Analyzing Similar Build Configurations Across Different GitHub Projects

Calin Manoli

Supervisor(s): Sebastian Proksch, Shujun Huang

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 28, 2024

Name of the student: Calin Manoli
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang, Julia Olkhovskaia

Abstract

GitHub is the home of hundreds of millions of Open Source Software(OSS) repositories where users collaborate on projects and find inspiration for new ideas. Some of these projects have certain build configurations set up to make building, testing, and deploying the software more time-efficient and less error-prone. However, setting up the correct configurations usually requires a lot of time and a high level of knowledge. This paper aims to analyze the current practices for setting up build configurations like the Maven files and GitHub actions while clustering some of these practices based on the scope of the project. Thus, we provide useful information in terms of discovering similar projects based on the build configurations and discuss the feasibility of build configuration analysis. In summary, we provide a comprehensive analysis of project similarity based on Maven build configurations and workflow files, shedding light on the importance of build configurations for identifying similar projects, and laying the groundwork for future exploration in the realm of build configuration analysis.

1 Introduction

GitHub hosts hundreds of millions of Open Source Projects (OSS) and facilitates millions of users to collaborate on creating software projects. To effectively manage these projects, developers often employ automated workflows that streamline repetitive tasks, minimize potential failure points, and conserve valuable time. There exists a plethora of tools that can be used for this purpose, such as the Maven and Gradle build tools, GitHub Workflow pipelines, Docker, etc. For the majority of these tools, developers are required to know the best practices of how to properly configure and use them beforehand. In addition, the best practices can change depending on the scope of the project. While initiatives like the GitHub Actions Marketplace aim to simplify the setup process for the GitHub workflows, developers still require a comprehensive understanding of the most suitable pipelines for their project's scope and the most appropriate tools for their specific use cases. For example, imagine this scenario: you are a developer working on a project, and as the complexity of the project increases, the time spent on doing repetitive tasks also increases. Tasks can be linting code, running tests to ensure everything works as intended, building and deploying the project, etc. By discovering projects with similar configurations, developers could potentially get recommendations so that they get inspiration and examples of possible pipelines and build plugins they can use to ease and automate their jobs.

This research delves into the feasibility of identifying similar OSS repositories based solely on the Maven files and their workflows. Maven and Gradle are very similar tools in terms of their purpose, in the sense that they are both tools used for automating the build process of Java-based applications. Although similar, recent reports are showing that Maven is generally the preferred option of Java developers, with more than 76% of them using it [13]. Hence, we decided to focus our analysis on Maven. Workflow configurations have also been selected for analysis since they have the same structure across all types of projects. Consequently, the project structure for incorporating such workflows, as well as the files present in these workflows are standardized, therefore they can provide valuable data regardless of the programming language or the purpose of the project. Due to the differences in Maven files and workflow files, this research is conducted separately for each type of configuration. Nevertheless, many of the techniques employed for analysis can be applied to both configuration types, as elaborated further in subsequent sections. In essence, throughout this research, we aim to address the following fundamental question:

"Is it feasible to detect similar GitHub projects in terms of their build configurations?"

While answering this question, we also answered the following sub-questions:

- How can we compare workflow and Maven files and what metrics can we use in doing so?
- How can Maven analysis be useful for discovering similar projects in terms of their build configurations?
- What patterns can we identify when analyzing workflow configurations?
- How does the analysis of build configurations differ from other industry-standard project similarity detection tools?
- What are the recommendations for related future work?

There are some existing papers and frameworks that aim to analyze OSS project similarity. Frameworks such as RepoPal revolutionized similarity search effectiveness by considering two key data points: ReadME files and GitHub stars [15]. MUDABlue [6] used a more intuitive approach, where the main focus is on analyzing the source code similarities. However, build configuration analysis has not yet been adequately addressed by any of these frameworks. This research aims to bridge this gap by incorporating some of the techniques used by frameworks such as RepoPal and MUDABlue, and extending them for build configurations and Maven configurations, followed by a comparative analysis of the results.

This research is structured as follows: The Methodology section describes the selection of configuration files and the techniques employed in analyzing the similarity of these files. This section also provides the metrics used to analyze the effectiveness of these techniques. The Experimental Setup and Results section provides detailed information about mining the projects analyzed, a high level overview of the experiments that we decided to run, and finally give the results we have achieved. In the Framework Comparison Section, we delve into the investigation of the differences between our study and an existing project similarity detection framework. In the Responsible Research section, we reflect on the ethical aspects of this research and discuss the reproducibility of the methods used. The Discussion section provides our thoughts on potential meanings of the results, as well as discuss our

limitations regarding this research. The Future Work section notes our recommendations for future research and possible improvements that could be further studied. Lastly, the Conclusion section summarizes the contents and key findings of this research.

In this research, we've explored the feasibility of identifying similar GitHub projects based on Maven files and workflows. Our findings reveal interesting results in build configurations, demonstrating the potential for effective project similarity detection. Additionally, insights into patterns across diverse projects were unveiled through the analysis of workflow configurations. As we delve into subsequent sections, we'll detail the metrics and techniques employed, present experimental results, and discuss the implications. This research fills a crucial gap in existing frameworks, challenging the traditional view of project similarity.

