Governors State University
OPUS Open Portal to University Scholarship

All Student Theses | Student Theses

Summer 2023

Automation Script For Evaluation Of Source Codes

Prudvi Raj Manukonda

Follow this and additional works at: https://opus.govst.edu/theses

Part of the Computer Sciences Commons

For more information about the academic degree, extended learning, and certificate programs of Governors State University, go to http://www.govst.edu/Academics/Degree_Programs_and_Certifications/

Visit the Governors State Computer Science Department
This Thesis is brought to you for free and open access by the Student Theses at OPUS Open Portal to University Scholarship. It has been accepted for inclusion in All Student Theses by an authorized administrator of OPUS Open Portal to University Scholarship. For more information, please contact opus@govst.edu.

Automation Script For Evaluation Of Source Codes

By

Prudvi Raj Manukonda

B.Tech in Computer Science
Jawaharlal Nehru Technological University, 2018

GRADUATE CAPSTONE THESIS PROJECT

Submitted in partial fulfillment of the requirements.

For the Degree of Master of Science,

With a Major in Computer Science

Governors State University
University Park, IL 60484
2023

ACKNOWLEDGMENT

I would like to extend my heartfelt gratitude to the following individuals who played a significant role in the successful completion of my thesis:

Dr. Soon-Ok Park, my mentor and chairperson, provided unwavering support and guidance throughout the entire research process. Her valuable advice and assistance were instrumental in shaping the direction of my work.

Professors Freddie Kato and Aslam Shahid, who graciously served on my committee, played a pivotal role in the success of my thesis. Their unwavering dedication and commitment to my research were evident as they generously shared their invaluable expertise, offered profound insights, and provided constructive feedback.

Through their guidance, my research was significantly refined, elevating the quality and impact of my work. I am deeply indebted to them for their support and mentorship throughout this academic journey.

I am especially grateful to Professor Juo Zune for remaining connected to my roots during this thesis. Your contributions and insights have been invaluable, and I am immensely grateful for your guidance.

Dr. Carrington, the division chair, for approving my enrollment in the program of study, which paved the way for this academic journey.

Nancy Rios, the department secretary, for her constant support, counsel, and assistance throughout my graduate studies. Her efficiency and dedication were truly appreciated.

I am especially grateful to McMullen Paula, my academic advisor, for her unwavering support and patience in explaining the process until I obtained authorization for my thesis. Her guidance and encouragement were instrumental in keeping me focused.

Special thanks to Makiko Maria for her tireless efforts in helping me with all the necessary documents and arrangements for my thesis. Her assistance made the process smoother and more manageable.

Finally, I want to express my deepest appreciation to my parents for their never-ending support and inspiration throughout my life. They have been a pillar of strength, constantly encouraging and believing in me.

No number of words can fully convey my gratitude to all these individuals for their contributions to my academic journey. Their support has been invaluable, and I am forever indebted to them for helping me reach this milestone.

ABSTRACT

This thesis focuses on the development of an automation script integrated with a Web application to extract crucial information from .NET projects. The objective was to streamline the process of retrieving database type, database name, and .NET version, build status zip files, generate comprehensive reports, and present key metrics on a dashboard.

The automation script was implemented in Python, utilizing packages such as os, subprocess, zipfile, re, json5, shutil, and xml.etree.ElementTree. The script automated the extraction of information from the zip files, eliminating the need for manual intervention. It executed the .Net build command to determine the success of the build and captured error details if any. The appsettings.json file was parsed to obtain the database type and name, while the csproj files provided the .NET version.

The developed automation script was integrated with a Web application, allowing users to upload zip files and apply the script effortlessly. The application displayed a dashboard presenting statistical insights, including the counts of database types used, the distribution of .NET versions, and the overall success rate of the build process. Reports were generated, providing detailed breakdowns of the build process and error details.

The experimental setup involved using various test files, including sample files representing SQL Server and SQLite databases and files intentionally modified to include build errors. The results obtained from running the automation script on the test files demonstrated its effectiveness and

efficiency in extracting information and generating accurate reports. The script showcased advantages over existing methods and tools, offering simplicity, cost-effectiveness, and flexibility.

The thesis concludes with a discussion of the strengths and limitations of the automation script, potential improvements, and recommendations for future automation efforts. Overall, the developed automation script proved valuable for extracting information from zip-filed .NET projects and demonstrated its potential for enhancing productivity and decision-making in software development processes.

Table of Contents

ACKNOWLEGMENT ii

ABSTRACT iv

List of Figures viii

1. INTRODUCTION 1

1.1 Background on Automation. 1

1.2 Importance of automating. 2

1.3 Purpose of the thesis. 2

2. OVERVIEW OF RELEVANT STUDIES 4

2.1 Overview of existing methods 4

2.2 Review of Relevant Literature on Automation Techniques. 6

3. PROBLEM STATEMENT 8

3.1 Identification of the problem 8

3.2 Challenges and limitations associated with the problem. 9

4. METHODOLOGY 12

4.1 Description of the proposed approach. 12

4.2 Explanation of the selected libraries and tools. 14

4.3 Overview of the steps involved in the automation process. 15

5. IMPLEMENTATION 17

5.1 Resources and Tools Used in the Implementation 17

5.2 Python Automation Script: A brief overview. 18

5.3 Web Application Implementation. 33

5.4 Report and Dashboard Implementation....35
6. RESULTS AND EVALUATION....37
6.1 Experimental Setup and Test Files....37
6.2 Results and Analysis....38
6.3 Evaluation of Effectiveness and Efficiency....41
6.4 Comparison with Existing Methods and Tools....42
7. DISCUSSION....44
7.1 Interpretation and Analysis of the Results....44
7.2 Discussion of the Strengths and Limitations....46
7.3 Potential Improvements and Future Work....48
8. CONCLUSION....50
8.1 Summary of the Thesis Objectives and Contributions....50
8.2 Recapitulation of the Key Findings and Insights....51
8.3 Final Remarks and Recommendations for Future Automation Efforts....52
9. REFERENCES....53

List of Figures

Figure 1. Requirements file 14

Figure 2. Overview of the implementation flow 16

Figure 3. Applying Automation Script gives Build, Database, Version Info 18

Figure 4. Reads database info from appsettings.json file. 28

Figure 5. Reads version info from application's csproj file. 30

Figure 6. Web UI helps to get reports and Dashboard. 33

Figure 7. Test Suites Evaluation Report 39

Figure 8. Admin Dashboard 40

1. INTRODUCTION

1.1 Background on Automation

Automation has transformed the evaluation of programming assignments, benefiting both educators and programmers. Manual evaluation tasks were often time-consuming, leading to delayed feedback. Automation streamlines processes like testing and feedback generation, allowing instructors to deliver prompt and constructive feedback, fostering continuous improvement in programmers' programming skills. Automated testing identifies errors early, enhancing code quality and learning outcomes. Additionally, automation ensures consistent and fair grading, reducing human errors in evaluation. Embracing automation optimizes resource utilization for educators and enhances the learning experience for programmers.

