# Hadoop-common

*Group Members:*
Mitul Nasit,
Arpita Poojari,
Abhijeet Desai,
Gautam Parmar,
Kiran Verma
**Group:** 06
**Git Repo**

# 1 Introduction

The project's motivation stems from the need to comprehend how design choices affect code quality and how design modifications affect software projects. Software engineers often choose critical designs that can significantly affect codebase design, maintenance, and overall quality. However, the direct relationship between these design decisions and their impact on compliance metrics remains a complex and challenging aspect of software development.

Looking to the assignment aims to understand how different types of design decisions manifest in code changes and affect the quality of the software. This understanding is important for improving software development practices, upgrading code maintainability, and making informed decisions regarding design choices.

Moving to Literature support, Literature always shows the importance of studying the impact of design decisions on the quality of the code. The study conducted by Bavota et al. (2015) mentioned that to preserve code quality and efficiently control technical expenses, design decisions must be taken into account during the software development process. Other studies by Olbrich et al. (2009) emphasize the impact of design decisions on software attributes such as complexity, integration, and cohesion.

The assignment aims to investigate and evaluate the relationship between design choices and code quality in the Hadoop-Common project. Find patterns, correlations, and insights that can guide software engineering practices and decision-making procedures by pinpointing particular design issues, extracting code metrics, and performing statistical analysis. We hope to add to the body of knowledge about how design choices affect code quality through this assignment, which uses Hadoop-Common project analysis and empirical data.

# 2 Study Design

For the Hadoop-Common assignment, the research questions are the following:

**RQ1:** How are design decisions reflected in the commit history?

**RQ2:** What patterns can be identified in the implementation of design decisions through commits?

**RQ3:** How do the size and quality metrics of commit changes vary based on the types of design decisions made?

**RQ4:** How to extract dependencies and implement clustering on extracted dependencies?

*Motivation:* The purpose of the research questions is to determine how design choices and code modifications relate to one another in the Hadoop-Common project. The assignment's goal is to understand the decision-making processes and their effects on software development practices within the particular context of Hadoop-Common by examining how design decisions appear in the commit history, spotting patterns in their implementation, and evaluating the impact on code size and quality metrics.

We followed a structured approach outlined in the assignment document to achieve the goal. We started by extracting commit information using the PyDriller library. To traverse through all the commits in the repo and identify the list of commits that are related to the issue list we wrote Python code. By executing the Python code we extracted relevant information from commits like commit hash, parent hash, commit message, files-lines-methods updated deleted, or added, and date-time of the commits in a .xls file to understand the context of each code change. For compiling projects with each commit, we have written a Python script.

Furthermore, we created an automated script with the loop on the commits and compiled the project, and it generated a jar file for each commit.

Using Matplotlib, Pandas, and Scipy libraries, we visualized the size and quality information for each commit, including the number of added, modified, and deleted files, lines of code, and methods. These metrics will provide details about the evolution of the codebase and help us understand the impact of design decisions on code complexity and maintainability.

## 2.1 Dependency extraction

To extract dependencies from commits, we developed a Python script. This script processes JAR files obtained from compiling each commit. Utilizing the JAR files as input, the script meticulously extracts dependencies, capturing them in both .rsf and vector files. This method enables us to systematically analyze the project's dependencies across different commits.

### 2.1.1 PKG, ACDC, and clustering implementation

Taking dependencies extracted .rsf file as input, we wrote Python scripts for PKG, ACDC, and clustering algorithms. As output, it generated a .rsf file for each input dependency's .rsf file

#### PKG Algorithm

Pkg is used to cluster classes based on their packages. This can be used as a baseline to compare it with the results of clustering algorithms. It takes the following parameters:

- Dispatch specifies the path to the .rsf file that contains the dependencies (generated by the parser).
- Project path is the path to place the clustering output.
- The project name is the name of the subject system.
- Project version is the version of the subject system. You can put here the commit ID.
- Language specifies the programming language: C or Java
- File level is a flag to specify if you work on package level or file level. So, it should be true or false.

#### ACDC Algorithm

ACDC is used to execute the ACDC algorithm and it takes two parameters:

- Path to rsf file input (generated by the parser).
- Path to rsf file output.