2 Literature Review

Several existing frameworks attempt to automatically categorize open-source software projects. For instance, MUDABlue [6] analyzes source code to assign relevant categories, utilizing a self-learning approach that evolves the category set dynamically. RepoPal [15] leverages topic modeling to group projects based on semantic similarities in their documentation and descriptions. In addition, it takes into consideration other project information, such as users who starred the project, in order to determine the groups of similar projects.

However, a crucial aspect often overlooked by these frameworks is the role of build configurations. These configurations specify how a project should be compiled and executed, significantly impacting its functionality and target environment. Ignoring these details can lead to inaccurate or incomplete categorization, hindering effective project discovery and collaboration.

Consider two projects—one configured for web deployment and another for embedded systems. Despite their distinct scopes, both projects extensively share build configurations, featuring similar workflow pipelines for automated tests, code linting, and build deployment. While they may use the same programming language, such as Java, and be configured with Maven, RepoPal [15] is likely to categorize them differently due to varying project scopes, leading to differences in documentation and audience interest. MUDABlue [6] may accurately identify the programming languages but overlook the shared build configurations. Thus, recognizing their analogous build configurations, tailored to different execution environments, becomes essential for unveiling potential cross-domain knowledge transfer in terms of build configurations.

3 Methodology

This section comprehensively explains the collection and preparation of the dataset, the selection of configuration files, the similarity analysis techniques employed, the evaluation metrics used, and the clustering approach used to group similar projects. Our primary aim in this section is to make this paper as reproducible as possible.

3.1 Data collection and preparation

In this research, we aim to capture the similarity in the diverse approaches used to create build configurations, such as Maven's Project Object Model files (pom.xml) and workflow configuration files. Hence, collecting as much data as possible about different GitHub projects is crucial in potentially unraveling patterns and trends that might have been unnoticed before.

Since Maven files are intrinsically different from workflow configurations, we decided to collect 2 different datasets of projects, each containing either projects that have been configured using Maven or projects containing workflow configurations. This decision is based on 2 key reasons. The first one is that, by only selecting projects that meet both criteria, we would essentially limit the total number of projects that can be analyzed, hence leading to a smaller dataset of projects. The second reason arises from the nature of workflow files. These files can essentially be configured for any project, regardless of the programming languages used. However, Maven is predominantly used for Java projects, therefore restricting the project space of the research to Java projects only. Hence, this research is essentially divided into two sub-studies, each focusing on a specific configuration type. As such, the foundation of our analysis lies in a curated dataset of approximately 743 OSS projects from GitHub configured with Maven (containing a total of 398 different plugins), as well as 846 projects containing workflow configurations with a total of 5345 files. To ensure the relevance and quality of our dataset, we selected projects that meet the following criteria:

1. Targeted Relevance: Projects were exclusively selected if they contain at least one file named "pom.xml" for Maven-based projects or a ".yml" file residing within a directory containing ".github/workflows" (hereafter referred to as pipeline files) for the workflow analysis. These criteria ensured a baseline level of build configuration information for future analysis.

2. Well-Structured Build Configurations: Only projects that have a correct configuration and structure of the Maven files (pom.xml files) have been selected. This was verified by employing the native Maven tools to build the "effective-pom.xml" file. Projects that either failed to build or exceeded a predetermined time threshold of 30 seconds were excluded to maintain the integrity of the dataset. We found that the average time needed for Maven tools to build the "effective-pom.xml" file is approximately 5.47 seconds with a standard deviation of 5.31 seconds, therefore we believe that a threshold of 30 seconds is generous enough to ensure that properly configured projects have enough time to be built.

The data has been collected using an algorithm that makes use of the GitHub API to query data from it. The algorithm is split into 2 phases. First, it uses the GitHub API to search for projects that meet the first criterion mentioned above. The resulting repositories that match the criterion are aggregated in 2 separate text files, retaining their unique identifier in the

format of "username/project_name". Given the research's focus on analyzing build configuration files, we decided that the majority of files within each project should be discarded. Only the "effective-pom.xml" file generated by Maven tools and the pipeline files were retained, forming the core datasets utilized throughout this paper. As such, the second phase of the algorithm consists of individually searching through the lists of projects and saving the needed files, each file being saved in its corresponding project folder. The algorithm, datasets, as well as all the files that have been saved, were made publicly accessible in an online repository [7], in order to provide transparency and a possible basis for future research.

In addition to the datasets of projects retrieved by using the GitHub API, we used another 2 datasets for validating our results. These datasets were used to compare our results with other frameworks. These datasets consist of projects that can be found within the CrossSim [9] repository. It is interesting to note that we chose these projects as validation sets because it allows us to use the same dataset of projects that have been used throughout the CrossSim benchmarking, hence mitigating potential biases in our data. In total, the CrossSim project mentioned 580 projects. These projects were once again filtered using the aforementioned criteria, resulting in a set of 340 Maven projects and 80 projects containing pipeline files.