By embracing automation, educators can focus on guiding and mentoring programmers effectively. Automated processes take care of routine tasks, enabling instructors to engage with programmers and provide personalized support. Timely and constructive feedback empowers programmers to understand their strengths and areas for improvement, promoting continuous learning and growth. Automation's consistency in grading ensures impartial evaluation, fostering a fair and equitable assessment process. Overall, automation in programming assignment evaluation creates a positive and impactful experience for both educators and programmers, advancing learning outcomes in the field of programming (Al Sweigart, Automate the Boring Stuff with Python).

1.2 Importance of automating

Automating tasks in programming assignment evaluation holds immense significance, benefiting both educators and learners. By implementing automation techniques, the evaluation process becomes streamlined and efficient, reducing manual efforts and saving time for instructors. Automated test execution allows for swift and consistent assessment of code submissions, ensuring faster feedback delivery to programmers. Furthermore, automation enhances the accuracy of grading, eliminating potential biases and ensuring fairness in evaluations. Test automation helps identify errors and bugs early in the development cycle, promoting higher code quality and better learning outcomes for programmers. Overall, embracing automation in programming assignment evaluation optimizes resource utilization, improves collaboration, and elevates the educational experience for all stakeholders involved.

1.3 Purpose of the thesis

The purpose of this thesis is to develop a specialized automation script for efficient evaluation of .Net Core programming assignments. By addressing challenges in handling complex assignments, the script aims to provide a tailored solution for programming assessment, streamlining the evaluation process for instructors and promoting faster feedback for programmers. The integration of a Web application enhances usability and customization, allowing instructors to easily adapt the automation script to their specific evaluation criteria. Additionally, compatibility testing ensures seamless integration with the .Net Core environment and databases, ensuring the automation script's versatility across various programming setups. The thesis ultimately seeks to deliver an optimized and comprehensive automation solution that enhances the evaluation experience for both instructors and programmers in the industry.

2. OVERVIEW OF RELEVANT STUDIES

2.1 Overview of existing methods

In today's rapidly evolving technological landscape, programming assignments play a crucial role in assessing programmers' understanding of coding concepts and problem-solving skills. As the number of programming assignments grows, educators and institutions face the challenge of efficiently evaluating and providing timely feedback to programmers. To address this, automation tools and scripts have become indispensable aids in streamlining the evaluation process for programming assignments. These tools offer a wide range of features that enhance grading consistency, reduce human bias, and enable faster assessment, thereby benefiting both programmers and instructors.

i. **Automated Test Execution and Grading Tools:** Automated test execution and grading tools are fundamental components of programming assignment evaluation. These tools facilitate the automatic execution of predefined test cases against programmer submissions and provide immediate feedback. Instructors can define test cases to cover various aspects of the assignment requirements and grading rubrics. Popular tools in this category include JUnit for Java, Pytest for Python, and Mocha for JavaScript. Such tools ensure that each programmer's code is thoroughly evaluated against a standard set of test cases, enabling fair and objective grading.

ii. **Code Quality Analysis and Static Code Analysis Tools:** Code quality analysis tools focus on assessing the overall quality of programmers' code and adherence to coding best

practices. Static code analysis tools play a vital role in identifying potential bugs, code smells, and vulnerabilities. These tools automatically analyze the codebase and provide valuable insights to both programmers and instructors. Examples of widely used tools include SonarQube, ESLint, and FindBugs. By integrating these tools into the evaluation process, instructors can guide programmers to write cleaner and more maintainable code.

iii. **Online Integrated Development Environments (IDEs) and Autograders:** Online Integrated Development Environments (IDEs) are cloud-based platforms that allow programmers to write, compile, and test their code directly in a web browser without the need for local installations. Within these IDEs, instructors can incorporate autograders, which provide real-time feedback as programmers write their code. This instant feedback helps programmers identify and rectify errors early in the development process. Repl.it, Code anywhere, and Cloud9 are popular examples of IDEs with built-in auto grading capabilities.

2.2 Review of Relevant Literature on Automation Techniques

In this subsection, we delve into automation techniques commonly used for evaluating programming assignments in educational settings. We explore automated test execution tools, such as JUnit, Pytest, and Mocha, which enable instructors to execute predefined test cases against programmers' code submissions and provide immediate feedback. Additionally, we examine code quality analysis tools like SonarQube and ESLint, which identify potential bugs, code smells, and vulnerabilities in the codebase. Understanding these automation techniques will inform the development of our specialized automation script for evaluating .Net Core programming assignments, ensuring efficient and accurate assessment.

Best Practices in Automation Script Design:

In this section, we investigate the best practices followed in designing automation scripts for programming assignment evaluation. We emphasize the importance of modular and reusable code structures to ensure maintainability and scalability. Providing comprehensive and constructive feedback to programmers is another key aspect that supports their learning and improvement. Moreover, we discussed the significance of version control systems in managing code submissions and tracking changes during the evaluation process. By incorporating these best practices, our automation script will be optimized to deliver effective evaluations for .Net Core projects (Andrew Troelsen, Philip Japikse. .NET Core 3: A Guide to the .NET Core Runtime).

By reviewing existing methods, tools, and literature in automation, as well as discussing solutions for working with zip files and executing .Net commands, this thesis aims to build upon established knowledge and propose.

3. PROBLEM STATEMENT

3.1 Identification of the problem

The identified problem is the manual extraction of zip files, which requires time-consuming and repetitive human intervention. The need for automation arises to streamline and speed up extracting information from these zip files efficiently. The objective is to develop an automation script and integrate it with a user-friendly web application, enabling seamless and accurate extraction of crucial data from .NET projects, such as database type, database name, and .NET version. The automation aims to eliminate the manual efforts involved in this task and provide a more efficient and reliable approach for extracting information from zip files.

3.2 Challenges and limitations associated with the problem

The problem of manually extracting zip files in the evaluation of programming assignments presents several challenges and limitations that necessitate the development of an automation script. This section delves into the key difficulties associated with manual extraction and highlights the drawbacks of not employing an automated solution.

1. **Time-Consuming and Repetitive Process:** Manually extracting files from numerous zip submissions is a time-consuming and repetitive process for instructors. As the number of assignments increases, the manual effort required escalates, leading to potential delays in providing timely feedback to programmers. This limitation affects the overall efficiency of the evaluation process, hindering instructors from focusing on other critical evaluation tasks.
2. **Prone to Human Errors:** The manual extraction of zip files leaves room for human errors, such as misplacing or overlooking files, especially when handling a large volume of submissions. Inconsistent file organization and accidental omission of important files may result in incomplete evaluations, leading to inaccurate feedback and unfair grading.
3. **Lack of Versatility and Compatibility:** Different assignments may have varying file structures and formats within the zip submissions. Manually handling these diverse formats can be challenging and time-consuming. Moreover, compatibility issues may arise when opening submissions in different Integrated Development Environments (IDEs), leading to inconsistencies in evaluations.

4. **Scalability Issues:** As the number of programmers and assignments grows, manual extraction becomes increasingly impractical. Instructors may struggle to manage the escalating workload, compromising the quality and promptness of feedback provided to programmers.

