shape developer guidance when it comes to maintaining their existing build files and can support the development of tools that can automatically detect and recommend refactoring opportunities. To highlight the relevance of refactorings in build files, we provide the example in Listing 1, which demonstrates a Don't Repeat Yourself (DRY) refactoring applied to a Gradle build file, to address Code Duplication technical debt. Here, 'Connect' and 'Insert' tasks are streamlined using a loop, which reduces repetitive task definitions, thus minimizing redundancy by avoiding repetitive setups for each task, making the code more

maintainable.

```
1  ['Connect', 'Insert'].each { taskName ->
2  task "$taskName" (type: JavaExec) {environment 'username',..}
3  task('Connect', type: JavaExec) {environment 'username', ..}
4  task('Insert', type: JavaExec) {environment 'username', ..}
```

Listing 1: Commit *560850d* in *biginsight-examples* Project: An Example of **DRY** Refactoring Type in Gradle.

In this paper, we address the knowledge gap on build refactoring types by conducting an empirical study on build refactorings in open-source projects. Our analysis includes building a taxonomy of build-related refactorings, an investigation into which TDs these refactorings address, and the creation of BuildRefMiner to automatically detect build refactorings in past commits. In this context, we address the following research questions:

RQ1: Which refactoring types are developers applying to build code? This RQ aims to build a taxonomy of build refactorings. We were able to develop a taxonomy of 24 build refactoring types classified into 6 main categories. 8 refactoring changes are Build-specific such as Dependency Organization and Synchronizing Shared Build Properties refactorings.

RQ2: How are build refactorings linked to technical debt? This RQ aims to uncover which TDs can motivate developers to implement build refactoring operations. In total, we were able to extract 5 TDs linked to refactoring categories. For example, DRY addresses the TD Code Duplication & Redundancy.

RQ3: What is the effectiveness of our tool BuildRefMiner in identifying build refactorings? To provide guidance to future research, we developed BuildRefMiner, to automatically detects build refactorings in the commit history of a project. We were able to achieve an F-1 score of 0.76 in detecting build refactorings.

The main contributions of this paper are:

1) The first dataset on refactorings in Build systems. The dataset can be further utilized in future research on developing tools and techniques for detecting and recommending refactoring opportunities.

2) The first quantitative and qualitative study on refactoring changes in multiple build systems: Maven, Ant, and Gradle. We propose a rich taxonomy of a total of 24 refactorings divided into 6 main build refactoring categories. Furthermore, we link 20 of these categories with TDs they address via 85 commit messages and 60 developer responses.

3) BuildRefMiner, which we develop for automatic build-refactoring identification, enabling researchers to detect refactoring patterns within build systems more efficiently.

II. BACKGROUND

A. Build Systems

Build automation is a critical part of modern software development [8]. Build tools provide flexible ways to model, manage, and maintain complex software projects. They rely on configuration files to define dependencies, specify build tasks, and orchestrate build process workflows [17]–[22]. Within this work, we specifically focus on Ant, Maven, and Gradle, widely-used build tools that provide a good representation of the different approaches to build automation.

Ant uses an XML-based highly customizable build script to define tasks and dependencies. Ant does not have predefined build lifecycle phases. Instead, build targets can be connected to indicate dependencies. **Maven** also uses an XML build script, centered around its project object model (POM.xml), which defines project configuration, dependencies, and settings. Rather than customizable tasks, Maven relies on plugins bound to these phases to execute goals in a consistent sequence. Finally, **Gradle** builds on Ant and Maven using a Groovy or Kotlin domain-specific language (DSL) for its build scripts. This allows flexible modeling of build requirements in a declarative, extendable way. Gradle build files consist of components like plugins, dependencies, configurations, tasks[1], and properties.

While Ant, Gradle, and Maven are commonly associated with Java, these tools support other programming environments, such as Kotlin, PHP, and Android. The inclusion of such projects, detailed in our appendix [1], ensures our findings are representative of diverse ways of build-systems usage in practice.

B. Refactoring and Technical Debt

Technical debt is defined as the cost of additional work created by choosing an easy solution now instead of using a better approach that would take longer [23]. It is a metaphor that describes the consequences of poor software design and implementation decisions. Technical debt can manifest in various forms in Build files, such as design flaws, and outdated dependencies. Over time, technical debt can accumulate and slow down development, increase maintenance costs, and reduce software quality [23]. Refactoring is defined as the process of restructuring existing code without changing its external behavior to improve readability, maintainability, and extensibility [23]. Refactoring can help reduce technical debt by improving the design and structure of the codebase. In the context of build code, refactoring can involve simplifying build scripts, removing duplication, and improving maintainability [24].

III. METHODOLOGY

In this section, we describe our research methodology. Figure 1 provides an overview of the process, which is composed of three primary phases: Data Preparation, Quantitative Analysis, and Implementation of BuildMiner.

A. Data Preparation

1) Data Collection: In this step, our goal is to have a gold set of commit changes in which the developers explicitly report the refactoring activity.

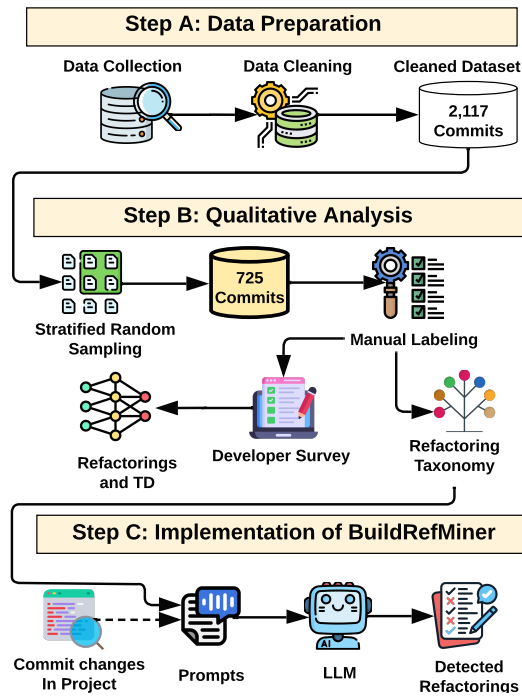[1]units of work or action in a build system (e.g., compiling, packaging).

Fig. 1: Approach Overview

To achieve this, we set out to retrieve commits that involve refactorings and are associated with Gradle, Maven, and Ant. We utilized BigQuery [25], an extensive collection of open-source GitHub projects, via the SQL query presented in Listing 2. We utilized a keyword-based mechanism to filter out all entries that do not contain the keyword 'refactor*' in the commit message. We use * to capture extensions like refactors, refactoring, etc. The keyword-based approach has been widely used in previous studies related to identifying refactoring changes [26]–[28], as it allows us to prune the search space to only consider code changes whose documentation matches a specific intention. The choice of 'refactor*', was specifically made to reduce the occurrence of false positives [27], [29], [30]. Furthermore, we mention variations of file names corresponding to Maven, Ant, and Gradle in our query. Overall, these steps helps ensure that the extracted refactorings are more likely to be build-related. Through this initial filtering, we collected a total of 5k commits.

```
1   SELECT * FROM
    "bigquery-public-data.github_repos.commits"
    WHERE (message LIKE '%refactor%') AND (message
    LIKE '%pom.xml%'
2   OR message LIKE '%build.xml%' OR message LIKE
    '%build.gradle%');
```

Listing 2: Extracting Refactoring-related Commits

2) Data Cleaning: In this step, we omit commits pertaining to repositories derived from other repositories, such as forked repositories, to prevent any redundancies and resulting potential biases in our study. In addition, we exclude projects that no longer exist, as they may be de-listed from GitHub but still exist in BigQuery. This left a total of 2453 commits, 691 correspond to Gradle, 1450 correspond to Maven and

312 correspond to Ant. Given the substantial size of the data and the considerable effort and time required for its manual examination, we applied a stratified random selection process to select a diverse and representative sample of build-related commits, following the recommendations of Levin et al. [31]. The stratification criteria was the build systems to which the commits correspond. For each build system, we selected a sample with a confidence level of 95% and a Margin of Error of 5%, which was determined to 248 are for Gradle, 173 for Ant, and 304 for Maven, for a total of 725 commits. These commits originated from 609 projects, which had an average 86K and median 9.7K lines of codes, an average 160 and median 1 forks, and finally an average 606 and median 5 stars.