For Maven projects, an additional data preparation step was taken. After each Maven file has been parsed, data regarding each build plugin has been stored in a project-plugin table, containing every plugin found in any of the pom.xml files extracted. Here, the term build plugin refers to plugin data that can be found encapsulated in the "build" tag of the effective-pom.xml file. We then augmented this table with an additional column containing the Java version information of the projects

3.2 Similarity Analysis

To capture the complex relationships between build configurations, we employed several similarity analysis techniques.

Initially, we started by vectorizing each data point. Here, data points can either be a pipeline file or a textual representation of every piece of data regarding a project from the project-plugin table (e.g, a row in this table). This step is crucial to transform the documents into vectors of numeric values, which can then be used to compute the similarity. The vectorization process is done by making use of the Term Frequency - Inverse Document Frequency (TF-IDF) encoder [3]. This technique is used to determine the relevance of each word, such that the most common words that can be found across the datasets of files are given a low relevance value, while a high relevance value is assigned to the rare words. According to J. Beel et al. [1], 83% of text-based recommender systems use TF-IDF to encode the text of the documents. Related literature, such as RepoPal [15], is no exception, as the paper suggests making use of this technique to encode the ReadME files to determine project similarity.

The second step of the analysis consists of calculating the cosine similarity in the vector space of the files [10]. The usage of this metric, in combination with TF-IDF for producing the vectors of the files, is also suggested by other related literature such as RepoPal [15], MudaBlue [6], and CLAN [8]. The cosine similarity metric, which produces values ranging from 0 to 1, where 0 denotes no similarity and 1 signifies that the files are identical, has been applied to each pair of projects, and the resulting computations are stored in a similarity matrix. This matrix has then been used throughout the rest of the research, serving as a foundation for subsequent clustering and analysis.

3.3 Clustering and visualization

In this research we aim to explore possible hidden patterns within the build configuration files, therefore we decided to use the K-Means clustering algorithm [11] to visualize the data. This algorithm is often used in unsupervised learning and pattern analysis, therefore, it perfectly fits the scope of this research. K-Means clustering works by partitioning the dataset into k different clusters, such that the resulting clusters contain the closest observation to the cluster's centroid. Additionally, according to W. Usino et al. [2], the K-Means algorithm can be used effectively with the cosine distance metrics.

The number k is a predefined number of clusters, which we identified by using the "elbow method". The elbow method serves as a heuristic for identifying the optimal number of clusters within a dataset. This technique involves graphing the intra-cluster variation against the number of clusters and selecting the bend or "elbow" in the curve as the optimal cluster count.

The dimensionality of the data is directly linked to the number of projects in the dataset. As such, each of the produced vectors is represented in a dimension far greater than 3, which is the maximum dimension we can visualize. Therefore, employed yet another technique to produce a set of clusters that we can visualize, namely Principal Component Analysis (PCA). PCA is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional space while preserving the most relevant information. Hence, we applied PCA on the similarity matrix, therefore reducing the vectors to 2 dimensions to facilitate cluster visualization.

4 Experimental Setup and Results

This section details the experimental setup and findings, covering the collection of GitHub projects, Maven build configurations, and GitHub workflow analyses. The project dataset was obtained using GitHub's API, with a focus on querying and storing results programmatically. Maven analysis involved plugin extraction, Term Frequency-Inverse Document Frequency(TF-IDF) vectorization, and K-Means clustering for categorization. Workflow analysis, while encountering challenges in content-based categorization, revealed an intriguing correlation between file similarity and their parent organization. The interpretation underscores the significance of build configurations in project analysis and introduces potential recommendations based on organizational patterns.

4.1 Project collection and preparation

The initial step of this research's experiments consists of collecting a large dataset of projects that can later be used in the analysis. To retrieve this set of projects, we used GitHub's

API, which is publicly accessible and built and maintained by GitHub itself. However, manually querying the API can be rather time-consuming, therefore we decided to make an algorithm that programmatically queries the API and then processes and stores the results for future use. For this task, we used and extended an open-source GitHub API wrapper[1] that allowed us to access GitHub's API within a Java application. We extended this wrapper by adding methods for specifically searching for projects with build configurations and storing these project names in a new text file. As such, each line of the text file contains the name of the author of the repository and the repository name, separated by a slash. On GitHub, the combination of these two data points can be used to identify a project uniquely. Searching GitHub was based on a Code Search query, hence only projects that matched this criteria were taken into consideration. Code Search queries only retrieve the files within projects that match the given criteria, however, we decided to only collect the name of the projects each file belongs to at this stage.

Since the goal is to retrieve as many projects as possible that either use Maven configurations or have GitHub workflows configured, we separated the search process into two steps, each with its corresponding query. Therefore, the query for selecting projects with Maven configured is "path:*pom.xml", while projects with Workflows configured were selected using "path:.github/actions/**.y*ml". The former query selects all projects containing a pom.xml file, since this file is specific to Maven and every Maven project requires at least one file named like this. The second query selects all projects that have a subfolder structure ".github/actions" present, within which there can be an arbitrary number of subfolders that must contain files with either ".yml" or ".yaml" extensions. GitHub actions require this type of project structure to be able to run workflows on a project, hence every project that has some workflows correctly set up will be a part of the set returned by this query.