To address these challenges and limitations, the proposed automation script aims to streamline the extraction of zip files, providing a robust and efficient solution for evaluating programming assignments. Automation will alleviate the burden of manual extraction, improve accuracy, ensure consistency, and enhance the overall evaluation experience for both instructors and programmers. By identifying and mitigating these challenges, the automation script will optimize the evaluation process and enable instructors to deliver timely and constructive feedback, fostering a conducive learning environment for programmers.

To address the identified problem and overcome the associated challenges, this thesis aims to answer the following research questions and achieve the following objectives:

Questions:

1. How can the automation of zip file extraction be implemented effectively using Python?
2. How can .Net build and run commands be executed automatically and reliably within the automation process?
3. How can the automation solution accurately determine the success of the build process based on the command output?

Objectives:

1. Develop an automated Python solution to extract zip files and handle various file structures effectively.
2. Implement a mechanism to execute .Net build and run commands automatically within the automation process.
3. Capture and analyze the output of .Net build commands to accurately determine the success or failure of the build process.
4. Evaluate the effectiveness and efficiency of the automation solution through experimentation and performance analysis.
5. Recommend further improvements and future research in automating software development projects' extraction and build processes.

By addressing these research questions and achieving the stated objectives, this thesis aims to contribute to the automation of the extraction and build processes in software development, improving efficiency, reliability, and productivity in the development workflow.

4. METHODOLOGY

4.1 Description of the proposed approach

This section outlines the proposed approach to automate the extraction of zip files and execute .Net build commands using Python, aligning with the principles of the **Linear Testing Framework** and incorporating **Non-Functional Testing processes**. The automation strategy focuses on harnessing Python's versatile libraries and tools to streamline the evaluation process efficiently (Harish Bajaj, Technical Test Lead ECSIVS, Infosys 2018).

The automation process begins by identifying the target zip files and their respective paths. Python offers a rich set of functions and modules to interact with the file system, enabling seamless identification and selection of zip files for extraction. Following the Linear Testing Framework, the automation script systematically processes each zip file, ensuring an organized and orderly evaluation.

Next, the proposed approach integrates Non-Functional Testing processes to check the compatibility of the submission with the rubrics of the assignment. The automation script verifies crucial aspects like the version used, database type, and database name, ensuring adherence to the specified requirements. This compatibility testing enhances the evaluation's accuracy and consistency, promoting fair grading and an objective assessment of the submissions.

To execute .Net build commands automatically, the automation script utilizes Python's subprocess module. By leveraging this module, the script interacts with the system shell, executing .Net build commands within the Python environment. The output of the build process

is captured and analyzed to determine its success or failure. In case of errors or issues, the automation script handles them gracefully, ensuring a robust and reliable evaluation process. By adhering to the principles of the Linear Testing Framework and incorporating Non-Functional Testing processes, the proposed approach ensures a systematic and comprehensive evaluation of .Net Core programming assignments. The Python-based automation solution optimizes efficiency, accuracy, and scalability, empowering instructors to deliver timely and constructive feedback while fostering an enhanced learning experience for programmers.

4.2 Explanation of the selected libraries and tools

To automate the extraction of zip files and execute .Net build commands, specific libraries and tools are chosen to facilitate the automation process (Python 3.10 Documentation, The Python Package Index).

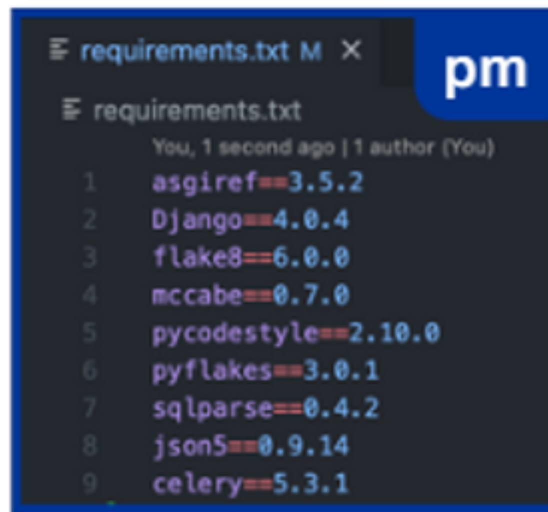

Figure 1. Requirements file

The shutil library in Python provides a comprehensive set of functions for working with files and directories. It includes functionalities for extracting zip files, copying files, and managing file-related operations. By utilizing the shutil library, the automation script can effectively remove the contents of the zip files and organize them as required.

The subprocess module in Python enables the execution of external commands, such as .Net build and dotnet run, from within the automation script. It allows the hand to interact with the command-line interface, execute the desired commands, and capture the output for further

analysis. The subprocess module provides flexibility and control over the execution of these commands, enabling seamless integration within the automation process.

4.3 Overview of the steps involved in the automation process

The automation process follows these steps:

Extracting zip files: The code searches for zip files in the specified directory and extracts their contents using the extract_zip() function.

Deleting extracted folders: The code removes the folders that were extracted from the zip archives but does not delete the corresponding zip files. This is achieved using the delete_extracted_folders() function.

Building projects: The code iterates through the extracted folders and executes the .Net build command for each project using the execute_dotnet_build() function. The build results are saved using the write_results() function.

Retrieving database information: The code reads the appsettings.json file in each project and extracts the database type and name using the get_database() function.

Extracting .NET version: The code parses the .csproj file in each project and retrieves the .NET version using the get_version() function.

Displaying and storing results: The code displays the build status, database type and name, and .NET version for each project. The results are stored in a list and can be further processed or analyzed.

These steps collectively automate the tasks described in the code using Python.

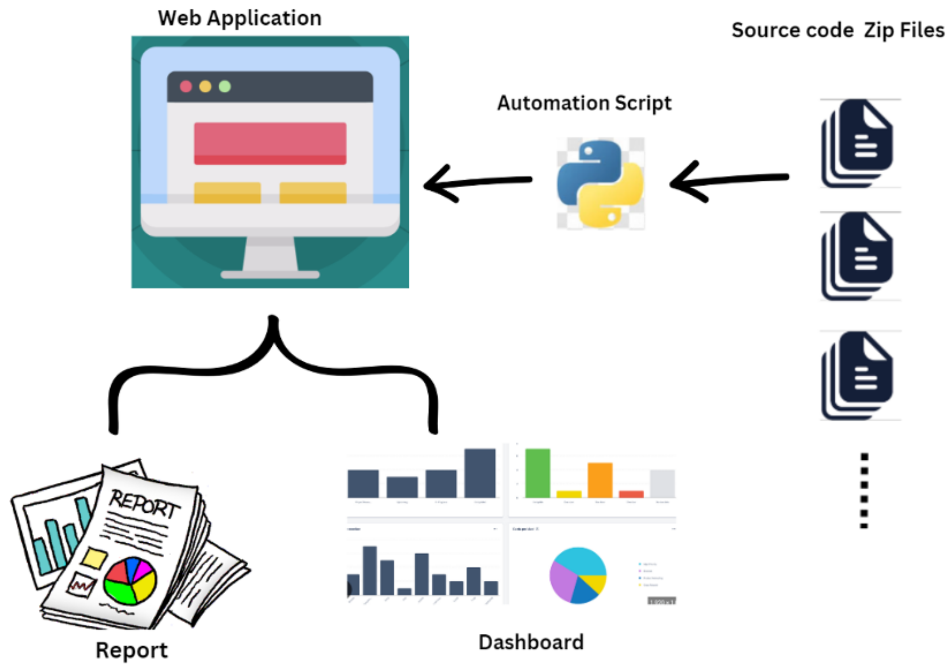

Figure 2. Overview of the implementation flow

5. IMPLEMENTATION

5.1 Resources and Tools Used in the Implementation

In this thesis, a range of essential tools were employed to support and expedite the research process.