These projects used a varied set of 30 programming languages. For example: 487 projects used Java, 51 used Kotlin, 19 used PHP, among other programming languages, as outlined in the appendix [1].

B. Qualitative Analysis methodology

1) Commit Analysis Methodology: Given that this study represents the initial investigation into the development of code-specific refactorings for build systems, it was imperative to conduct a manual examination of commits to identify the different refactorings as well as any relationship they may have with TD. The labeling was based on an open-coding process that allowed refactoring types to emerge from the data rather than preconceptions. We purposefully avoided enforcing existing refactoring operations since some of them may be limited to other artifacts like source code, while others may be specific to build code. We also followed the recommendations outlined by Usman et al. concerning the construction of taxonomies [32].

First, during the planning phase, three co-authors agreed on the area of focus being build refactoring. The main aim as the identification of the different types of refactoring that developers use in build scripts and to ascertain whether these were utilized to address technical debt. Finally, the classification framework is represented as a tree. Although a build refactoring taxonomy had been proposed in the literature by Simpson et al. [24], it was not utilized in this process to avoid bias, as itlackes empirical validation, and was technology-specific to an old version of Ant.

Second, the identification and extraction phase was performed over two rounds. For the first round (Identification), two of the co-authors with refactoring and build scripts experience separately observed all the build commits diffs in order to identify any refactoring changes and explicit mentions of TD. Their primary criterion was the definition of refactoring as "restructuring existing code to improve its quality without altering its external behavior". If a commit changes behavior, they don't consider it a refactoring. To systematically identify the observed refactorings, they used the build systems' official documentation, refactoring principles, commit names, and messages.

The two co-authors kept a shared record of high-level descriptions of the refactoring changes serving as a dynamic

reference, containing for each refactoring candidate a textual definition, a candidate-name, and an illustrative labeler-written code stub (to reduce bias/information leakage between the labelers in order to avoid any redundant definitions).

For the second round (categorization), the third co-author with previous build scripts experience was asked to rejoin this process. This was to minimize the bias, as this author did not participate in the identification of refactoring changes in the previous round. These three co-authors then categorized refactoring operations identified in the first round and mentions of TD (a refactoring commit is only linked to a TD category if it is explicitly mentioned in commit messages or developer feedback). These rounds were performed over one month to avoid labeling fatigue.

After the two labeling rounds concluded, Fleiss' Kappa score reflecting the agreement on both the identification of refactoring commits and their classification into specific types, was calculated at 0.76 signaling high agreement [33]. Then, three rounds of consensus meetings were carried out to resolve any disagreements.

Third, for the design and construction phase, a card-sorting process [34], [35] was performed by the three authors. This process grouped refactoring changes of similar characteristics into 24 categories. Subsequently, they refined this classification by determining 6 high-level main categories, for an easier generalizability and usability, giving it the format of a tree. Furthermore, we provide specific examples along with the definitions of the categories in order to avoid definition bias. This taxonomy is presented and detailed in Subsection IV-A.

Finally, for the Validation phase, a fourth co-author who possesses extensive expertise in the field of build systems confirmed the applicability of the various categories, ensuring they reflect significant recurring patterns and constitute valuable knowledge for developers.

Regarding the relationship between refactorings and TD, Peruma et al. [36] discuss how refactoring does not always address TDs. Indeed, 89% of refactoring commits they analyzed did not remove TD. Therefore, the authors found that this process alone was insufficient to establish a link between build refactorings and technical debt. We address this in subsubsection III-B2.

2) Developer Opinion Collection: Concerning the uncovering of links between the identified refactoring categories and TDs, a main challenge was the lack of documentation, along with ambiguous descriptions, which made it difficult to identify the prevalence of TD in the context of build refactoring. Hence, we opted for a conservative approach, where we only considered TD as being addressed by the refactorings if it was explicitly mentioned in the commit message or code comments. However, this approach may have led to the underestimation of the actual number of instances of TD repayment via build refactoring. To compensate for this shortcoming, and in line with previous studies [27], [37]–[40], we performed a cold-calling-based survey. We emailed and send direct-messages to the commit authors to enquire about the different refactoring instances and whether they were

linked to a TD. This survey was open-ended, and only took the form of one question: The motivation behind the applied refactoring changes. This was done to minimize the burden on the developers. This survey was conducted over a period of one month.

In total, we identified 85 commit messages/descriptions that clearly describe the motivation behind applying the refactoring changes. Furthermore, we received survey responses from 60 developers, out of 250 originally contacted, achieving a 24% response rate, in line with the average response rate of 15-30%, for software engineering surveys [41]–[43]. The survey responses provided us with additional insights into the refactoring changes and their relationship with technical-debt. For example, commit [44] contains an *Extract Module* refactoring , which was accompanied with the ambiguous commit message '*build.gradle refactoring*'. The developer clarified that this refactoring was used to address the modularity, and provided us with the following explanation: *build.gradle was too long (500 lines) and divided the functions...for better modularity*—thus clarifying the link between the different refactorings in this commit and TD. The results of this process and this survey are detailed in Subsection IV-B.

C. Implementation of BuildRefMiner

As mentioned in Subsection III-B, the difficulty and time-consumption of manually identifying instances of build refactoring highlights the need for automated, build-specific refactoring discovery. Hence, we design BuildRefMiner, a tool that can automatically analyze commits that affect build files to extract any refactoring operations that may have occurred. Currently, it utilizes GPT-4o, but it can be configured to use other LLMs. It utilizes LLMs due to their adaptability and proclivity with source-code analysis [45]–[48]. We utilized Prompt-Engineering practices [49], [50] while building this tool, specifically Zero-shot and One-Shot prompting.

The prompt, a snippet of which is shown in Figure 2, is composed of a system instruction, a list of the names and definitions of the build refactorings we discovered, inserted at the location of the text highlighted in green. As part of the One-Shot variant of the prompt, a code snippet example demonstrating each refactoring type is given along with each definition. Then, we introduce the new commit changes for analysis under the commit variable highlighted in yellow. This commit placeholder is composed of the full diff of that commit, including the name and path of the build file(s) impacted (e.g., src/pom.xml), Modified Lines: additions and deletions as commonly seen in Version-Control-Systems (e.g., git). We developed three distinct one-shot learning prompts, each tailored with examples specific to the build systems Gradle, Maven, and Ant for each refactoring type, where the build system and delimiter variables are changed depending on the build system being analyzed.