For projects configured with Maven, we added an additional step; we used the Maven build tools to build the project and extract the effective-pom file. This ensured that the projects were correctly configured and could be run within a reasonable time frame. It was necessary to include this step because some projects were either improperly configured or contained plugins that resulted in build errors. Improperly configured projects ranged from projects that had not been maintained in a long time to projects that did not include a pom.xml file at the root of the project. We decided that the threshold for considering the build time reasonable would be 30 seconds. Further details for deciding to set the threshold to 30 seconds can be found in the Methodology Section. These additional conditions reduced the number of projects from 873 to 743.

4.2 Maven analysis

The goal of analyzing Maven configurations in projects is to unravel potential patterns and conduct an in-depth analysis of project similarity. An important step in this endeavor is the extraction of crucial information from the effective-pom.xml file, specifically focusing on understanding the intricate details of build plugins and their configurations within each project. Therefore, the initial step for projects containing a Maven configuration was to extract the relevant information about the build plugins and their configurations within the effective-pom.xml file. Hence, a table with all the information was created such that:

- Each row of the table represents an individual project.
- Each column represents one of each of the plugins found in any of the pom.xml files. Additionally, we appended a column containing information about the Java version used throughout the project.
- Each project-plugin value contains all the plugin configuration data for its corresponding project. Configuration data contains information regarding the plugin name, its version, as well as the additional configuration and execution setups if present. All this information is structured as a JSON object and stored as a string. An example of such an object can be found in Appendix B.

Hence, the final table consists of 743 rows and 1 Java version column as well as another 398 plugin columns. For this amount of selected projects, visualizing the data becomes difficult as the images produced are cluttered, hence a subset of 20 random projects has been used for visualizing the techniques explained below. In Appendix A, the visualizations of the complete sets of projects are provided, however, this section will only describe the results of the smaller subset. In the next sections, we will discuss the insights provided by analyzing the full set of projects, hence the subset is only used for facilitating the data visualizations.

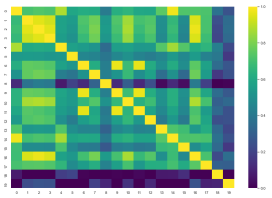

Figure 1: The normalized cosine similarity matrix of the Maven projects for a subset of 20 projects.

Each row of data corresponding to a project has then been converted to text and vectorized using the Term Frequency-Inverse Document Frequency(TF-IDF) vectorization technique [3]. This vectorization technique essentially extracts the most relevant information regarding each plugin and disregards the most common keywords. It is important to note that, for the scope of this research, we decided against further stemming the plugin data. Therefore, keywords such as "artifactId" are present in every single data point. This decision is based on the fact that common words are automatically assigned a very low relevancy score by the nature of TF-IDF vectorization. While stemming could potentially improve the accuracy of the results and reduce the vector space of each project, possibly increasing the efficiency and scalability of the algorithm, it does not bring additional information regarding the feasibility of analyzing build configurations, therefore it is beyond the scope of this research.

[1] https://github.com/hub4j/github-api

Another benefit of vectorizing the data is that, instead of text, we are now left with a vector of numbers. While it is not interpretable by humans, it enables us to effectively use the cosine similarity metric [10] to measure the distance between these vectors, as seen in related literature such as RepoPal [15] and MUDABlue [6]. This computation has been applied for every pair of projects, and the results have been stored in a similarity matrix. This matrix was then normalized, such that the minimum value is 0, representing completely dissimilar projects, and the maximum value is 1 for identical projects. Figure 1 represents the visualization of the normalized cosine similarity matrix for a subset of 20 projects, for easier visualization.

The cosine similarity matrix was then used to cluster projects using the K-Means clustering algorithm [11]. In order to define the optimal number of clusters to be used, we employed the elbow method. Figure 6 (found in Appendix A) depicts the results of the elbow method, where the within-cluster-sum-of-squares is plotted on the y-axis, and the number of clusters is plotted on the x-axis. Hence, we can see that for the same subset of 20 projects chosen initially, the optimal number of clusters is 3. However, the dimensionality of the data remains very high, therefore visualizing the clusters in 2 dimensions is impossible without applying a dimensionality reduction technique. Overcoming this problem has been done by making use of the Principal Component Analysis(PCA) procedure to reduce the data to only 2 dimensions while aiming to preserve as much vector information as possible. PCA is commonly used across different studies to plot data in 2 dimensions, enabling the visual identification of clusters among closely related data points, according to I. Jollife et al. [5]. With the optimal number of clusters defined and the vector dimensionality of only 2, we have now employed the K-Means clustering algorithm. Figure 2 depicts the clusters created by this algorithm. In this image, the closer a project lies to another project, the more similar they are in terms of the Maven file configuration.