### Clustering Algorithm

Here, we executed three clustering algorithms WCA with UEM, WCA with UEMNM, and LIMBO. To execute these algorithms, you need to provide certain parameters. Some of these are mandatory and the others are optional. Some parameters require certain values, and others are flags. Mandatory parameters:

- algo specifies which algorithm you will run: WCA with UEM, WCA with UEMNM, LIMBO
- language specifies the programming language: C or java
- deps specifies the path to the .rsf file that contains the dependencies (generated by the parser).
- measure specifies the similarity measure desired. This should be compatible with each algorithm: JS, SCM, UEM, UEMNM, IL, ARCIL, ARCUEM, ARCUEMNM, WJS, PKG. WCA is compatible with UEM and UEMNM. Limbo is compatible with IL, ARC is compatible with ARCIL, ARCUEM, and ARCUEMNM.
- projname is the name of the subject system.
- projversion is the version of the subject system. You can put here the commit ID.
- projpath is the path to place the clustering output.

## 2.2 Analyze the results of clustering algorithms

We followed the provided instructions by implementing the following steps:

**Counting Clusters and Entities:** We wrote a Python script to read the files and count the number of clusters and entities (classes) per cluster for each clustering algorithm per commit.

**Calculating A2A and CVG Metrics:**
We've developed a Python script to analyze architectural changes between a commit and its parent commit, focusing on two metrics: A2A (Architecture-to-Architecture) and CVG (CodebaseView Generator), for each clustering algorithm.

**Storing Data:** We stored all the collected data in a JSON format, keeping it organized with the previous data on quality and size from the previous steps. This comprehensive dataset will be analyzed in the next phase of our project.

# 3 Results

The analysis began by sorting through the issue IDs connected with the unique design decisions for each project. Using PyDriller, commit messages were carefully filtered to discover those related to these design flaws. That Issue IDs not appear in commit messages were temporarily excluded from the analysis. The parent commits linked with the indicated commits were then determined.

The project source code was automatically compiled to simplify the analysis and prepared for future clustering techniques. This included writing Python Scripts that could iterate over the detected commits and compile the project automatically. **We compiled almost all commits**.

Using Matplotlib, Pandas, and Scipy, a variety of size and quality metrics were extracted for each commit. These metrics included the number of added, modified, and deleted files for each commit, as well as data like added and deleted lines of code per file, added, deleted, and updated methods per file.

We did six types of analysis: Lines Added and Deleted Over Time, DMM Unit Size vs DMM Unit Complexity, Histogram of Lines Added and Deleted, Boxplot of Lines Added and Deleted, Scatterplot Matrix of Code Metrics, and Time Series of Commit Metrics for Classes Added, Classes Deleted, Methods Added, and Methods Deleted per Commit.

## 3.1 Lines Added and Deleted Over Time

The trend of lines added and deleted over time is shown in this plot. The commit date is shown on the x-axis, and the number of lines is shown on the y-axis. There are two plots: one for newly added lines and another for deleted lines. The plot illustrates times of high activity or notable changes, offering insights into the evolution of code changes over time.