We evaluated the performance of both prompting approaches, and we detail the results of this evaluation in Subsection IV-C. While it is possible to use static analysis techniques to implement this tool, the reliance on LLMs has allowed us to quickly utilize, evaluate, and improve BuildRefMiner.

Fig. 2: BuildRefMiner Prompt

Furthermore, this implementation is more flexible and easier to extend for other build systems, unlike a static-analysis tool that would need to be customized to support every specific build tool. We provide BuildRefMiner as an accompaniment to our taxonomy and as a proof of concept concerning the relevance and importance of build refactorings and to facilitate future work. The implementation of BuildRefMiner with the usage of static analysis and its comparison with the LLM implementation can be the subject of an interesting future work.

IV. STUDY RESULTS

A. RQ1:Which refactoring types are developers applying to build code?

To uncover and categorize the Build refactoring types that are present within build scripts, we followed the methodology discussed in Subsection III-B and carried out a manual analysis of 725 commits. We discovered that 32% of them were false positives, as they did not contain any discernible refactorings. The primary cause of the inclusion of these false positives was the use which sometimes involved the addition or modification of existing functionality [51], [52]. This left a total of 403 true-positive Build-refactoring-related commits.

Figure 3 represents the taxonomy that we have generated based on a parent-child hierarchy. These refactoring types have been organized into 6 main categories for ease of classification and analysis. The tuple under each refactoring type indicates the number of refactoring changes we discovered from Gradle, Ant, and Maven respectively. Each percentage value represents the proportion of refactorings within a specific category out of the total of identified refactorings of the same main category. The classification into 6 main categories was based on the scope of their impact on the build code. These ranged from broad, project-level impacts ('Code Clean Up') to more specific levels, including Module ('Module Hierarchy Organization'), Method/Task ('Subroutine Organization'), Dependencies ('Dependency Organization'), Shared Properties ('Synchronizing Shared Build Properties'), and Local Variables ('Variables Organization'). The different

subcategories offer greater specificity as they represent the different kinds of refactoring identified. In total, we were able to develop a taxonomy of 24 build refactoring types. In the rest of this section, we discuss each of the Build-related Refactoring categories as well as their sub-categories by giving detailed definitions. In addition, we provide code examples to mitigate the potential bias inherent in definitions. It is notable that these definitions may be extended for application in other code artifacts.
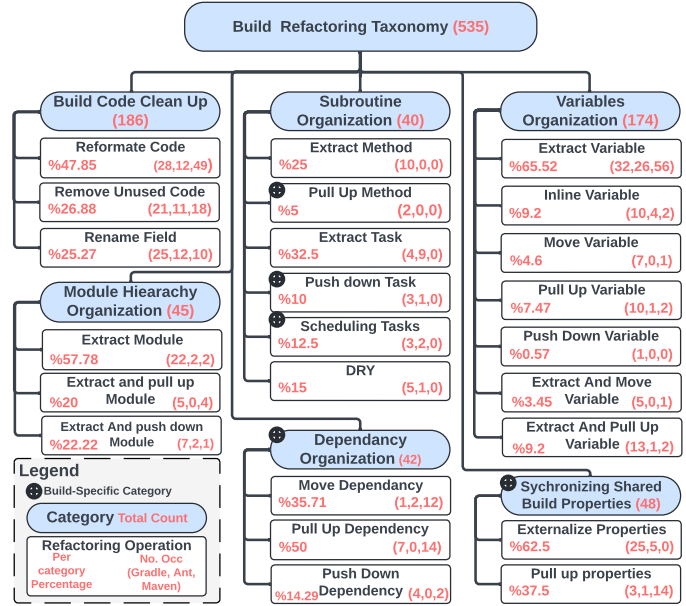

Fig. 3: Build-related refactorings taxonomy

*1) **Build Code Clean Up**:* This particular category encompasses alterations that serve to improve the cleanliness, readability, and comprehensibility of the build script. The detected refactoring operations represented the most prevalent refactorings for the three build systems, accounting for 34.77% of all refactorings. Three subcategories were observed:

1.1 Reformat Code (RC) This refactoring makes up 47.8% (89/186) of the Build Code Clean Up category. It involves aligning code with specific style conventions, adjusting indentation, reorganizing code blocks, and ensuring consistent coding style for clarity. Listing 3 shows an example in a Gradle script, switching to the plugins DSL for readability and modern best practices.

1.2 Remove Unused Code (**RUC**) This removes redundant or unused build code, accounting for 26.88% (50/186) of the Build Code Clean Up category.

1.3 Rename Field (RF): This refactoring, 25.27% (47/186) of the total, involves renaming build components (e.g., files, tasks, methods) to improve code clarity.

```
1  plugins {
2      id 'jacoco'}
3      id 'groovy'}
4  apply plugin: 'jacoco'
5  apply plugin: 'groovy'
```

Listing 3: Commit *f833d19* in *android-gradle-jenkins-plugin* Project: An example of Reformat Code Refactoring Type in Gradle.

2) Module Hierarchy Organization: This particular category pertains to the process of dividing the build script into separate modules (build files), with each build file being assigned a specific role or functionality. These modules are then arranged hierarchically, taking into consideration their dependencies and relationships. The refactoring operations that were identified collectively accounted for 8.41% of the total refactorings. The study identified three subcategories:

2.1 Extract Module (EM) represents 57.78% (26/45) of the refactorings in this category, which involves extracting and relocating functionality or responsibilities into a new build file within the same hierarchy. Notably, this refactoring type is more common in Gradle (22 occurrences) than in Maven and Ant (2 occurrences each), making it one of the most prevalent refactorings found in Gradle. Commit [53] illustrates an example of Extract Module. In this commit, developers extracted a segment of the parent 'build.gradle' script detailing multiple functionalities and relocated it to a new build file within the same hierarchy, 'publish.gradle'. The initial build file utilizes apply from 'publish.gradle' to invoke the extracted responsibilities.

2.2 Extract And Pull Up Module (EPUM) represents 20% (9/45) of the refactorings, it entails moving functionality or responsibilities from a more specialized module or class (child) to a newly created more generalized or parent module. Commit [54] illustrates an example of Extract and Pull Up Module in Maven. This change extracts a segment of a build code that delineates Maven properties from "lib/pom.xml" and relocate them to a superior newly created Maven file "pom.xml".

2.3 Extract And Push Down Module (EPDM) represents 22.22% (10/45) of the refactorings, it shifts functionality or responsibilities from a general or parent module to a more specific one newly created (child).

3) Subroutine Organization: This category covers operations modifying subroutines, specifically tasks or methods, primarily in Gradle and, to a lesser extent, Ant. It is noteworthy that Maven lacks comparable concepts. In Gradle, tasks represent actions, and methods configure them. Ant has predefined XML tasks but no methods. Therefore, refactoring related to methods applies solely to Gradle, while task-related refactoring includes both Ant and Gradle but excludes Maven. This category accounts for 7.48% of all identified refactorings. In total, we discovered two refactorings that focus on methods, and four refactoring types that are focused on tasks.