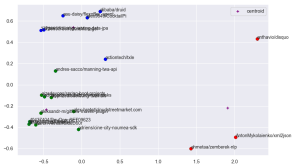

Figure 2: The resulting 3 clusters for the subset of 20 Maven projects.

4.3 Workflow analysis

The projects containing workflows have been analyzed in a relatively similar manner. Hence, the key procedures of analysis have been preserved, such as encoding the contents of the files using the TF-IDF encoder, producing a cosine similarity matrix, applying PCA to reduce the dimensionality for easier visualization and clustering the data by employing the Elbow-method technique together with the K-Means clustering algorithm. However, one of the key differences is we decided to compare the files themselves, rather than the projects containing them. After applying the elbow method, we found that the optimal number of clusters is 16. Figure 3 depicts the resulting clusters.

Figure 3: A plot of every workflow file, belonging to one of the 16 possible clusters.

The natural intuition is, of course, to interpret these clusters as possible categories of files. Based on our knowledge and past experience with configuring workflows, certain types of workflows are used more often than others. Naturally, we decided to test whether the clusters shown in Figure 3 correspond to the most common types of workflows. One of the key observations in testing this hypothesis was that the majority of files have certain keywords in the name of the file that can help us identify which type of workflow a particular file corresponds to. Therefore, we created the following mapping of possible types and their corresponding keywords that a file needs to have to belong to that particular category:

- Test: ”test”, ”unit”, ”integration”
- Build: ”build”, ”compile”, ”make”
- Deploy: ”release”, ”deploy”, ”publish”
- Project Management: ”board”, ”issue”, ”label”, ”milestone”, ”pull”, ”request”, ”assign”, ”ticket”
- Documentation: ”doc”, ”readme”, ”license”
- Package: ”package”, ”pack”, ”artifact”
- Docker: ”docker”, ”container”
- Dependency: ”dependency”, ”dependencies”, ”deps”
- Static Analysis: ”lint”, ”checkstyle”, ”codeql”, ”sonar”, ”static”, ”analysis”, ”check”, ”audit”
- Formatting: ”style”, ”format”, ”prettier”
- Plugin: ”plugin”, ”extension”, ”api”, ”bot”

The clustering process produced a total of 16 categories, however, the distance between some of these clusters was rather small and the sets were fuzzy, hence, we decided that these are the main categories that should be considered. The data points in the previous plot, each corresponding to a workflow configuration file, have been re-labeled based on the newly found category. However, the results were unexpected. The metric used to compute cluster similarity between these 2 labelings is the Rand Index adjusted for chance [4]. This measure computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs assigned in the same or different clusters in the predicted and true clustering. The computed adjusted Rand Ind value for these clustering is 0.03. Therefore, the clusters built by analyzing the contents of the file and the clusters built on

the file categories are minimally similar. This can also be seen in Figure 4. Here, Figure 4 only shows files belonging to the Test, Build, and Deployment categories for easier visualization as adding the rest of the categories produces a very cluttered image, however, the image generated with every category included has been included in Appendix A for completeness. Therefore, we concluded that analyzing the file's contents is not a viable method for discovering the category of the file, since the differences between the contents are not significant enough, and the categories overlap in their corresponding vector spaces.

Figure 4: A plot containing all the workflow files belonging to one of the following categories: Test, Build, Deploy.

Although clustering based on the file's content did not produce significant results in understanding which type of file it is, we noticed another interesting fact about the clusters. Files tend to be more similar to other files created by the same user/organization, regardless of which type of file it is or which project it belongs to. In order to measure this, we decided to measure the average similarity between 2 randomly selected files from all the available files, as well as the average similarity between files created by the same user or organizations. Since this process is computationally expensive, we decided to approximate the averages by making use of the Law of Large numbers [14], thus repeating the selection of the 2 random files for a total of 1000 times. Consequently, we obtained the following results: For files that have been randomly selected from the complete set of files, the average similarity is approximately 0.17, while files belonging to the same organization have, on average, a similarity of 0.56. Figure 5 depicts the first 5 organizations with the biggest amount of files created. Therefore, it becomes clear that files created by the same organization are more likely to have similar contents, regardless of the file's category.

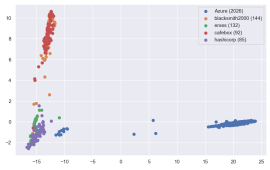

Figure 5: A plot with the workflow files belonging to the 5 organizations with the most total files created across all projects.

5 Framework Comparison

In this section, we embark on a detailed comparison between our framework and RepoPal [15], a well-established tool for analyzing project similarity on GitHub. Noteworthy is the fundamental distinction in their focus: RepoPal's results primarily focus on the semantic analysis of documentation files, while also incorporating project metadata such as the users who starred the projects. In contrast, our framework delves into the analysis of build configurations, encompassing Maven pom files and workflow files. The motivation underlying these comparisons differs from benchmarking performance; it seeks to uncover whether our framework possesses the capability to identify similarities that may have eluded detection by existing frameworks such as RepoPal. Currently, RepoPal is one of the frameworks that achieve the best results in terms of project similarity detection. The other framework with high-quality results is CrossSim [9], however, it was designed as a tool that can be extended to effectively analyze different project features. Extending CrossSim to incorporate build configurations is beyond the scope of this research, therefore we decided in favor of RepoPal as a comparison framework, which has a predetermined set of features that are analyzed.