Resources/Tool	Purpose
IDE - VS Code	Evaluation Environment (Style Guide for Python Code)
.Net Core	Corss-Platform .Net Framework (Microsoft .NET Documentation)
.Net CLI	Command-Line Interface for .Net
Python	Automation Scripting Language
Django	Web Framework
SqlServer, Sqlite	Databases Management System
GitHub:	Version Control and Collaboration Platform

Resources / Tools

These tools collectively empowered the thesis project and supported successful outcomes.

5.2 Python Automation Script: A brief overview

The *apply_automate_script(zip_file)* function is the entry point for the automation process. It takes the path of a .Net Core zip file as input and applies the automation script to that file. Here's an explanation of the flow:

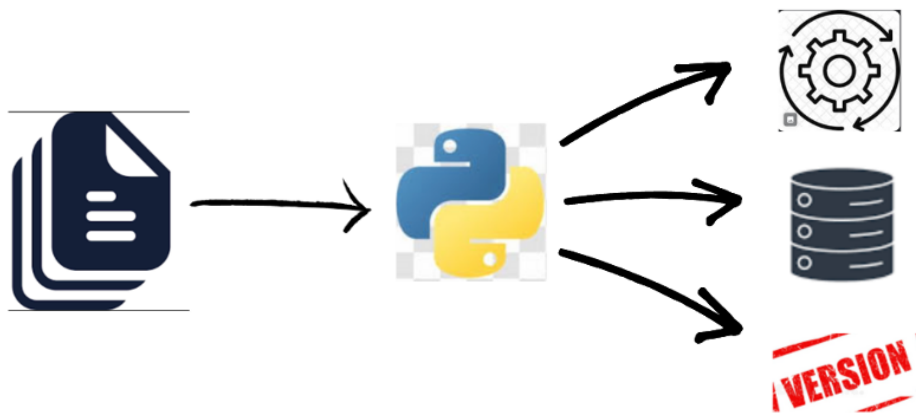

Figure 3. Applying Automation Script gives Build, Database, Version Info

apply_automate_script(zip_file):

```
def apply_automate_script(zip_file):
    zip_file = f"{extract_path}/{zip_file}"
    # delete_extracted_folders()
    folder_name = extract_zip(zip_file)
    return get_file_data(folder_name)
```

The function receives the zip_file parameter, which is the path of the zip file to be processed.

The function calls the *extract_zip(zip_file)* function to extract the contents of the zip file to a specified extraction path. This function uses the zipfile library to open and extract the zip file. It returns the name of the extracted folder.

```
def extract_zip(zip_file):
    extracted_folder = None
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        first_item = zip_ref.infolist()[0]
        extracted_folder = os.path.dirname(first_item.filename)
        zip_ref.extractall(extract_path)
    delete_mac_extract_folders()
    return extracted_folder
```

The function calls the *get_file_data(folder_name)* function, passing the extracted folder name as the parameter. This function retrieves the required data for the folder, including the build details, database information, and .NET version (Nate McMaster, Dustin Metzgar .NET Core in Action).

delete_mac_extract_folders():

```
def delete_mac_extract_folders():
    for entry in os.listdir(extract_path):
        folder_path = os.path.join(extract_path, entry)
        if os.path.isdir(folder_path) and entry == "__MACOSX":
            # Use shutil.rmtree() to delete the entire folder and its contents
            shutil.rmtree(folder_path)
```

The function delete_mac_extract_folders() is a method designed to remove unnecessary folders that may be created during the extraction process of a zip file. Specifically, this function targets the "__MACOSX" folder, which is commonly generated when zipping files on a Mac operating system.

The implementation of delete_mac_extract_folders() involves iterating over the entries in the extraction path, which is typically the folder where the contents of the zip file are extracted. For each entry, the function checks if it is a directory (folder) and if its name matches "__MACOSX".

If the conditions are met, the function utilizes the shutil.rmtree() function to delete the entire folder and its contents. shutil.rmtree() is a function from the shutil module in Python that recursively removes a directory tree. By passing the folder path to shutil.rmtree(), the function effectively deletes the "__MACOSX" folder and all its subdirectories and files.

The purpose of deleting the "__MACOSX" folder is to eliminate any unnecessary clutter and maintain a clean and organized extraction path. This folder is specific to macOS systems and may not contain relevant information for the automation process or subsequent analysis.

By implementing delete_mac_extract_folders() within the automation script, it ensures that any extracted zip files are free from extraneous "__MACOSX" folders, promoting a streamlined and efficient workflow.

In the context of the thesis, this method can be mentioned as part of the implementation details to demonstrate the consideration for maintaining a clean extraction path and eliminating unnecessary folders. It showcases the awareness of platform-specific artifacts and the steps taken to ensure data integrity and organization within the automation process.

get_file_data(folder_name):

```
def get_file_data(folder_name):
    result = {}
    build_details = execute_dotnet_build(folder_name)
    database = get_database(folder_name)
    version = get_version(folder_name)
    result = {
        "folder_name" : folder_name,
        "build" : build_details["build"],
        "error_details" : build_details["details"],
        "db_name" : database["db_name"],
        "db_type" : database["db_type"],
        "version" : version
    }
    return result
```

The *get_file_data(folder_name)* function performs the following steps:

1) **Extracting Build Information**: This may be accomplished in several phases, such as running the command for the specified source folder, handling errors and logging, and extracting error data to display the build result.

a) Executing the Command

It calls the execute_dotnet_build(folder_name) function to execute the .Net build command for the specified folder. This function runs the .Net build command using the subprocess library and captures the output. It determines whether the build was successful or not and retrieves the error details if any.

To execute the .Net build command, capture the console output, and determine the success or failure of the build process. Additionally, it includes functionality to extract error details, such as file name, line number, error code, and error message, in case the build fails.

To execute the command, the function utilizes the subprocess.Popen() method from the subprocess module. It specifies the command to be executed (.Net build) and sets the current working directory to the project path using the cwd parameter.

The console output is captured using the stdout=subprocess.PIPE parameter, which directs the output to a pipe. The output is then retrieved using the communicate()[0] method.

b) Exception Handling and Error Logging

In case of any exceptions during the execution of the .Net build command, such as command not found or invalid project path, the function handles the exception and returns a dictionary indicating a build failure, along with the error details.

The function then processes the console output, converting it to a string and replacing unnecessary characters. It saves the build results to a file using the write_results(content) function.

To determine the success or failure of the build process, the function searches for the phrase "Build succeeded." in the console output using a regular expression pattern. If the pattern is found, the function returns a dictionary indicating a successful build. Otherwise, it proceeds to extract error details using the get_error_details(build_output) function.

c) Extracting the Error Details

The get_error_details(build_output) function utilizes a regular expression pattern to extract error information from the build output. It searches for lines containing error

messages and captures details such as the file name, line number, error code, and error message.