3.1 Extract Method (EM) This type represents 25% (10/40) of the Subroutine Organization category. It entails the identification of a cohesive code fragment and its subsequent relocation into a newly created method or function. Subsequently, the original code is substituted with an invocation of the recently generated method. Commit [55] provides an example of the Extract Method refactoring applied to a Gradle script. In this instance, the methods *createChecksum()* and *createExecutable()* were extracted from the *makeExecutables* task and invoked within the same task (Lines 34-41), improving code extensibility and reusability.

3.2 Pull Up Method (PUM) accounts for 5% (2/40) of the total refactorings in this category. It involves moving an existing method to the relevant Parent build file. An example of this refactoring is illustrated in commit [56] where *uploadArchives* method has been pulled up from a subfile *concourse-cli/build.gradle* to the parent file *build.gradle*.

3.3 Extract Task (ET) accounts for 32.5% (13/40) of the total refactorings in this category. It involves identifying a cohesive code fragment, relocating it into a new task, and replacing the original code with a call to the newly created task. Listing 4 is an example of Extract Task refactoring in Gradle. It defines a new task, resolveDependencies, which is set as a prerequisite for the publish task. Commit [57] serves as an illustration of the Extract Task refactoring type within the Ant build system. In this instance, eleven defined tasks (such as *start*, *start-secure*, and *start-batch*) exhibited identical behaviors, leading to code duplication. To eliminate this redundancy, a new task, *exec-scipio-jar*, was introduced (line 1033) to consolidate and encapsulate the shared behavior of these tasks.

```
1  publish.dependsOn {
2  task resolveDependencies{doFirst {
3      project.publishing {....}}}
4  publish.dependsOn(resolveDependencies)
```

Listing 4: Commit *4f03800* in *droidmate* Project: An example of Extract Task Refactoring Type in Gradle.

3.4 Push Down Task (PDT) represents 10% (4/40) of the refactorings. It entails moving a task defined in the root project's build file to the build file of one or more sub-projects.

3.5 Scheduling Tasks (ST): This refactoring type represents 12.5% (5/40) of the total refactorings in this category. It ensures a specific order of execution of tasks, or the execution multiple tasks simultaneously. In Ant, sorting tasks can be done using *depends* attribute that specifies the order of execution by indicating which tasks need to be completed before the current task begins. Listing 5 shows an example of a Scheduling Tasks refactoring in Ant. The changes were in place to ensure that *init* task was executed only after the *clean* task (Line 2).

3.6 Don't Repeat Yourself (DRY):

This refactoring type is related to the process of consolidating the common behavior of multiple tasks into a single location method to void redundancy. Listing 1 shows an example of the *DRY* refactoring type in Gradle. Initially, tasks such as Connect and Insert were explicitly configured, each possessing its own set of parameters such as environment

variables (Line 3-4). In the revised iteration, the identical configurations are dynamically applied to tasks by iterating through a list that includes the task names Connect and Insert (Line 1). As a consequence, the script becomes more succinct by eliminating repetitive configurations and replacing them with a generalized loop.

```
1  <target name="init"
2  <target name="init" depends="clean">
3      <mkdir dir="${bin}"/>.....
4  </target>
```

Listing 5: Commit *717417f* in *aDTN-platform* Project: An example of Scheduling Tasks Refactoring Type in Ant.

4) Dependency Organization: This particular category is a build-specific category, concerned with the reorganization of dependencies between build files, accounting for 7.85% of the overall count of identified refactoring changes. It plays a crucial role in determining the order and manner in which build dependencies are built and integrated. The study conducted identified three distinct types:

4.1 Move Dependency (MD) It refers to the transfer of dependency between two build files within the same hierarchy. It accounts for 35.71% (15/42) of the overall count of identified refactoring changes in this category, with 12 occurrences for Maven, compared to 1 and 2 occurrences each for Gradle and Ant respectively.

4.2 Pull Up Dependency (PUD) It involves moving a dependency from a sub-build file to its parent Build file; It accounts for 50% (21/42) of the overall count of identified refactoring changes in this category. Notably, this refactoring is more frequent in Maven, with 14 occurrences, compared to 7 and 0 occurrences each for Gradle and Ant respectively. Commit [58] exemplifies this type of refactoring by consolidating various dependencies, including the Spring Boot Gradle plugin and the Kotlin Gradle plugin, that were previously declared across multiple sub-build files. These dependencies were eliminated from the individual sub-build files and migrated to the parent build file *build.gradle*, specifically within lines 1-38, to enhance maintainability and reduce redundancy.

4.3 Push Down Dependency (PDD) It accounts for 14.29% (6/42) of the overall count of identified refactoring changes in this category. It entails moving a dependency from a parent file to a child or specific build file.

5) Synchronizing Shared Build Properties: This category is another discovered Build-specific refactoring category and represents 8.97% of the overall count of the identified refactoring types. It refers to the process of ensuring that multiple parts or modules of a build system use a consistent set of properties. These properties can be configurations, versions, paths, or any piece of data that affects how the build process operates. When projects grow in complexity and include multiple components or modules, it's essential to maintain a single source of truth for shared properties to avoid inconsistencies. Two distinct types were observed during the study:

5.1 Externalize Properties (EP): Accounts 37.5% (18/48).

This type of refactoring focuses on the centralized and harmonized handling of build configurations by extracting environment-specific configurations, credentials, and settings from the build script and application code. This is typically achieved by utilizing external properties or separate configuration files, such as *.properties* files. This type of refactoring is particularly common in Gradle, with 25 instances, compared to 5 and 0 instances for Ant and Maven, respectively. Listing 6 shows an example of Externalize Properties. The initial implementation involved hardcoding the version value directly into the script. The revised modifications improve version management efficiency by introducing an external *build.properties* file (Line 2) that encapsulates essential configurations. This approach decouples version management from the build script, allowing users to update configurations easily by modifying the properties file rather than directly altering the script itself. Decoupling the components of the build system not only improves the ease of reading, but also enhances the ability to maintain it. This allows developers to make changes to configurations without needing to delve into the fundamental build logic.

```
1  version = "1.7.10-0.1"
2  ext.configFile = file "build.properties"
3  configFile.withReader {def prop = new Properties()
4    project.ext.config = new ConfigSlurper().parse prop}
5  version = config.version
```

Listing 6: Commit *611c4b3* in *YetAnotherBackupMod* Project: An Example of Externalize Properties Refactoring in Gradle.

5.2 Pull Up Properties (PUP) Accounts 62.5% (30/48): pertains to the process of consolidating frequently used configurations and properties within a build file into a centralized location or method within the root build file. This refactoring type is most common in Gradle Maven with 25 occurrences compared to 3 and 1 occurrences for Gradle and Ant respectively. A specimen of this refactoring type is illustrated in Commit [59]. The changes involve migrating configuration elements from a child Maven build file *core/pom.xml* (Line 24-29) to the root *pom.xml* (Line 56-60). By centralizing the source control management metadata in the root file, the project benefits from improved coherence and ease of updates, as any changes to these properties need to be made in only one location.

6) Variables Organization: This category constituted the second largest refactoring category after the Code Clean Up Category, accounting for 32.52% of the total count of identified refactoring types. It is primarily associated with the organization of variables within a single build file or across the hierarchy of build files. All the identified subcategories of variable refactoring are more frequent in Gradle compared to Ant and Maven. They are as follows: **Extract Variable(EV)** accounting for (65.52%), **Inline Variable (IV)** (9.2%), **Move Variable (MV)** (4.6%), **Pull Up Variable (PUV)** (7.47%), **Push Down Variable (PDV)** (0.57%), **Extract And Move Variable (EMV)** (3.45%), and **Extract And Pull Up Variable**

(EPUV) (9.20%).