The comparisons of Maven and workflow files have been conducted separately. In order to compare the similarities, we used the project dataset provided by CrossSim as a validation dataset. We decided to use this validation dataset to mitigate potential biases in our project selection. The Methodology Section explains the reasoning behind this decision in more detail. Since the CrossSim framework used this dataset to compare results with other frameworks that have not been designed to measure build configuration similarity, some projects in the dataset do not have Maven or workflows configured. These projects have been therefore ignored. As such, out of the 580 projects provided, only 340 of them have been used for the Maven analysis, and only 80 for workflow analysis, respectively.

The initial step of the comparison has been to run RepoPal and store the results in text files created for each project. For Maven analysis, the process of comparing our tool with RepoPal went as follows: we selected a project and computed its cosine similarity to the rest of the projects. For each such computation, we also looked in the RepoPal results folder and retrieved the cosine similarity between the selected project and the other projects. This resulted in 2 different arrays of numbers, where each array contains either the similarity computed by us or the similarity computed by RepoPal. The indices of the arrays corresponded to the same project in both arrays. In order to test whether the results are correlated, we used Spearman's rank correlation coefficient [12]. This correlation coefficient is particularly useful in this scenario because it essentially checks whether the ranking of the results is correlated. As such, the resulting coefficient is high in a scenario where the similarity scores are different, as long as the ranking is preserved. This process has then been repeated until each project has been selected. Consequently, we found that the average Spearman correlation between our results and RepoPal's results is 0.033, with a standard deviation of 0.05. Workflow similarity has been tested in an identical manner. The resulting average Spearman correlation coefficient is 0.13, with a standard deviation of 0.10. These correlations indicate a very weak, almost non-existent, positive cor-

relation. Therefore, we can conclude that our results are very different from RepoPal. However, these are the expected results, since these tools are designed to measure different data about projects.

6 Responsible Research

In conducting this research, we are committed to upholding the highest standards of ethical conduct, transparency, and openness. This section outlines key principles guiding our approach, including ethical considerations, responsible data collection and usage, and our dedication to reproducibility and openness. By adhering to these foundational elements, we aim to not only ensure the integrity of our study but also foster collaboration, knowledge sharing, and the advancement of research within the broader scientific community.

Ethical Considerations This research project adheres to ethical principles to ensure responsible and respectful treatment of data. We prioritize transparency and open access to our methodology and findings, fostering collaboration and knowledge sharing.

Data Collection and Usage We diligently adhere to data collection best practices, ensuring that the data used in this research has been collected ethically. We only selected open-source projects from GitHub, a publicly accessible repository, ensuring that no unauthorized access to private or sensitive data was obtained. The dataset used in this research has been made publicly available in an online appendix [7], promoting transparency and facilitating further research.

Reproducibility and Openness To promote reproducibility and facilitate independent verification of our results, we provided detailed documentation of our methodology, including data collection and preprocessing steps, similarity analysis techniques, and clustering algorithms. We also made the algorithms used in this research publicly available [7], enabling other researchers to replicate our findings and extend our work.

7 Discussion

Our analysis of build configuration similarities in open-source projects has shed light on intriguing patterns and raised promising questions for future exploration. By focusing on Maven files and GitHub Actions workflows, we aimed to gain insights into how similarities in building and deployment tools and configurations might reflect deeper connections between projects.

After the Maven analysis, we manually checked some of the results. An example of one of the findings is that the projects "amuthansakthivel/SDET"[2] and "8thlight/CoffeeMaker"[3] appear to be remarkably similar in terms of the Maven plugins used and their corresponding configurations. The similarity score of these projects is approximately 0.97. Interestingly, while these projects use almost the same plugins, and even the plugin versions are strikingly similar, the scope of the projects is very different. As such, the first project is a framework for testing the code more easily, while the latter represents an API for programmatically interacting with coffee machines. Therefore, projects that traditionally would not be considered similar, can be very similar in terms of their build configurations. Hence, we would argue that it is, in fact, of key importance that build configurations are taken into consideration when finding similar projects to find inspiration regarding possible build tools and configurations. If only similar projects in terms of the project's scope were to be considered, it would be more difficult to provide recommendations of tools that could ease the development, building, and deployment stages of the projects. To prove that traditional methods would not work particularly well for this purpose, we decided to run the similarity analysis with one of the well-established tools also presented in the Literature Review subsection, namely RepoPal. Further information on the process and results of comparing our results with other project similarity frameworks are discussed in the Framework Comparison Section.