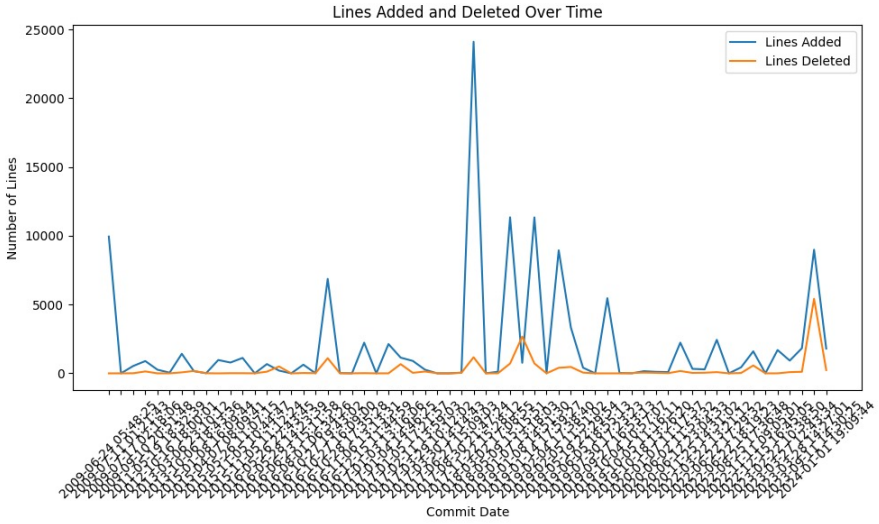


Figure 3.1: Lines Added and Deleted Over Time

## 3.2 DMM Unit Size vs DMM Unit Complexity

This scatter plot illustrates the correlation between the two parameters, unit complexity and DMM (Design Maintenance Metrics) unit size. The DMM unit size on the x-axis and the unit complexity on the y-axis determine the position of each point in the plot, which represents a commit. The plot makes it easier to see if the size and complexity of code units correlate or follow any pattern. Determining the maintainability and architectural quality of the codebase can be made easier with an understanding of this relationship.

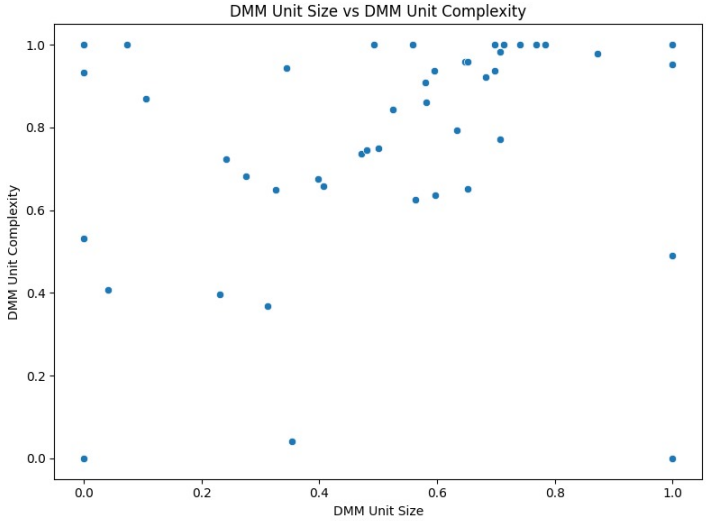


Figure 3.2: DMM Unit Size vs DMM Unit Complexity

## 3.3 Histogram of Lines Added and Deleted

This histogram shows the distribution of lines added and removed between commits. The x-axis represents the number of lines, and the y-axis represents the frequency (or count) of commits. The distribution of added lines is shown by the sky blue bars, and the distribution of deleted lines is shown by the salmon bars. An estimate of the probability density function of the data is given by the KDE (Kernel Density Estimate) curves.

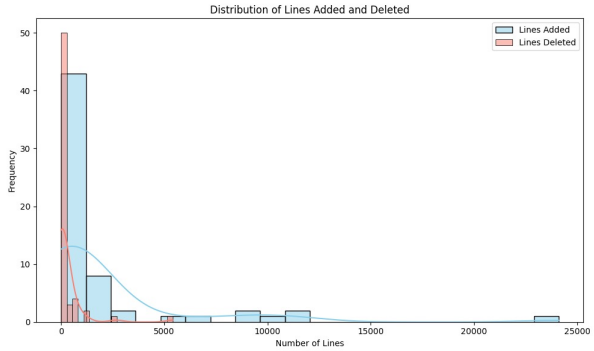


Figure 3.3: Histogram of Lines Added and Deleted

## 3.4 Boxplot of Lines Added and Deleted

This boxplot shows the distribution and central tendency of the added and removed lines. The median, or 50th percentile, of the data, is shown by the central line inside each box. Each box's upper and lower edges correspond to the third and first quartiles, or the 75th and 25th percentiles, respectively. Any data points outside the whiskers are regarded as outliers. The whiskers extend to 1.5 times the interquartile range (IQR) from the box's edges.