The function keeps track of unique errors and counts the total number of errors encountered. If the phrase "Build Failed" is present in the build output, indicating a failed build, it appends the corresponding message.

Finally, the function returns the error details as a multiline string, which includes the extracted error information and the total error count.

By implementing this logic, the automation script effectively executes the .Net build command, captures the console output, and determines the success or failure of the build process. Additionally, it extracts error details in case of build failures, providing valuable information for further analysis or troubleshooting.

In the context of the thesis, this implementation detail can be highlighted to demonstrate the script's ability to execute and analyze the build process, capture console output, and extract error details. This functionality ensures accurate monitoring of the build process and facilitates effective identification and resolution of any encountered errors.

.NET cli command used to build Data

execute_dotnet_build(folder_name):

```
def execute_dotnet_build(project_path):
    cmd = ['dotnet','build']
    cwd = extract_path + f'/{project_path}/'
    try:
        output = subprocess.Popen(cmd, stdout=subprocess.PIPE,cwd=cwd).communicate()[0]
    except Exception as error:
        error_type = type(error).__name__  # Get the name of the error type
        error_message = str(error)  # Get the error message
        return {"build" : False, "details" : 'Failed'}

    content = str(output).replace('b\'','').replace('\'','').replace('\\n','\n')
    # Save the build results to the given file
    write_results(content)
    if content and re.search('Build succeeded.',content):
        return {"build" : True, "details" : ""}

    details = get_error_details(content)
    return {"build" : False, "details" : details}
```

```python
def get_error_details(build_output):

    error_pattern = r"([^/()]+\.cs)\((\d+),\d+\): error (CS\d+): (.+?) \[.+?\]"

    # Initialize error count and unique error messages
    error_count = 0
    unique_errors = set()
    error_details = []

    # Extract errors and update error count
    error_matches = re.findall(error_pattern, build_output)

    # Process error matches
    for match in error_matches:
        filename, line_number, error_code, error_message = match
        error = f"{filename} - {line_number} - {error_code} - {error_message}"
        if error not in unique_errors:
            error_details.append(
                f"\nFile: {filename}\nLine number: {line_number}\nMessage: {error_message}\nError Code: {error_code}\n\n"
            )
            unique_errors.add(error)
            error_count += 1

    # Check if "Build FAILED" line is present
    if "Build FAILED" in build_output:
        error_details.append("Build FAILED.\n")

    error_details.append(f"Total Errors: {error_count}\n")

    # Create a single multiline string
    detail_multiline_string = "".join(error_details)
    return detail_multiline_string
```

2) **Extracting Database information:** It calls the *get_database(folder_name)* function to retrieve the database information for the folder. This function checks if the appsettings.json file exists in the folder and reads the connection strings from it. It determines the database type (e.g., Sqlite or SqlServer) and extracts the database name.

   The get_database(project_path) function is responsible for retrieving the database type and database name from the appsettings.json file within the specified project path. The function takes the project path as a parameter, which represents the path to the extracted project folder.
   If the appsettings.json file exists, the function proceeds to read the connection string from the file using the read_connection_string() function. The connection string typically contains the necessary information to establish a connection with the database.

   a) **Identify the Database Type**

      To identify the database type, the function checks if the connection string value contains the ".sqlite" substring. If it does, the database type is identified as SQLite, and the function extracts the database name using a regular expression pattern specified in the get_database_name() function. The extracted database name and database type are then stored in the database dictionary.

b) Identify the Database Name

If the connection string does not contain the ".sqlite" substring, the function assumes the database type is SQL Server. Similarly, it extracts the database name using the get_database_name() function and assigns the values to the database dictionary.

The get_database_name(connection_string, pattern) function is a helper function that utilizes regular expressions to extract the database name from the connection string. It takes the connection string and an optional regular expression pattern as parameters. The default pattern is used for SQL Server databases, while a custom pattern is provided for SQLite databases. The function searches for a match based on the specified pattern and returns the extracted database name.

By implementing this logic, the automation script can effectively identify the appsettings.json file, extract the database type and name from the connection string, and provide this information for further processing or analysis. This ensures that the relevant database details are retrieved accurately and utilized appropriately within the Web application.

Finally, the function returns the database dictionary containing the extracted database name and database type.

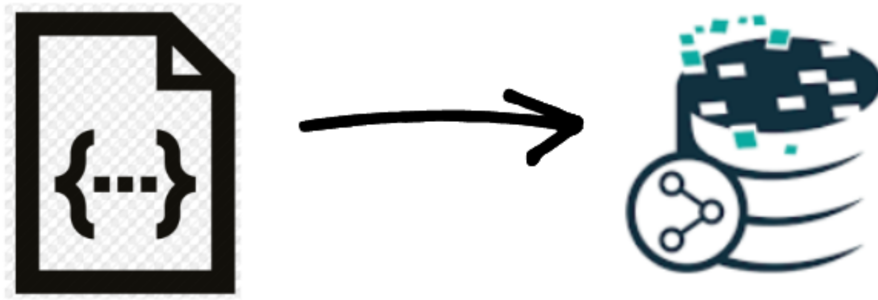

Figure 4. Reads database info from appsettings.json file.

get_database(folder_name):

```
def get_database(project_path):
    database = {'db_name' : '', 'db_type' : '' }
    settings_path = f"{extract_path}/{project_path}"
    is_file_existed = is_appsettings_json_existed(settings_path)
    if is_file_existed:
        connection_string = read_connection_string(f"{settings_path}/appsettings.json")
        if connection_string:
            for key, value in connection_string.items():
                # sqlite
                if isinstance(value, str) and '.sqlite' in value:
                    database = {
                                'db_name' : get_database_name(connection_string[key], r"Filename=(.*?)\.sqlite"),
                                'db_type' : 'Sqlite'
                            }
                # sqlserver
                else:
                    database = {
                            'db_name' : get_database_name(connection_string[key]),
                            'db_type' : 'SqlServer'
                        }
    return database
```

```
def get_database_name(connection_string, pattern = r"Database=(.*?);"):
    match = re.search(pattern, connection_string)
    if match:
        return match.group(1)
    else:
        return None
```

3) **Extracting Version Information:** It calls the *get_version(folder_name)* function to retrieve the .NET version used in the folder. This function searches for .csproj files in the folder, reads and parses them using the *xml.etree*.ElementTree library, and finds the "TargetFramework" element. It extracts the .NET version from the element.

The following explanation outlines the implementation details of the logic used to retrieve the version of .NET Core from the csproj file of a .NET application. This logic enables the automation script to identify and extract the .NET version used in the project. The get_version(project_path) function is responsible for retrieving the .NET version from the csproj file within the specified project path. The function takes the project path as a parameter, which represents the path to the extracted project folder.

To locate the csproj file, the function utilizes the os.walk() function to traverse the project directory and its subdirectories. It searches for files with the .csproj extension and creates a list of the file paths using the csproj_files variable.

Next, the function initializes the .Net_version variable to store the retrieved .NET version. It then iterates through each csproj file in the csproj_files list.