Pipeline files revealed yet another interesting observation that arises from the fact that files are more likely to be similar if they have been created by the same organization. This observation is that it could be possible to recommend similar projects in terms of build configurations by extracting similar organizations. For example, in Figure 5, "blacksmith2000" and "cafebox" appear to be similar. This can also be calculated by randomly selecting a file made by each organization and comparing the similarity. If this process is repeated 1000 times, we obtain an average file similarity of 0.55. Conversely, if we do the same process for different organizations, such as "blacksmith2000" and "erxes", the resulting average similarity is only 0.06. We can compare all the pairs of organizations in the same manner and store only the most similar ones. These results can be used, in turn, to recommend projects that might have similar build configurations by recommending other projects within the same organization or projects made by a similar organization. Additionally, precomputing the information about organizations could potentially lead to improved time efficiencies for finding similar projects. Therefore, if, for example, "blacksmith2000" creates a new project, it is already possible to recommend build configurations by looking at other projects made by either themself or "cafebox". The category-mapping can also be used in this process, by recommending files from different projects of similar organizations that correspond with the desired category of the file. For example, if user "cafebox" creates a new file titled "deploy.yml", we can already recommend files corresponding to the same category from the user "erxes". These results could then be combined with the Maven similarities to achieve a final build configuration similarity score. Interestingly, this score can be, in turn, combined with other similarity metrics such as dependency similarity, code similarity, etc. to produce a "true similarity" score. However, the process of obtaining this composed similarity score and retrieving recommendations based on it needs to be further studied and potentially optimized, as it is beyond the scope of this paper.

It is crucial to acknowledge the limitations of our study. Primarily, the analysis focused on publicly available data

[2]https://github.com/amuthansakthivel/SDET

[3]https://github.com/8thlight/CoffeeMaker

from GitHub, potentially skewing results toward popular choices and configurations. While we did not impose any popularity conditions for retrieving data from the GitHub API, we believe that GitHub may return files within larger projects first, simply because the probability of selecting one of the files within such a project is higher. This bias can be removed by adding additional selection criteria, such as the size or popularity of the project, and then selecting an equal number of large and small projects, popular and less popular, etc. Additionally, half of the research focuses on Java projects configured with Maven. However, Maven is not the only tool that can be used to configure Maven projects. While we believe that this analysis would work on other build tools such as Gradle, this has not been tested throughout the research. Therefore, the validity of the results may be restricted to Maven. In the future, this can be mitigated by further studying other popular tools, for example, Gradle and Docker, as well as Makefiles, etc.

To conclude, our research unveils the potential of analyzing build configurations as a useful tool for uncovering hidden connections between projects. This novel approach challenges traditional notions of project similarity and offers promising views for knowledge sharing in the software development community.

8 Future Work

This study aims to pave the way and establish a foundation for future analyses of build configurations by exploring their feasibility and potential techniques, an area that has not been adequately investigated. While this research lays the groundwork, numerous unexplored avenues await future exploration. We propose a set of experiments and studies for the future to uncover additional insights, as well as enhance and optimize the build configuration analysis.

Expand the data pool: Incorporating projects from other platforms beyond GitHub can provide a more comprehensive and diverse perspective on build configuration similarities. More importantly, other build configurations such as Gradle files, Docker files, Kubernetes files, Makefiles, etc. can be taken into consideration for a more complete and powerful analysis framework. Furthermore, augmenting the data collection process with additional facets, such as gathering information on the frequency of updates of the chosen files, the pipeline runtimes, and other relevant metrics, can refine the dataset for a more nuanced analysis.

Explore alternative analysis techniques: Utilizing different clustering algorithms and similarity metrics can help mitigate potential biases, such as organization similarity, and reveal additional insights into project relationships. For example, X-Means clustering could be potentially utilized to cluster the projects. This algorithm would remove the need for employing the elbow method, which would, in turn, potentially create a more robust algorithm that needs less human supervision and analysis. Additionally, metrics such as Jaccard similarity could be used to analyze the similarity of Maven files, where only the presence or absence of certain plugins would contribute to the similarity score. Furthermore, different vectorization algorithms could provide additional similarity accuracies. Transformer-based encoders are an example of such vectorizers, and they are commonly used in Natural Language Processing as they can encode large amounts of text while preserving the context of the files, which could also potentially prove useful for this analysis.

Investigate specific clusters: Delving deeper into the identified clusters can provide a richer understanding of the shared challenges, tools, and best practices within each group. We discussed the observations made throughout this research, however, further manual analysis is needed to ensure there are no underlying patterns left uncovered for each cluster.

Develop practical applications: Translate the insights gained into actionable tools and recommendations that support developers in choosing appropriate build configurations for their projects, in real-time. For example, the insights gained in this research can be used to develop recommender systems that automatically detect the configurations of the project a developer is working on and queries GitHub to discover similar projects, then automatically recommend different types of configurations that might be useful.

Add complexity: Integrating results from this framework with other aspects such as code similarity, documentation similarity, dependency similarity, etc. could offer a more comprehensive understanding of project complexity. This multi-faceted approach provides a more nuanced perspective on the intricacies of projects.

Enhance Data Preprocessing: Applying stemming techniques to the collected data could also improve the accuracy of the algorithm. Different stemming techniques could be used for this purpose, such as ignoring plugins that are essentially in every project for Maven analysis. The plugin version could also be stemmed since most of the plugins use semantic versioning, in the form of Major.Minor.Patch. Therefore, the patch values could be discarded, since they do not provide any additional functionalities. For example, the version "3.3.0", would essentially become "3.3" and all plugins that have a version starting with this value would be considered identical.