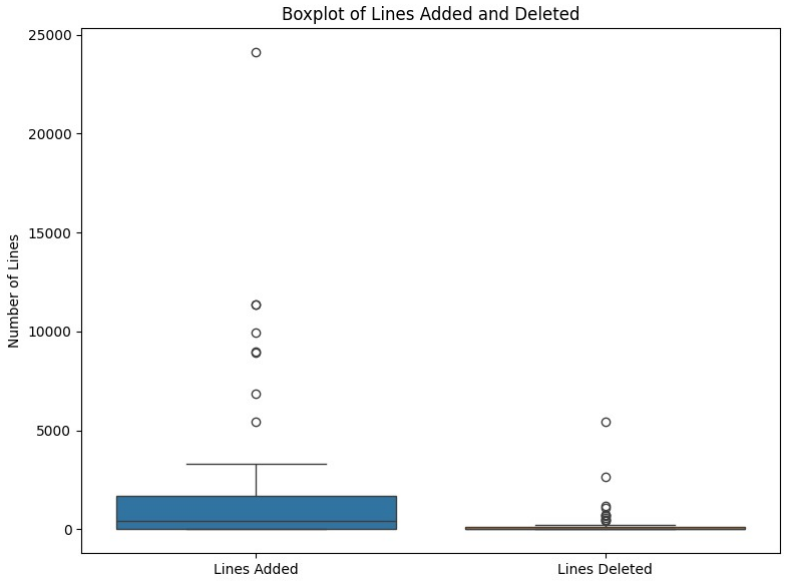


Figure 3.4: Boxplot of Lines Added and Deleted

## 3.5 Scatterplot Matrix of Code Metrics

Pairwise relationships between various code metrics, such as lines added, lines deleted, classes added, methods added, DMM unit size, and DMM unit complexity, are displayed in this scatterplot matrix. A scatterplot of two variables is represented by each cell in the matrix, and the diagonal of the matrix contains distribution histograms for each variable.

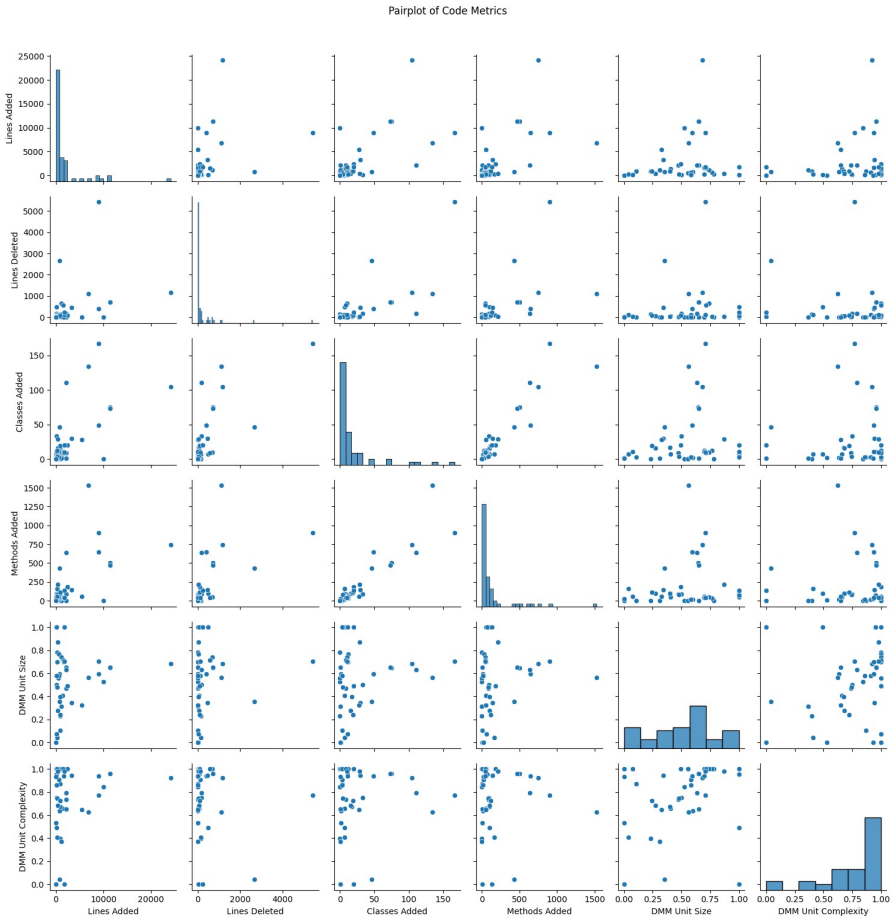


Figure 3.5: Scatterplot Matrix of Code Metrics

## 3.6 Time Series of Commit Metrics

This collection of line graphs shows how different commit metrics have changed over time. Plotted against the commit date, each subplot displays a distinct measure, such as lines added, deleted, DMM unit size, and DMM unit complexity.