Out of these refactoring categories, some are that are analogous to other artifacts and adapted to the context of build systems (e.g Pull Up Properties) and some that emerge due to the peculiar nature of build artifacts (Build-specific) that do not have equivalents in other artifacts (e.g., source code). Among the 24 refactorings, we identified 8 as Build-specific types. These include the Tasks-related refactorings, under the *Subroutine Organization* main category (Scheduling Tasks, Push Down Task, and Extract Task), and all the refactoring belonging to the *Dependency Organization* and *Synchronizing Shared Build Properties* main categories. We observed that developers have to specifically adapt to the context of Build files when applying these refactoring changes. This distinction arises from the nature of their primary concerns, which revolve around the foundational elements of build systems: tasks, dependencies, and properties.

Comparing our Taxonomy with Simpson et al. [24], the categories we defined are much more technology agnostic, in comparison to some of their technology-specific categories, such as *Move element to Antlib* and *Replace Exec with Apply*. Furthermore, Simpsons' taxonomy does not include some categories we discovered, such as DRY and Code Cleanup. While some categories are indeed similar between the two taxonomies, such as Simpson's Introduce Property File and our Externalize Properties Categories, our taxonomy is organized into empirically validated 6 Main categories that are dependent on scope, and we use naming conventions that are more inline with Refactoring in other domains, which makes our taxonomy easier to understand and utilize.

> **Finding 1**
>
> We created a build refactoring taxonomy composed of 24 refactorings types, that fall under 6 main refactoring categories. 8 of the 24 refactorings are specific to build systems.

B. RQ2: How are build refactorings linked to technical debt?

Our analysis of 85 commit messages and 60 developer responses shows that 20 out of the 24 identified refactoring types are associated with technical debt (TD) reduction, accounting for 94.01% of the total refactoring instances. However, five refactoring types—Move Dependency, Extract and Move Variable, Move Variable, Push Down Task, and Push Down Variable—were not classified as related to TD repayment. The rationale behind the application of these five refactoring types remains unclear, suggesting the need for further investigation into their role in TD repayment.

1) Technical Debt Categories : We extracted 5 technical debt categories related to *Extensibility & Maintainability* , *Modularity*, *Clarity & Readability*, *Code Duplication & Redundancy*, and *Security*. We organized the different refactoring types based on their rationale, e.g., technical debts they address as shown in Table I. The percentage associated with each technical debt represents the cumulative number of refactor-

Fig. 4: Developer Response on Commit [61]

Fig. 5: Developer Response of Commit [63]

ings undertaken to address that particular TD out of the total number of refactorings identified.

Clarity & Readability is a TD that receives considerable attention during the process of refactoring build files by 39.63% of the total refactorings. It was linked with the following refactoring types: *Inline Variable*, *Rename Field*, *Reformat Code* and *Remove Unused Code*. These refactorings were used when code is not clear, and hard to read requiring huge efforts to understand it. For example, Rename field was used to avoid using irrelevant names for build artifacts and to ensure consistency in naming conventions, like the example commit message *"In order for this to become more comprehensive ... the package name needed to be changed"* from the commit [60].

Reformat Code and Inline Variable were used to ensure better readability and understandability as shown in the developer response in Figure 4 to commit [61].

Extensibility & Maintainability was among the most addressed TDs in build files, and was mainly associated with: *Extract Variable*, *Extract And Move Variable*, *Extract Method*, *Extract Task*, and *Move Task*, repaid by 30.66% of the total refactorings.

As software grows and evolves, adding new components or improving existing ones often requires modifying or duplicating elements in the build system, which can lead to challenges in both extensibility and maintainability. Refactoring techniques like Extract Variable, Extract Task, and Extract Method aim to increase the system's abstraction levels and reusability, making future modifications easier and reducing the risk of widespread bugs. This is illustrated by a example commit message *"Task 'retroweaver' moved to the prepare target so it can be reused in several places"* for the commit [62].

Also, externalizing properties ensures that changes can be made with minimal disruption, enhancing both the flexibility to adapt the system and the ease with which future changes and maintainability efforts as in the example of a developer response shown in Figure 5

Modularity This technical debt is addressed by 14.95% of the total refactorings, such as Extract Module, Extract and pull up Module, Extract and Push Down Module, Push Down Task, Pull Up Method, Pull Up Variable, Extract and Pull Up variable and Push Down Dependency. These refactoring techniques break down complex build files into smaller, modular units, ensuring that each component is situated at the most appropriate level of the build system hierarchy,

```
... I have decided that build.gradle was too long (500
lines) and divided the functions in it according to
responsibilities, for better modularity ...
```

Fig. 6: Developer Response for Commit [44]

```
Split dependencies in optional modules (at this time
gumetree was growing to much and we started to make it
more decoupled)
```

Fig. 7: Developer Response for Commit [64]

making management easier, clearer, and reducing errors during updates. This is illustrated by commit messages [44]: *"refactor build.xml for centralized usage for all plugins.",* and developers answers such as the ones shown in Figure 6 and Figure 7. Extract Module, Extract and pull up Module, Extract and Push Down Module, Push Down Task, Pull Up Method, Pull Up Variable, Extract and Pull Up variable and Push Down Dependency. These refactoring techniques break down complex build files into smaller, modular units, ensuring that each component is situated at the most appropriate level of the build system hierarchy, promoting simplicity, clarity, making it easier to manage, and less complex, and reducing the likelihood of errors when components are changed. This is illustrated by developers answers shown in Figure 6 and Figure 7, and the commit message *"refactor build.xml for centralized usage for all plugins."* for commit [44]

Code Duplication Duplicated code and redundant elements make a system more difficult to maintain, prone to errors, and less efficient. This is addressed by 4.20% of the total refactorings Refactoring techniques such as Pull Up Properties, Pull Up Method, Pull Up Dependency, Extract and Pull Up Variable, and Pull Up Variable focusing on consolidating shared code fragments across various classes into higher-level modules. By pulling up shared elements, these techniques eliminate duplication, making the code more manageable, and reducing the likelihood of inconsistencies. This is illustrated by a commit message *"Reduced duplicate POM XML content"* for commit [65], where the refactorings Pull Up Properties and Pull Up Dependency where applied. Also, an example of a developer response for commit [63] shown in Figure 8, where Pull Up Variable and Pull Up Dependency were applied.

Also, implementing the DRY principle is key in reducing code redundancy, and improving clarity and maintainability while reducing the risk of future bugs. An example of DRY change in commit [66] where the developer states in the commit message *"Refactor of productFlavors So now we aren't duplicating a bunch of buildConfigField values".*

Security: This technical debt was mainly addressed by

```
This was purely to dedup code, so gradle changes could
be made in a single place, instead of multiple files,
for each sub project.
```

Fig. 8: Developer Response of Commit [63]

Externalize Properties refactoring (1.12%) . Build scripts may expose sensitive information or have vulnerabilities that could be exploited. Refactoring techniques such as "Externalize Properties" help address these security risks by reducing potential breaches through improved access control and securing configurations, safeguarding the system from attacks. The goal of this refactoring was also described in the commit message *"Fixes security consideration issues"* for the commit [67], where sensitive build properties were externalized to a separate file that is then added to .gitignore.