9 Conclusions

Our exploration of build configuration similarities has yielded valuable insights into the hidden relationships between projects. By analyzing publicly available data from GitHub, we have demonstrated that it is, in fact, feasible to analyze project similarity through the lens of build configurations. Furthermore, we discovered that projects with seemingly different goals can share similarities in their build tools and configurations.

The visualizations of the cosine similarity matrix for Maven projects provide a preliminary glimpse into the landscape of project similarities. This matrix, coupled with the application of K-means clustering offers a more structured understanding of these relationships(Figures 1 and 2). This revealed distinct clusters of projects sharing similar build configurations, even across diverse project types. Further analysis of specific clusters yielded fascinating insights. We observed that projects with seemingly disparate

functionalities, such as ”amuthansakthivel/SDET” (testing framework) and ”8thlight/CoffeeMaker” (API for coffee machines), showed remarkable similarity in their build configurations. This suggests that projects, though very different in their goals, may share common ground in terms of how they are built. This opens up exciting possibilities for cross-pollination of knowledge and tools across seemingly unrelated projects.

Additionally, it is interesting to note that our analysis highlights the potential of pipeline configurations as a novel lens for identifying projects facing similar development challenges or utilizing toolsets. For instance, the clustering analysis might reveal groups of projects, such as projects being developed by the same organization or a similar one, that are, for example, grappling with specific optimization issues or security concerns, despite belonging to different application domains or being developed in different programming languages. This knowledge can empower developers to discover potentially relevant solutions and best practices by looking beyond their immediate project context.

By building upon these conclusions and exploring the ideas of future work, we can further understand the potential of build configuration analysis as a tool for the software development community and begin unveiling unexpected connections between projects, even across different programming languages and application domains. This novel approach challenges traditional notions of project similarity, highlighting that projects surpass the limits of source code analysis due to their inherent complexity, thus laying the foundation for further exploration, which can ultimately lead to more efficient, innovative, and robust software development practices.

A Additional visualizations

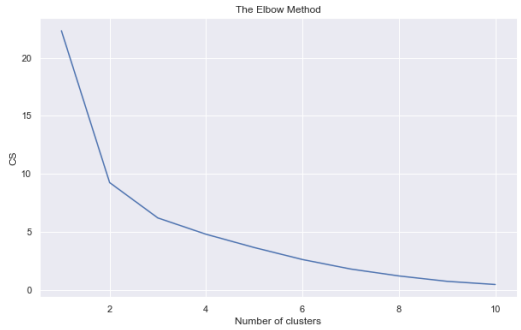

Figure 6: The Elbow Method plot for the subset of 20 Maven projects.

Figure 7: All of the available workflow files, each belonging to one of the defined categories.

B Data examples

The corresponding value in the project-plugin table for project "27786653/ScheduleManager" and plugin "maven-jar-plugin" is:

```
{
  'artifactId': 'maven-jar-plugin',
  'version': '2.5',
  'configuration': {
    'archive': {
      'manifest': {
        'addDefaultImplementationEntries':true
      }
    }
  }
}
```

The complete code, project-plugin table, as well as additional examples and visualizations, can be found in the Online Appendix [7].

References

[1] Joeran Beel, Béla Gipp, Stefan Langer, and Corinna Breitinger. Research-paper recommender systems: a literature survey. *International Journal on Digital Libraries*, 17(4):305–338, Jul 2015.

[2] W. Usino et al. Document similarity detection using k-means and cosine distance. *International Journal of Advanced Computer Science and Applications*, 10(2), 2019.

[3] Lukáš Havrlant and Vladik Kreinovich. A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). *International Journal of General Systems*, 46(1):27–36, 2017.

[4] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, Dec 1985.

[5] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A*, 374(2065):20150202–20150202, Apr 2016.

[6] S. Kawaguchi, P.K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. In *11th Asia-Pacific Software Engineering Conference*, pages 184–193, 2004.

[7] Calin Manoli. Analyzing Similar Build Configurations Across Different GitHub Projects, Jan 2024. https://doi.org/10.5281/zenodo.10577178.

[8] Collin Mcmillan, Mark Grechanik, and Denys Poshyvanyk. *Detecting Similar Software Applications*.

[9] Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of github repositories. 28(2):595–631, jun 2020.

[10] Faisal Rahutomo, Teruaki Kitasuka, and Masayoshi Aritsugi. Semantic cosine similarity. In *The 7th international student conference on advanced science and technology ICAST*, volume 4, page 1, 2012.

[11] Kristina P Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE access*, 8:80716–80727, 2020.

[12] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[13] Brian Vermeer. Jvm ecosystem report, 2021.

[14] Kai Yao and Jinwu Gao. Law of large numbers for uncertain random variables. *IEEE Transactions on Fuzzy Systems*, 24(3):615–621, 2016.

[15] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. pages 13–23, 02 2017.