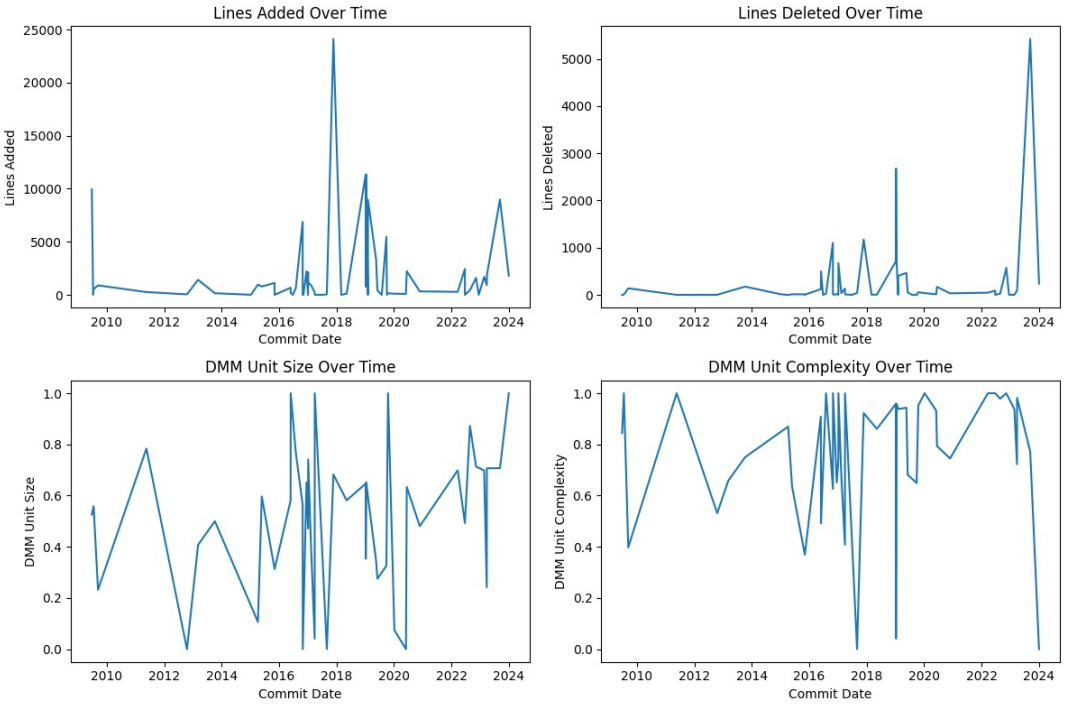


Figure 3.6: Time Series of Commit Metrics

## 3.7 Dependencies Extraction

To extract dependencies from commits, we developed a Python script. This script processes JAR files obtained from compiling each commit. Utilizing the JAR files as input, the script meticulously extracts dependencies, capturing them in both .rsf and vector files. This method enables us to systematically analyze the project's dependencies across different commits. These output files are then stored in a designated folder named **Extracted dependencies** facilitating easy access and further analysis of the algorithm's findings

### 3.7.1 PKG Algorithm

After obtaining dependencies in the .rsf file format, we proceeded by developing Python scripts with the loop to implement the PKG Algorithm. This algorithm, also known as the Package Dependency Graph Algorithm, is a graph-based approach used for analyzing and managing dependencies within software projects. Our Python scripts were designed to incorporate all necessary parameters required by the PKG Algorithm, enabling us to execute the algorithm efficiently. Upon execution, the scripts generate output .rsf files containing the results of the PKG Algorithm's analysis. These output files are then stored in a designated folder named **Pkg outputs** facilitating easy access and further analysis of the algorithm's findings.