Technical Debt	Percentage	Refactorings
Clarity & Readability	39.63%	RF, IV, RUC, RC
Extensibility & Maintainability	30.66%	EV, EP, EM, EV
Modularity	14.95%	EM, EPUM, PUV, EPDM, PDD
Code Duplication	14.20%	PUP, PUM, PUD, EPUV, PUV, DRY
Security	1.12%	EP

TABLE I: build refactoringTechnical Debt Categories

Finding 2

Among the 24 refactoring types identified, 20 are linked to reducing technical debt. We identify which of these refactoring types addresses which of the five specific types of technical debt.

C. RQ3: What is the effectiveness of our tool BuildRefMiner in identifying build refactorings?

After the manual labeling of refactoring commits undertaken in Subsection III-B, we created a gold set of labeled commits and corresponding refactorings within them. We use this labeled set to evaluate the performance of BuildRefMiner. We calculate the precision (PR), recall (RE), and F1-score (F1) of BuildRefMiner across the different refactoring types. The results of this evaluation are shown in Table II. It's important to note that the refactoring code examples used within the One Shot variant of the prompt were removed from this evaluation set when evaluating the One-Shot variant. This was done to avoid information leakage and ensure the accuracy of our evaluation.

Using Zero-Shot prompting, BuildRefMiner yielded mixed results across the 24 refactoring types, achieving an overall precision of 0.67, recall of 0.72, and F1-score of 0.66. Notably, two refactorings such as Extract and Move Variable and Push Down Variable achieved an F1-score of 0.95 and 1 respectively, indicating a strong alignment of BuildRefMiner with the empirical definitions in this category. In four refactoring types—Extract and Pull Up Module, Extract Module, Push Down Dependency, and Move Dependency—BuildRefMiner maintained a high degree of performance in its classification, with an F1-score from 0.7 to 0.84. However, a notable deviation was observed for certain types, including Push Down Module, Extract Method, Move Variable, and Pull Up Properties, which presented lower F-1 scores, between 0.25

and 0.57, suggesting that the Zero-Shot model struggles with nuances in detecting these refactorings without prior context.

A significant improvement in BuildRefMiner performance across several refactoring types can be seen when using One Shot Prompting. The overall precision, recall, and F1-score go up to 0.79, 0.78, and 0.75 respectively. Going over the performance of BuildRefMiner over the different refactoring types, BuildRefMiner demonstrated enhanced performance in detecting 20 out of the 24 refactorings overall. A notable example is the dramatic increase in F1-score from 0.25 to a 1 for the Extract and Push Down Module refactoring. This substantial improvement highlights the effectiveness of providing contextual examples, enabling the model to better interpret and accurately detect code changes without generating false positives or missing instances. Additionally, in two other refactoring categories—Extract Module and Extract and Move Variable—BuildRefMiner maintained high precision (1 and 0.96) alongside near-perfect recall, achieving 0.95 in both cases.

However, despite the tool achieving significant recall improvements for Rename Field, Remove Unused Code, Scheduling Tasks, Reformat Code, and Move Variable, ranging from 0.81 to 1, the precision for these categories was noticeably lower, falling between 0.55 and 0.72. This indicates that, although the tool successfully identified every true instance of these refactorings, it also misclassified some unrelated changes as refactorings, resulting in moderate precision. Thorough manual validation, it became apparent that some of these refactorings were often intertwined with other refactoring actions, which contributed to the false positives observed. BuildRefMiner tended to identify them as distinct refactorings even when they were embedded within broader refactoring activities, thus inflating the number of false positives. For example, Move Variable was frequently accompanied by actual refactorings such as Move Method, Move Task, or Extract Module. Remove Unused Code frequently overlaps with refactorings that involve moving code to another class or module (e.g Extract Module, Move Method etc..).

To demonstrate the usefulness of BuildRefMiner on projects in the wild, we share the result of running it on the gradle-modifying commit [68] from `Google/Nomulus`. This particular commit involved multiple modifications across different build files at various levels of the project hierarchy, making it hard to parse manually. BuildRefMiner categorized the changes as shown in Figure 9, of which we manually verified the veracity, thus giving a glimpse into the usefulness and time-efficiency of our tool for future research.

- Extract Module
  Details: Code changes in 'gradle/build.gradle' have been extracted to 'appengine_war.gradle', simplifying the main build file and centralizing common configurations.
- Extract and Pull Up Module
  Details: In 'gradle/buildSrc/build.gradle', configurations such as buildscript dependencies, plugins, and checkstyle configurations have been extracted and referenced using 'apply from: '../java_common.gradle''.

Fig. 9: BuildRefMiner output on Commit [68]

V. RELATED WORK

Recent research has highlighted the significance of build code maintenance as a software system evolves. Mcintosh et

Refactoring Types	Zero-Shot			One-Shot		
	PR	RE	F1	PR	RE	F1
Rename Field	0.54	0.71	0.61	0.72	0.82	0.76
Remove Unused Code	0.53	0.77	0.63	0.56	0.81	0.66
Scheduling Tasks	0.48	0.84	0.60	0.55	1.00	0.70
Reformat Code	0.53	0.80	0.64	0.55	0.83	0.66
Extract and Pull Up Module	1.00	0.75	0.81	1.00	0.78	0.88
Extract and Push Down Module	0.17	0.50	0.25	1.00	1.00	1.00
Extract Module	0.79	0.95	0.84	0.96	0.95	0.93
Extract Method	0.33	0.57	0.42	0.75	0.6	0.55
Pull Up Method	1.00	0.50	0.67	1.00	1.00	1.00
Extract Task	0.50	0.63	0.55	0.58	0.63	0.60
Push Down Task	0.75	0.67	0.70	0.75	0.67	0.7
DRY	0.75	0.60	0.67	0.88	0.80	0.84
Push Down Dependency	0.84	0.84	0.84	0.84	0.84	0.84
Pull Up Dependency	0.82	0.53	0.64	0.95	0.53	0.67
Move Dependency	0.78	0.81	0.70	0.78	0.81	0.70
Pull Up Properties	0.52	0.63	0.57	0.65	0.67	0.66
Externalize Properties	0.73	0.57	0.64	0.81	0.65	0.72
Extract Variable	0.74	0.74	0.74	0.77	0.78	0.77
Inline Variable	0.60	0.55	0.51	0.83	0.58	0.61
Move Variable	0.50	0.50	0.50	0.67	1.00	0.80
Pull Up Variable	0.78	0.47	0.58	0.62	0.53	0.57
Push Down Variable	1.00	1.00	1.00	1.00	1.00	1.00
Extract And Move Variable	1.00	0.90	0.95	1.00	0.95	0.96
Extract And Pull Up Variable	0.50	0.62	0.49	0.83	0.68	0.68
All Refs.	**0.67**	**0.72**	**0.66**	**0.79**	**0.78**	**0.76**

TABLE II: BuildRefMiner Performance across the refactoring types with Zero-Shot and One-Shot Prompting.