For each csproj file, the function reads and parses the XML structure using the ET.parse(csproj_file) method from the xml.etree.ElementTree module. This enables the script to navigate and extract information from the csproj file.

The function searches for the TargetFramework element within the csproj file using the XPath expression ./PropertyGroup/TargetFramework. If the element is found, the text content of the TargetFramework element is extracted, representing the .NET version used in the project.

Once a valid .NET version is found, the function assigns it to the .Net_version variable, prints it to the console for verification purposes, and breaks the loop to avoid unnecessary iterations.

By implementing this logic, the automation script can effectively extract the .NET version used in the project by examining the csproj file. This information can be utilized within the Web application for reporting, analysis, or any other necessary functionalities related to the .NET version.

In the context of the thesis, this implementation detail can be highlighted to showcase the script's ability to read and parse the csproj file, locate the TargetFramework element, and extract the .NET version accurately. This ensures that the correct version information is retrieved and utilized within the automation process, contributing to the overall functionality and effectiveness of the solution.

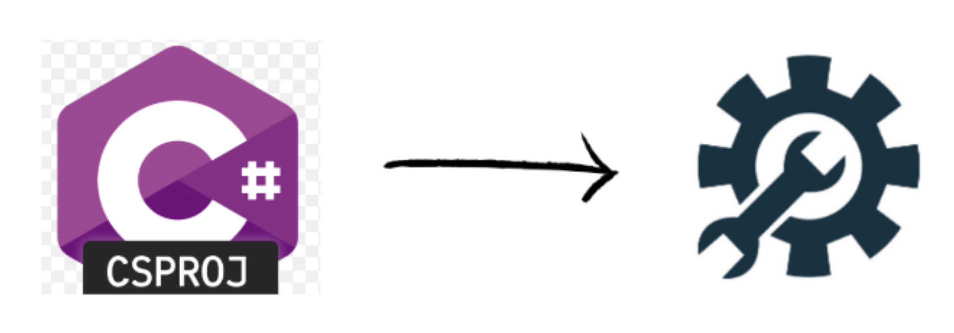

Figure 5. Reads version info from application's csproj file.

get_version(folder_name):

```python
def get_version(project_path):
    csproj_files = []
    for root, dirs, files in os.walk(f"{extract_path}/{project_path}"):
        for file in files:
            if file.endswith(".csproj"):
                csproj_files.append(os.path.join(root, file))

    dotnet_version = ''
    for csproj_file in csproj_files:
        # Read and parse the .csproj file
        tree = ET.parse(csproj_file)
        root = tree.getroot()

        # Find the TargetFramework element
        target_framework_element = root.find("./PropertyGroup/TargetFramework")
        if target_framework_element is not None:
            # Extract the .NET version from the TargetFramework element
            dotnet_version = target_framework_element.text
            # Print the version and break the loop if a valid version is found
            print("The .NET version is:", dotnet_version)
            break

    return dotnet_version
```

4) **Collecting the Information:** It creates a dictionary containing the folder name, build status, error details, database name, database type, and .NET version.

   The ***get_file_data(folder_name)*** function returns the dictionary containing the extracted data. The ***apply_automate_script(zip_file)*** function returns the dictionary received from the get_file_data(folder_name) function.

   This function serves as a bridge between the Web application and the automation script. In the Web application, you can call this function by passing the path of a zip file as a parameter. It will then apply the automation script to the zip file, extract the necessary information, and return it as a dictionary. The Web application can use this information to display the dashboard, generate reports, or perform any other required actions.

   The flow of the automation process ensures that the zip file is extracted, and the necessary data is retrieved using the .Net build command, appsettings.json file, and .csproj files. The extracted data can be further utilized according to the requirements of the Web application.

   Please note that the code provided earlier in the conversation might need modifications or adaptations to fit seamlessly into the Web application architecture.

5.3 Web Application Implementation

The implementation of the Web application plays a crucial role in facilitating the seamless integration of user inputs, file processing, and interaction with the automation script. This section provides a detailed explanation of the development process and key components involved.

The Web application is built using the Web Application framework, which provides a robust foundation for developing web applications. The application incorporates a user interface that allows users to upload files and initiate the automation process. To ensure data integrity and consistency, the application performs thorough input validation to ensure that the uploaded files adhere to the required naming convention (Web Software Foundation, Holovaty, Adrian; Kaplan-Moss, Jacob. The Definitive Guide to Django: Web Development Done Right).

Upon receiving a file upload, the application processes the uploaded file and extracts relevant information, such as usernames and assignment numbers, from the filenames. This information is crucial for associating the uploaded files with specific users and assignments.

Figure 6. Web UI helps to get reports and Dashboard.

The application interacts with the automation script by invoking the apply_automate_script(zip_file) function, which serves as the entry point for the automation process. The path of the uploaded zip file is passed as a parameter to this function, triggering the extraction of the zip file contents and subsequent application of the automation script.

To manage the extracted data effectively, the application utilizes Web's built-in database models. These models define the structure and relationships of the data, allowing for seamless storage and retrieval of information. The extracted data, including database type, database name, .NET version, and build status, is stored in the database for further analysis and reporting purposes.

5.4 Report and Dashboard Implementation

The report and dashboard components of the Web application play a vital role in providing users with meaningful insights into the processed data. This section delves into the implementation details of the report generation and dashboard development, highlighting their functionalities and how they leverage the extracted information.

The report generation functionality is designed to provide a comprehensive overview of the build process for each uploaded file. By utilizing the stored data, the application generates reports based on usernames and assignment numbers. These reports indicate whether the build process was successful or encountered errors, allowing users to quickly identify any issues.

The implementation of the report generation functionality involves querying the database for relevant data based on user inputs, such as usernames or assignment numbers. The application then compiles this data into a well-structured report format, which can be presented in various formats, such as PDF or HTML, depending on the requirements.

Furthermore, the Web application incorporates a dynamic and interactive dashboard to visualize key statistics and trends derived from the processed data. The dashboard provides a user-friendly interface that displays counts of different database types used, the distribution of .NET versions, and the overall success rate of the build process.

The dashboard is designed to be interactive, enabling users to filter and drill down into specific data subsets based on their preferences. This enhances the user experience by providing a customizable and tailored view of the processed data.

To implement the dashboard, the application leverages visualization libraries such as Matplotlib or Plotly, which enable the creation of insightful charts and graphs. These visual representations effectively convey the processed data, allowing users to grasp important patterns and make informed decisions based on the displayed information.

In conclusion, the report generation and dashboard implementation within the Web application offer valuable insights into the processed data. The reporting functionality enables users to obtain detailed information on the build process, while the dashboard provides a visually appealing and interactive platform to analyze key statistics and trends. Together, these components enhance the usability and effectiveness of the Web application, empowering users to make informed decisions based on the processed data.

6. RESULTS AND EVALUATION

6.1 Experimental Setup and Test Files

To evaluate the effectiveness of the automation script, a meticulously designed experimental setup was established. The setup aimed to cover various scenarios and test the script's capabilities comprehensively. A diverse range of test files was prepared, including sample files representing SQL Server and SQLite databases, as well as files intentionally modified to include build errors.