### 3.7.2 ACDC Algorithm

After successfully obtaining dependencies in the .rsf file format, our next step involved developing Python scripts tailored for the ACDC Algorithm. ACDC, short for "Agglomerative Clustering for Directed Graphs with Clustering," is a graph-based algorithm utilized for clustering nodes within a directed graph, commonly applied in software dependency analysis.

These Python scripts were meticulously crafted to incorporate all requisite parameters essential for executing the ACDC Algorithm effectively. The ACDC Algorithm operates by iteratively merging nodes based on their similarity, ultimately forming cohesive clusters representative of modules or components within the software project.

Upon execution, the scripts generate an output .rsf file encapsulating the results derived from the ACDC Algorithm's clustering process. To ensure organized management of these outputs, we designated a dedicated folder named **ACDC outputs** for storing the resultant files.

### 3.7.3 Clustering (WCA with UEM) Algorithm

After obtaining the dependencies in the .rsf file format, our focus shifted towards implementing the WCA Algorithm with UEM (Unweighted Coefficient Matrix) with Python scripts. The WCA Algorithm, which stands for Weighted Cluster Algorithm, is a method utilized for clustering nodes within a graph. In this case, it's used for analyzing and managing dependencies within software projects.

Our Python scripts were meticulously crafted to encompass all essential parameters necessary for the effective execution of the WCA Algorithm with UEM. This algorithm operates by assessing the strength of connections between nodes based on their dependencies, ultimately clustering them into cohesive clusters.

Upon execution, these scripts generate an output .rsf file that encapsulates the results derived from the WCA Algorithm's clustering process. To maintain the organized management of these outputs, we established a dedicated folder named clustering output for storing the files.

### 3.7.4 Clustering (WCA with UEMNM) Algorithm

After obtaining the dependencies in the .rsf file format, our next step involved developing Python scripts tailored for the WCA Algorithm with UEMNM (Unweighted Coefficient Matrix with Node Merging). The WCA Algorithm, which stands for Weighted Cluster Algorithm, is a powerful method utilized for clustering nodes within a graph, commonly applied in software dependency analysis.

Our Python scripts were meticulously crafted to encompass all essential parameters necessary for the effective execution of the WCA Algorithm with UEMNM. This algorithm operates by evaluating the strength of connections between nodes based on their dependencies and incorporates node merging to refine the clustering process further.

Upon execution, these scripts generate an output .rsf file that encapsulates the results derived from the WCA Algorithm's clustering process with UEMNM. To maintain organized management of these outputs, we designated a dedicated folder named **WCAUEMNM output** for storing the resultant files.

### 3.7.5 Clustering (LIMBO) Algorithm

After obtaining the dependencies in the .rsf file format, our focus shifted to implementing Python scripts for the LIMBO Algorithm. LIMBO, which stands for Lightweight, Iterative, Modular-Based Overlap, is a graph-based algorithm used for clustering nodes within a software dependency graph.

Our Python scripts were meticulously designed to incorporate all necessary parameters essential for the effective execution of the LIMBO Algorithm. This algorithm operates iteratively, group-

ing nodes into clusters based on their overlapping dependencies. It emphasizes modularity and scalability, making it suitable for analyzing dependencies in large software projects.

Upon execution, these scripts generate an output .rsf file containing the results of the LIMBO Algorithm's clustering process. To maintain organization, we established a dedicated folder named **limbo outputs** to store these output files.

## 3.8 Analyze the results of clustering algorithms

### 3.8.1 The number of clusters and number of entities per clustering algorithm per commit

To analyze the results of clustering algorithms, we developed a Python script that examines the number of clusters and number of entities produced by each clustering algorithm for each commit. This script takes the RSF file generated by each clustering algorithm as input. To traverse through all the RSF files of all clustering algorithms, we implemented a loop within the Python script.

During the iteration process, we calculate the number of clusters for each commit and each algorithm. We stored the output in the JSON files. However, for all the algorithms, we successfully determined the number of clusters and the number of entities.