Finding 3

We tested two BuildRefMiner variants, finding that the One-Shot prompt variant achieved Precision, Recall, and F-1 scores of 0.79, 0.78, and 0.76, respectively, outperforming the Zero-Shot variant, which scored 0.67, 0.72, and 0.66.

al. [69] have investigated the co-change of source and test code with build files. However, they did not investigate the specific types of build changes, unlike our sharp focus on build refactorings. Macho et al. [70] have extracted detailed build changes from Maven build files. However, all of the change types studied are CRUD (create, remove, update, delete) changes to meet new build requirements. Hence, they are not refactorings as they modify the system;s behavior. In their study, Hardt et al. [71] have introduced Formiga for Ant, which aims to facilitate build maintenance and dependency discovery. It only includes a limited set of rename and code cleanup refactorings. Simpson et al. [24], have discussed the need for Ant build files refactoring and lists 23 such refactorings. However, both these works is focused only on Ant build systems and the taxonomy/categories provided is based on the Author's experience and subjective opinions contrasting with our research which incorporates a broad empirically-grounded quantitative and qualitative analyses that also includes the more modern Maven and Gradle build systems. Shridhar et al. [72] conducted a qualitative analysis of the build commit history of 18 open-source projects from the Maven and Ant systems. Their study highlights that changes are predominantly functional, and while "Preventive" and "Perfective" improvements are mentioned, they are not a focus of this study and the different refactorings that may constitute these changes are

not detailed. In their research, Xiao et al. [73] analyzed 500 commits across 291 projects' Maven build systems, identifying technical debt (TD) primarily arising from tool constraints, library limitations, and dependency management challenges. They also notified developers of TD presence and, in some cases, submitted pull requests to address rudimentary TDs related to dependency incompatibilities. However, they did not investigate or propose exhaustive and generic refactoring approaches, nor did they link specific fix and TD categories. Our study proposes a deeper investigation into build system refactorings and how they relate to technical debt. It builds on previous research by empirically examining various build refactoring changes made by practitioners. We aim to make their findings more accessible and useful by offering a tool, BuildRefMiner, which automates the detection of build refactorings to support ongoing research in this field.

VI. THREATS TO VALIDITY

Internal Validity: These threats may stem from errors in the categorization of refactoring changes in build code, which could introduce bias due to limited supporting documentation. To reduce this risk, a panel of three experts independently evaluated selected commits, where they documented refactorings and justifications with caution, only when highly confident. Results showed strong agreement across identification and classification.

External Validity: Random sampling may have excluded some large commits, potentially missing certain refactorings. However, we applied a stratification strategy to obtain a 95%-confidence representative sample, and the commits selected span a large variety of systems, giving credence to our results.

For BuildRefMiner evaluation, as we adopted a stratified sampling, some sub-categories have small class sizes. Therefore, the evaluation metrics may not be representative. Increasing the sample size of each sub-category is ideal but would require more manual labeling. We remedy this with a more broad evaluation of the overall performance of BuildRefMiner across all refactoring types. A more exhaustive evaluation can be part of future work.

Construct Validity:

A potential risk is the possibility that commits categorized as refactorings may actually break the build, thereby contradicting the intent of refactoring as behavior-preserving changes. To mitigate this threat, we employed a multi-round, multi-labeler process where all labelers confirmed no behavioral changes occurred in the identified refactorings, ensuring accurate classification.

VII. STUDY IMPLICATIONS

For Researchers: We help familiarize researchers with build refactoring, by providing them with an empirically-grounded and definition-supported Taxonomy composed of 24 Build refactorings, and we highlight the 5 TDs some of these refactorings address. Furthermore, we believe that the 2117 Build Refactoring commits we collected, BuildRefMiner which we developed, and the findings we reach can provide

a groundwork to help guide future research into the field of Build refactoring.

For Practitioners: Due to the lack of empirically-grounded existing taxonomies on Build refactoring, practitioners are unaware or misinformed about this practice. We believe the taxonomy we provided, along with the descriptions of the different TDs addressed by the refactorings within this taxonomy, can help guide practitioners in performing Build refactoring procedures. BuildRefMiner is intended for researchers, primarily for automatic large-scale build refactorings mining. We hope to make it more useful for developers in future work via automatic TD identification and remediation.

VIII. CONCLUSION

In this study, we performed an empirical analysis on 725 commits that were related to build code refactoring from 609 open-source projects. Through manual analysis, we classified the 24 build-related refactorings we discovered into 6 main categories, and we distinguish 8 of which as build-specific. We also linked some of them to 5 TD categories with the help of developer feedback. Finally, we developed BuildRefMiner to help guide future research into this field. To the best of our knowledge, this is the first empirical study of refactorings and technical debt in Build Systems.

IX. ACKNOWLEDGMENT

REFERENCES

[1] "Refactoring in build systems - replication package," https://sites.google.com/view/buildref.

[2] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 38–47.

[3] S. McIntosh, B. Adams, and A. E. Hassan, "The Evolution of ANT Build Systems," in *Proc. of the Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 42–51.

[4] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An Empirical Study of Build Maintenance Effort," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2011, pp. 141–150.

[5] B. Varanasi, *Introducing Maven: A Build Tool for Today's Java Developers*. Apress, 2019.

[6] B. Matzke, *Ant: The Java Build Tool in Practice*. Charles River Media, Inc., 2003.

[7] A. L. Davis and A. L. Davis, "Gradle," *Learning Groovy 3: Java-Based Dynamic Scripting*, pp. 105–114, 2019.

[8] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, vol. 17, pp. 578–608, 2012.

[9] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 41–50.

[10] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 724–734.

[11] G. Kumfert and T. Epperly, "Software in the doe: The hidden overhead of"the build"," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2002.

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

[13] M. Nejati, M. Alfadel, and S. McIntosh, "Code review of build system specifications: Prevalence, purposes, patterns, and perceptions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1213–1224.

[14] J. Ivers, A. Ghammam, K. Gaaloul, I. Ozkaya, M. Kessentini, and W. Aljedaani, "Mind the gap: The disconnect between refactoring criteria used in industry and refactoring recommendation tools," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2024, pp. 138–150.

[15] W. Aljedaani, A. Ghammam, M. W. Mkaouer, and M. Kessentini, "From boring to boarding: Transforming refactoring education with game-based learning," in *Proceedings of the ACM/IEEE 8th International Workshop on Games and Software Engineering*, 2024, pp. 20–27.

[16] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," 2016.

[17] B. Muschko, *Gradle in Action*. Simon and Schuster, Feb 2014.

[18] G. B. T. 2023, "https://gradle.org/," online; accessed on 22 September 2023.

[19] C. MacNeill, "Apache ant - welcome." [Online]. Available: https://ant.apache.org/

[20] B. Porter, "Maven – welcome to apache maven." [Online]. Available: https://maven.apache.org

[21] A. Ghammam, T. Ferreira, W. Aljedaani, M. Kessentini, and A. Husain, "Dynamic software containers workload balancing via many-objective search," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2575–2591, 2023.

[22] A. Ghammam, R. Khalsi, M. Kessentini, and F. Hassan, "Efficient management of containers for software defined vehicles," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–36, 2024.

[23] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 5, p. 51–54, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2507288.2507326

[24] S. Duncan, "The thoughtworks® anthology, essays on software technology and innovation," *Software Quality Professional*, vol. 11, no. 2, p. 53, 2009.

[25] S. Fernandes and J. Bernardino, "What is bigquery?" in *Proceedings of the 19th International Database Engineering & Applications Symposium*, 2015, pp. 202–203.

[26] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 348–357.