The test files were carefully crafted to align with the required naming convention. Each file name consisted of the respective usernames and assignment numbers, ensuring compatibility with the automation script's logic. Multiple versions of .NET solutions were included in the test files, enabling a thorough assessment of the script's functionality across different .NET environments. The experimental setup was carried out in a controlled environment, with standardized hardware and software configurations to ensure consistent and reliable results. The automation script was executed on each test file individually, capturing and analyzing the outputs for further evaluation.

6.2 Results and Analysis

The execution of the automation script on the test files yielded a comprehensive set of results, providing valuable insights into the build process and the extracted information. The analysis of these results facilitated a deeper understanding of the automation script's performance and its ability to retrieve crucial data from the files.

The list of zip file information served as an overview of the processed files, presenting key details such as usernames, assignment numbers, build status (success or failure), database type, database name, and .NET version. This consolidated information enabled a quick assessment of the files and their associated properties.

The generated reports offered a detailed breakdown of the build process for each file. The reports indicated whether the build was successful or encountered errors, enabling users to promptly identify any issues. In the case of build failures, the reports provided specific error details, including file names, line numbers, error codes, and error messages. This level of granularity empowered users to pinpoint and address the root causes of build failures efficiently.

Figure 7. Test Suites Evaluation Report

The dashboard, a visual representation of statistical insights derived from the processed data, provided a holistic view of the automation script's performance. It presented metrics such as the counts of different database types used, the distribution of .NET versions, and the overall success rate of the build process. The dashboard's intuitive visuals facilitated data interpretation and decision-making, allowing users to identify trends, patterns, and areas that required attention.

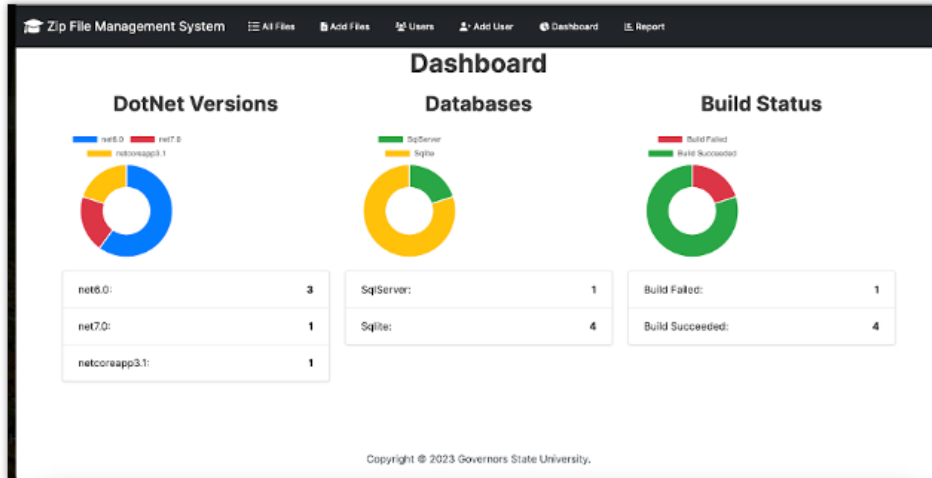

Figure 8. Admin Dashboard

6.3 Evaluation of Effectiveness and Efficiency

The evaluation of the automation process emphasized its effectiveness and efficiency in several aspects. Firstly, by eliminating the need for manual extraction of zip files and manual inspection of code in an Integrated Development Environment (IDE), the automation script significantly reduced manual effort and minimized the potential for human error. The automated approach ensured consistent and accurate results, enhancing the reliability of the information extracted from the files.

Furthermore, the automation script demonstrated efficiency in retrieving the required details from the files, such as database type, database name, and .NET version. By leveraging the inherent structure and organization of the files, the script efficiently extracted the necessary information, reducing processing time and optimizing the overall efficiency of the automation process.

Moreover, the automation process offered ease of use and improved productivity. It provided a streamlined workflow, eliminating the need for additional tools or complex setups. Unlike existing methods or tools like Jenkins, Docker, or Travis CI, which require installation, configuration, and potentially premium subscriptions, the automation script offered a lightweight and user-friendly solution. It facilitated a seamless integration with the Web application, allowing users to effortlessly upload files, extract information, and generate reports without the need for extensive technical knowledge.

6.4 Comparison with Existing Methods and Tools

In comparison to existing methods and tools, the automation script exhibited several advantages. Unlike tools such as Jenkins or Docker, which require setup and configuration, the automation script offered a straightforward approach. It eliminated the need for manual intervention and complex setups, streamlining the process of extracting vital information from the files. Furthermore, the automation script provided a cost-effective solution, as it did not require any premium subscriptions or licenses. In contrast, some existing tools may have associated costs, particularly when considering enterprise-level usage or advanced features. The automation script, being based on open-source Python code, offered a freely available and customizable solution to meet specific requirements.

Compared to writing customized YAML files for tools like Jenkins or Travis CI, the automation script offered greater flexibility and ease of modification. As the script was written in Python, a widely adopted and versatile programming language, it was easy to understand and modify. This flexibility allowed for rapid improvements, adjustments, and customizations based on evolving project needs, without the need to navigate through complex configuration files.

In conclusion, the automation script proved to be an effective and efficient solution for extracting vital information from .NET projects. It offered a streamlined workflow, ease of use, and improved productivity, without the need for complex setups or additional tools. Moreover, the script provided a cost-effective and customizable alternative to existing methods and tools, empowering users to gather and analyze essential data with ease. The automation process demonstrated its effectiveness and efficiency, offering significant benefits in terms of time

savings, accuracy, and overall productivity compared to traditional manual approaches or complex tools.

7. DISCUSSION

7.1 Interpretation and Analysis of the Results

The results obtained from running the automation script on the test files provide valuable insights into the build process, database information, and .NET versions used in the projects. The interpretation and analysis of these results shed light on the effectiveness and accuracy of the automation script in extracting relevant information and generating reports and dashboards.

The list of zip file information offers a comprehensive overview of the processed files. By extracting usernames, assignment numbers, build status, database type, database name, and .NET version, the script provides a clear snapshot of the projects. This information enables users to quickly identify the characteristics and properties of each file, facilitating efficient tracking and management.

The generated reports play a crucial role in understanding the build process for each file. By indicating whether the build was successful or encountered errors, the reports help users assess the overall quality of the projects. The inclusion of specific error details, such as file names, line numbers, error codes, and error messages, provides actionable insights for resolving build failures promptly. This detailed analysis allows users to address potential issues effectively and optimize the build process.

The dashboard, presenting statistical insights derived from the processed data, offers a comprehensive view of the project's characteristics. The number of different database types used, and the distribution of .NET versions provide valuable information about the technology landscape. Also, the overall success rate of the build process assesses the projects' quality and

reliability. The visual representations in the dashboard facilitate quick understanding and enable users to make informed decisions based on the presented data.