Our Python script provides valuable insights into the clustering outcomes for each commit and algorithm, aiding in the analysis and understanding of the clustering results within our project.

### 3.8.2 The a2a and cvg metrics for architectural changes between a commit and its parent commit, per clustering algorithm.

We've developed a Python script to analyze architectural changes between a commit and its parent commit, focusing on two metrics: A2A (Architecture-to-Architecture) and CVG (Codebase View Generator), for each clustering algorithm.

The script accepts a directory path as input, allowing it to traverse through the directory to locate the required RSF (Relational Structure File) files. These RSF files contain information about the dependencies between components in the software project.

Once the script identifies the RSF files corresponding to the two commit hashes of interest, it proceeds to calculate the A2A and CVG metrics for these files.

By implementing this Python script, we can systematically assess the architectural changes between commits across different clustering algorithms. This analysis helps us understand the impact of code changes on the software architecture, aiding in the evaluation and management of architectural evolution within the project.

### 3.8.3 The cvg metric between the results of the different clustering algorithms for the same commit.

We've created a Python script designed to compute the CVG (Codebase View Generator) metric between the results obtained from different clustering algorithms for the same commit.

This script operates by taking as input a directory containing the clustering output of all algorithms. It then iterates through the folders within this directory, identifying and selecting two RSF (Relational Structure File) files from different algorithm folders, each corresponding to the same commit hash.

Once the script has obtained these RSF files, it proceeds to calculate the CVG metric between them. This metric provides insights into the differences in the codebase view generated by the two clustering algorithms, indicating variations in how the software components are grouped or organized.

The script performs this process iteratively, generating multiple comparisons between RSF files from different algorithm outputs. As a result, it generates multiple JSON files, each containing the CVG metric computed for a specific pair of RSF files generated by different clustering algorithms.

By employing this Python script, we can systematically evaluate the consistency and differences in the codebase views produced by various clustering algorithms for the same commit. This analysis aids in understanding the robustness and effectiveness of different clustering approaches in capturing the software architecture's structural characteristics.

# 4 Discussion

Based on the study design for the Hadoop-Common assignment, the discussion section focuses on interpreting the findings and implications of the research questions from the researcher's and practitioners' points.

Researchers can understand, how design decisions are revealed in commit histories that offer important insights into the dynamics of decision-making within software development projects. This knowledge can be used in future studies on the long-term effects of design choices on software evolution and guide the integration of design decision analysis into development workflows. Additionally, researchers can use tools like PyDriller and data visualization libraries to learn more about design decision patterns across diverse projects, comparative studies, and advancing research.

Practitioners can make use of insights from this research to improve project documentation and traceability and can improve transparency and knowledge sharing within their teams. By studying implementation patterns and estimating the impact of design decisions on code metrics, practitioners can trim the development processes, identify best practices, and make decisions effectively. Moreover, practitioners can take on tools and methodologies used in the study, such as automated compilation scripts and data visualization libraries, to analyze design decisions in their projects, for improvement and innovation in software development practices.

# 5 Threats To Validity

## 5.1 1st week

In the 1st-week assignment, we focused on exploring the software repository and determining commits that implement design issues. Several potential threats to validity should be considered:

### 5.1.1 Data Quality:

**Threat:** Commit messages may contain inaccurate or incomplete data or there may be issue tracking systems which means there can be misinterpretation of design decisions and their implementation in commits.

**Mitigation:** To validate the accuracy of Commit data, can cross-check information from multiple sources, and ensure consistency in data interpretation to mitigate data quality issues.

### 5.1.2 Tool Limitations:

**Threat:** Dependency on specific tools like PyDriller for commit extraction and analysis may be limitations in data collection and processing.

**Mitigation:** To mitigate tool limitations issues can validate the reliability and accuracy of tools used, and can also explore alternative tools or methods for data extraction.

## 5.2 2nd week

In the 2nd-week assignment, we focused on extracting dependencies and executing algorithms on the extracted dependencies. Several potential threats to validity should be considered:

### 5.2.1 Selection Bias:

**Threat:** The selection of clustering algorithms (WCA, Limbo, ACDC, PKG) may introduce bias if certain algorithms are more suitable for the specific characteristics of the source code being analyzed.