[27] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, and A. Ouni, "Behind the scenes: On the relationship between developer experience and refactoring," *Journal of Software: Evolution and Process*, vol. 36, no. 1, p. e2395, 2024.

[28] ——, "An exploratory study on refactoring documentation in issues handling," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 107–111.

[29] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, pp. 1192–1223, 2016.

[30] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Systems with Applications*, vol. 167, p. 114176, 2021.

[31] S. Levin and A. Yehudai, "Towards software analytics: Modeling maintenance activities," *arXiv preprint arXiv:1903.04909*, 2019.

[32] M. Usman, R. Britto, J. Börstler, and E. Mendes, "Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method," *Information and Software Technology*, vol. 85, pp. 43–59, 2017.

[33] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews," *Sociological Methods & Research*, vol. 42, no. 3, p. 294–320, Aug. 2013. [Online]. Available: https://doi.org/10.1177/0049124113500475

[34] S. Fincher and J. Tenenberg, "Making sense of card sorting data," *Expert Systems*, vol. 22, no. 3, pp. 89–93, 2005.

[35] D. Spencer and T. Warfel, "Card sorting: a definitive guide," *Boxes and arrows*, vol. 2, no. 2004, pp. 1–23, 2004.

[36] A. Peruma, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "Refactoring debt: myth or reality? an exploratory study on the relationship between technical debt and refactoring," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 127–131.

[37] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE, 2021, pp. 238–250.

[38] S. Baltes and S. Diehl, "Worse than spam: Issues in sampling software developers," in *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, 2016, pp. 1–6.

[39] A. Nauman Ghazi, K. Petersen, S. S. V. R. Reddy, and H. Nekkanti, "Survey research in software engineering: Problems and strategies," *arXiv e-prints*, pp. arXiv–1704, 2017.

[40] F. J. Fowler Jr, *Survey research methods*. Sage publications, 2013.

[41] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE software*, vol. 12, no. 4, pp. 52–62, 1995.

[42] C. Pacheco and I. Garcia, "Stakeholder identification methods in software requirements: empirical findings derived from a systematic review," in *2008 The Third International Conference on Software Engineering Advances*. IEEE, 2008, pp. 472–477.

[43] M. Ciolkowski, O. Laitenberger, S. Vegas, and S. Biffl, *Practical experiences in the design and conduct of surveys in empirical software engineering*. Springer, 2003.

[44] "build.gradle refactoring · ati-ozgur/kdd99reproducibleexperiments@fd166ab." [Online]. Available: https://github.com/ati-ozgur/KDD99ReproducibleExperiments/commit/fd166dab91663cc8395cb8b6edd5f6c928bd4f7e

[45] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[46] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using llms and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.

[47] Z. Li, C. Wang, P. Ma, C. Liu, S. Wang, D. Wu, C. Gao, and Y. Liu, "On extracting specialized code abilities from large language models: A feasibility study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[48] R. Patil and V. Gudivada, "A review of current trends, techniques, and challenges in large language models (llms)," *Applied Sciences*, vol. 14, no. 5, p. 2074, 2024.

[49] L. Giray, "Prompt engineering with chatgpt: a guide for academic writers," *Annals of biomedical engineering*, vol. 51, no. 12, pp. 2629–2633, 2023.

[50] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[51] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[52] J. Bauer, "Make the featurefactory an option get the parser's objects from the... stanfordnlp/corenlp@ 616d524," *Stanford NLP*, 2014.

[53] "Gradle refactoring. · zoltu/stubkafkabroker@489dac4." [Online]. Available: https://github.com/Zoltu/StubKafkaBroker/commit/489dac4d35d33936d588507f8879dbc703c09e7a

[54] "Refactored build.gradle · pixplicity/gene-rate@1800eae." [Online]. Available: https://github.com/thrasher/f8api/commit/fd38cf6

[55] "refactored the maven pom.xml layout to include modules for parent · thrasher/f8api." [Online]. Available: https://github.com/HMCL-dev/HMCL/commit/e38bfdfc736f07b7e803341a43a8d618085c6786

[56] "Gradle build refactoring · hcuffy/concourse@13555a8." [Online]. Available: https://github.com/hcuffy/concourse/commit/13555a80ade66111c603b0aabe21c2e0a2bc6c86

[57] "Refs 1514 build.xml." [Online]. Available: https://github.com/ilscipio/scipio-erp/commit/015abe743d333a865ba0927abb268974275c83bc

[58] "Refactoring gradle kotlin configuration to root multiproject build.gr... · atrujillofalcon/spring-guides-kotlin@8732237." [On-

line]. Available: https://github.com/atrujillofalcon/spring-guides-kotlin/commit/87322375ba294504879e53668b488f16d451184a

[59] "changed pom.xml – refactoring · treasure-data/td-logger-java@4e9590c." [Online]. Available: https://github.com/treasure-data/td-logger-java/commit/4e9590c3dd3719ee12a5b5042a4c426efbc77474

[60] "Refactor sample module · farbodsz/usefulviews@70e7b00." [Online]. Available: https://github.com/farbodsz/UsefulViews/commit/70e7b00ffebe65c3eab7c280bc854255ab74260e

[61] "mockito version bump · koral–/android-gradle-jenkins-plugin@f833d19." [Online]. Available: https://github.com/koral--/android-gradle-jenkins-plugin/commit/f833d19b8a21c103e0665848bdccc936ab5db159

[62] "Refactorized build.xml." [Online]. Available: https://github.com/jimmycd/liquibase/commit/8fdc2e10410e84311241cfc80525b54deb13d7a7

[63] "Gradle refactor issue49 (51) · airbnb/epoxy@49cd795." [Online]. Available: https://github.com/airbnb/epoxy/commit/49cd79581fe93c2e620cb90e68c72779ed8d5706

[64] "Refactored and upgraded dependencies · caiusb/gumtree@bf85ae7." [Online]. Available: https://github.com/caiusb/gumtree/commit/bf85ae70f602aa561cf8e7ab6b2314ba7377e61e

[65] "Refactored examples to use a parent pom. reduced duplicate pom xml co... · matthewmccullough/encryption-jvm-bootcamp@11840e3." [Online]. Available: https://github.com/matthewmccullough/encryption-jvm-bootcamp/commit/11840e3d5bdbb5cac2a696f2ce309d87b12586fa

[66] "build.gradle." [Online]. Available: https://github.com/iFixit/iFixitAndroid/commit/3b4c4f5a5147304d8b16a11ee4c90bcc784a0894

[67] "refactor." [Online]. Available: https://github.com/jatindhankhar/shorl/commit/0bd7cd766513926085c9eac3ed865bbe936f9ace

[68] "Refactor gradle project setup · google/nomulus." [Online]. Available: https://github.com/google/nomulus/commit/d1b7824

[69] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 241–250.

[70] C. Macho, S. McIntosh, and M. Pinzger, "Extracting build changes with builddiff," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 368–378.

[71] R. Hardt and E. V. Munson, "Ant build maintenance with formiga," in *2013 1st International Workshop on Release Engineering (RELENG)*. IEEE, 2013, pp. 13–16.

[72] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, 2014, pp. 1–10.

[73] T. Xiao, D. Wang, S. McIntosh, H. Hata, R. G. Kula, T. Ishio, and K. Matsumoto, "Characterizing and mitigating self-admitted technical debt in build systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4214–4228, 2021.