7.2 Discussion of the Strengths and Limitations

The developed automation script possesses several strengths that contribute to its effectiveness and usability. Firstly, it eliminates the need for manual extraction of zip files and manual inspection of code, streamlining the process and reducing manual effort. The script automates the extraction of relevant information, providing consistent and accurate results.
The script's flexibility allows it to handle various scenarios and versions of .NET solutions, (Mark J. Price. C# 9 and .NET 5 – Modern Cross-Platform Development) accommodating different project structures and configurations. By leveraging Python's capabilities, the script can be easily customized and enhanced to meet specific requirements. This adaptability ensures the script's longevity and scalability as new versions of .NET and evolving project structures emerge.
Moreover, the automation script provides a lightweight and user-friendly solution, avoiding the complexities associated with installing and configuring additional tools or frameworks. Its integration within the Web application enables a seamless workflow, empowering users to upload files, extract information, and generate reports effortlessly.
However, there are certain limitations to consider. The script relies on specific naming conventions for the uploaded files, which may require users to adhere to a predefined structure. Any deviation from the naming convention may result in inaccurate or incomplete information extraction. Additionally, the script's reliance on specific file structures and conventions may limit its applicability to projects that deviate significantly from the standard practices.
Furthermore, the script currently focuses on extracting database information and .NET versions, with limited support for other project aspects. While the reports provided and dashboard offer

valuable insights, they may not cover all aspects that stakeholders may be interested in, such as code quality metrics or performance analysis. Expanding the scope of the script to incorporate additional analysis and reporting capabilities would enhance its utility and value.

7.3 Potential Improvements and Future Work

To improve the automation script, several potential enhancements can be considered. Firstly, refining the error detection and reporting mechanism would provide more detailed and actionable insights into build failures. The script can be enhanced to capture and analyze additional error information, such as stack traces or specific error codes, facilitating more effective debugging and resolution.

Additionally, incorporating advanced analytics techniques could enable the script to generate more comprehensive reports and insights. Implementing code quality metrics, performance analysis, or security checks would offer a deeper understanding of the projects' overall health and adherence to best practices.

Furthermore, expanding the script's compatibility to other frameworks, such as Flask or Spring Boot, would widen its applicability and cater to a broader range of projects. This expansion would require adapting the script to handle different project structures, configuration files, and build processes specific to each framework.

In terms of performance and efficiency, optimizing resource consumption and handling port conflicts during the script's execution could be addressed. Ensuring efficient resource utilization and implementing strategies to handle potential port conflicts would enhance the overall reliability and scalability of the automation process.

In conclusion, the automation script exhibits strengths in its flexibility, ease of use, and integration within the Web application. However, certain limitations related to naming conventions and the scope of analysis should be considered. By implementing potential improvements, such as refining error reporting, incorporating advanced analytics, expanding

compatibility with other frameworks, and optimizing resource consumption, the script's effectiveness and utility can be further enhanced. These improvements and future work contribute to the ongoing evolution and refinement of the automation process, ensuring its relevance and value in the rapidly evolving landscape of .NET development.

8. CONCLUSION

8.1 Summary of the Thesis Objectives and Contributions

The objective of this thesis was to develop an automation script for extracting essential information from .NET projects and integrating it into a Web application. The automation script aimed to streamline the process of extracting database type, database name, .NET version, and build status from zip files, generating reports, and presenting key metrics on a dashboard.
The developed automation script successfully achieved these objectives and made valuable contributions to the field of software development and automation. By leveraging Python's capabilities, the script provided a user-friendly and efficient solution for extracting information from .NET projects. Its integration with the Web application allowed users to upload zip files, extract data, and generate reports effortlessly.

8.2 Recapitulation of the Key Findings and Insights

Through the execution of the automation script on test files, several key findings and insights were gained. The script effectively extracted information such as database type, database name, .NET version, and build status from the zip files, providing a comprehensive overview of the projects. The generated reports offered a detailed breakdown of the build process and error details, facilitating prompt resolution of build failures. The dashboard presented statistical insights, enabling users to analyze the distribution of database types, .NET versions, and overall success rates.

The analysis of the results demonstrated the effectiveness and efficiency of the automation process. By eliminating manual extraction and inspection of code, the script reduced human effort and potential errors. It provided accurate and consistent results, enhancing productivity and decision-making. Furthermore, the automation script highlighted several advantages over existing methods and tools by offering simplicity, cost-effectiveness, and flexibility.

8.3 Final Remarks and Recommendations for Future Automation Efforts

In conclusion, the developed automation script for extracting information from .NET projects and integrating it into a Web application proved to be a valuable tool for streamlining processes and improving productivity. It highlighted the power of automation in reducing manual effort with the added integration of technologies like Celery, (Celery Documentation) and providing accurate insights.

For future automation efforts, several recommendations can be made. Firstly, expanding the capabilities of the automation script to cover additional aspects, such as code quality metrics or performance analysis, would provide a more comprehensive evaluation of the projects. Incorporating advanced analytics techniques could offer deeper insights into the projects' health and adherence to best practices.

Additionally, optimizing resource consumption and handling port conflicts during the execution of the automation script would improve its performance and scalability. Ensuring efficient resource utilization and implementing strategies to handle potential conflicts would contribute to a smoother automation process (Jithin Alex, Network Automation using Python).

In conclusion, the developed automation script provided a valuable contribution to software development and automation. Its integration with the Web application facilitated the extraction of essential information from .NET projects, enabling efficient analysis and decision-making. The findings and insights gained from this research open possibilities for further enhancements and future automation efforts in the realm of .NET development and beyond.

9. REFERENCES

Python 3.10 Documentation (n.d.). os, zipfile, subprocess, json5, shutil & xml.etree.ElementTree modules. Available at: https://docs.python.org/3.10/library/

Web Software Foundation. Available at: https://docs.djangoproject.com/en/4.0/

Style Guide for Python Code. PEP 8. Available at: https://www.python.org/dev/peps/pep-0008/

Microsoft .NET Documentation. Available at: https://learn.microsoft.com/en-us/dotnet/

Celery Documentation. Available at: https://docs.celeryproject.org/en/stable/

The Python Package Index. PyPI. Available at: https://pypi.org/

Harish Bajaj, Technical Test Lead ECSIVS, Infosys 2018. Choosing the Right Automation Testing Framework for Your Project. Available at: https://www.infosys.com/services/it-services/white-papers/documents/choosing-right-automation-tool.pdf

Al Sweigart, Automate the Boring Stuff with Python. Available at: https://automatetheboringstuff.com/

Jithin Alex, Network Automation using Python 3, Available at: https://www.amazon.com/Network-Automation-using-Python-Administrators-ebook/dp/B084GFJB41

Holovaty, Adrian; Kaplan-Moss, Jacob. The Definitive Guide to Django: Web Development Done Right. Available at: https://link.springer.com/book/10.1007/978-1-4302-1937-8

Andrew Troelsen, Philip Japikse. .NET Core 3: A Guide to the .NET Core Runtime - Available at: https://books.google.com/books?id=AvLuCgAAQBAJ&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

Mark J. Price. C# 9 and .NET 5 – Modern Cross-Platform Development Available at: https://www.amazon.com/NET-Cross-Platform-Development-intelligent-Framework-ebook/dp/B08KQK22LJ

Nate McMaster, Dustin Metzgar .NET Core in Action - Available at: https://books.google.com/books?id=xzczEAAAQBAJ&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false