**Mitigation:** Use a diverse set of clustering algorithms and compare their results to mitigate bias. Additionally, consider using ensemble methods to combine the results of multiple algorithms.

### 5.2.2 Dependency Extraction:

**Threat:** The accuracy and completeness of the dependencies extracted from the source code can influence the clustering results. Inaccurate or incomplete dependency information may lead to misleading clustering outcomes.

**Mitigation:**Validate the accuracy of dependency extraction by manual inspection or comparison with existing dependency analysis tools. Implement robust preprocessing techniques to ensure accurate and complete dependency information.

## 5.3 3rd Week

### 5.3.1 Interpretation Bias

**Threat:** Subjective interpretation of clustering results may introduce bias in the analysis.

**Mitigation:** Involve multiple team members or domain experts to review and validate the interpretations of the clustering results. Use consensus-building techniques to ensure objectivity in the analysis.

### 5.3.2 Cluster Size Variation:

**Threat:** Inconsistent cluster sizes across different clustering algorithms may impact the comparison and analysis of architectural changes.

# 6 Conclusion And Future Work

The analysis has laid the groundwork for further exploration into how design choices impact code quality and the occurrence of design modifications within software repositories. A strong foundation has been established by identifying contributions that implement design flaws, automating the compilation process, and extracting size and quality data from commits.

Moving forward, this foundational work will be utilized to investigate variations and relationships between code quality and design modifications for different types of design choices. The aim is to uncover patterns and trends in software architecture development through the analysis of gathered metrics and the application of clustering methods. Ultimately, the research will advance understanding of the connection between design choices and software quality, informing future practices and decision-making in software engineering.

Based on the data and knowledge generated through this assignment's work, In the future, we can analyze the code structure and its design decision-focused development. This information can be helpful to direct the future development of the project as well as it can help in optimizing the code base. It can be used to make a fine balance between efforts spent on which design decision.

# 7 Hours spent in the assignment, and the contribution of each student

| Student Name | Work | Hours |
|---|---|---|
| **Mitul Nasit** | Created bash script and created a code to run provided commits<br>Read Excel file and build the commits to create a jar file<br>Requirements understanding<br>Try to generate jar files of commits and troubleshooting problems while generating jar files<br>Organize project structure on gitlab<br>Updated script of generating jar files<br>Generate output of clustering, dependencies code<br>Generate jar files for hadoop-common project<br>Extracted dependencies generated PKG and WCA for Hadoop common project | 9+14+8 |
| **Abhijeet Desai** | commit metrics<br>stored in an Excel sheet<br>DMM and complexity metric extraction<br>Statistical analysis<br>cluster analysis number of clusters and entities of the cluster per commit per algorithm<br>cvg and a2a metric<br>cvg metric between results of different clustering algorithms for same commit hash<br>reviewed and updated documentation | 8+11+12 |
| **Arpita Poojari** | Created bash script and code, troubleshooting building the commits to create jar files<br>Report: Results and conclusion section<br>Setup Hadoop Project with all the libraries and tried to execute Parent commits and generated Jar files<br>Worked on Report<br>created and ran ACDC, WCA with UEMNM, WCA with UEM script.<br>Reviewed documentation and fixed some sections and related information | 8+10+8 |

Table 7.1: Students Report

| Student Name | Work | Hours |
|---|---|---|
| **Gautam Parmar** | Git Repo analysis<br>analysis of Prydrill and Python code for commit fetching<br>Dependency extraction clustering WCA, Restructuring project,<br>ACDC Dynamic Implementation and PKG Dynamic Implementation<br>Fixed environment setup and missing dependency for compiling the project<br>Fixed code for A2a and cvg algorithm and designed output JSON for meaningful interpretation<br>Reviewed documentation and fixed some of its section with related information | 9+17+8 |
| **Kiran Verma** | Documentation<br>Understand every code to explained it in the documentation<br>PKG Algorithm Static Implementation<br>ACDC Algorithm Implementation<br>Documentation<br>LIMBO Algorithm Implementation<br>Documentation | 8+10+8 |

Table 7.2: Students Report (Continued)