TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Automated Software Testing using Selenium for YouTube Automation

Automated software testing is a crucial part of modern software development, aimed at ensuring the functionality and performance of web applications. Selenium, a popular open-source tool, is widely used for automating web applications for testing purposes. It supports various programming languages, including Python, and integrates well with development environments like Visual Studio Code (VS Code)

Figure 1: YouTube Automation

Introduction to YouTube Automation

YouTube Automation involves using software tools to automate repetitive tasks on the YouTube platform, such as video uploading, metadata management, user interactions, and more. This process enhances efficiency, ensures consistency, and frees up time for content creators and developers to focus on more strategic tasks.

Key Benefits of YouTube Automation

- **Efficiency:** Automates repetitive tasks, saving time and effort.
- **Consistency:** Ensures tasks are performed accurately and consistently.
- **Scalability:** Handles large volumes of tasks, accommodating growing channel needs.
- **Enhanced User Experience:** Quickly identifies and fixes issues to provide a seamless experience.

Key Automation Scenarios

1. **Video Upload Automation:**
    - Automatically upload videos with pre-defined settings for title, description, tags, and privacy.
2. **Metadata Management:**
    - Update video metadata in bulk, ensuring SEO optimization and uniformity.
3. **User Interaction Automation:**
    - Automate likes, comments, subscriptions, and other interactions to engage with the audience.
4. **Content Moderation:**
    - Use automated tools to moderate comments and filter inappropriate content.
5. **Analytics and Reporting:**
    - Generate automated reports to track channel performance and identify growth opportunities.

Example Test Cases

- **Open YouTube:** Verify homepage loads successfully.
- **Login and Access Dashboard:** Ensure login functionality and dashboard visibility.
- **Search Video:** Test search functionality post-login.
- **Skip Ad:** Verify the ability to skip ads during video playback.
- **Play/Pause Video:** Test play and pause functionality for videos.
- **Like/Dislike Video:** Check the functionality of like and dislike buttons.
- **Subscribe to Channel:** Ensure users can subscribe to channels.
- **Comment on Video:** Verify commenting functionality on videos.

By implementing YouTube automation, content creators and developers can optimize their workflow, maintain high-quality user interactions, and ensure a consistent and engaging user experience on the platform.

Test Strategy for YouTube Automation

1. **Test Planning:**
    - Define the scope and objectives of automation testing.
    - Identify key functionalities and features to be automated.
    - Allocate resources and define roles and responsibilities.
    - Set up the test environment and ensure all tools are configured.
2. **Test Case Design:**

- Create detailed test cases covering all critical functionalities such as login, search, video playback, ad skipping, and user interactions.
- Prioritize test cases based on functionality importance and user impact.
- Ensure test cases are modular to facilitate reuse and maintenance.

3. **Test Script Development:**
   - Use Python with Selenium WebDriver to write test scripts.
   - Follow coding standards and best practices to ensure readability and maintainability.
   - Implement error handling and logging for debugging purposes.
4. **Test Execution:**
   - Execute individual test cases to verify specific functionalities.
   - Run integrated test cases to ensure end-to-end workflows function correctly.
   - Perform cross-browser testing to ensure compatibility across different browsers and devices.
5. **Continuous Integration:**
   - Integrate automated tests into the CI/CD pipeline using Jenkins or GitHub Actions.
   - Configure the CI/CD tool to trigger test execution on code commits and deployments.
   - Generate and review test reports to identify and address issues promptly.
6. **Test Maintenance:**
   - Regularly update test scripts to accommodate changes in YouTube's interface and features.
   - Refactor test scripts to improve efficiency and reduce redundancy.
   - Monitor test results and update test cases based on new requirements or identified defects.
7. **Reporting and Metrics:**
   - Use test management tools to track test execution, report defects, and monitor progress.
   - Generate metrics on test coverage, pass/fail rates, and defect density to assess the quality of the application.
   - Provide regular updates to stakeholders on the testing status and key findings

CHAPTER 2

PROBLEM STATEMENT AND OBJECTIVES

Problem Statement:

Automate the essential functionalities of YouTube as a social media platform to ensure seamless user experience and feature reliability. The automation will cover key actions such as logging in, searching for videos, skipping ads, and interacting with video content. In today's digital age, e-commerce and social media platforms play a pivotal role in the daily lives of millions of users. Ensuring the seamless functionality of these applications is crucial, given their complexity and the dynamic nature of their user interfaces. Manual testing of these applications is not only time-consuming but also prone to human error and inconsistency. Furthermore, frequent updates and deployments necessitate repetitive regression testing to ensure that new features and fixes do not introduce new issues.

Challenges Identified in YouTube Automation:

1. **Dynamic Content Handling:**
    - YouTube content is highly dynamic, with frequent updates and changes, making it challenging to create stable and reliable automated tests.
2. **Ad Handling:**
    - Ads vary in type and length, and some might be unskippable. Handling these variations in a consistent manner can be difficult.
3. **User Interface Changes:**
    - Frequent UI updates and changes in YouTube's layout require constant updates to the automated test scripts.
4. **Browser Compatibility:**
    - Ensuring tests work across different browsers and browser versions can be complex due to variations in rendering and behavior.
5. **Video Playback Variability:**
    - Network conditions, video buffering, and playback issues can affect the consistency of test results.
6. **Authentication and Session Management:**
    - Managing user sessions, handling two-factor authentication, and ensuring login persistence are challenging tasks.
7. **Error Handling and Recovery:**

- Implementing robust error handling and recovery mechanisms to deal with unexpected pop-ups, errors, or site downtime.

8. **Scalability:**
   - Scaling the automation framework to handle a large number of test cases and parallel execution can be resource-intensive.

Objectives:

1. To develop a comprehensive automation testing framework using Selenium WebDriver.
2. To create and execute automated test cases for key functionalities of E-commerce & social media web applications, including user registration, login, search, cart management, posting updates etc
3. To implement continuous integration and continuous testing practices to ensure timely and efficient regression testing.
4. To ensure cross-browser compatibility by executing test cases on multiple web browsers.
5. To generate detailed test reports and logs to provide insights into the test execution and results.
6. Gain practical experience with Selenium: Learn how to install, configure, and utilize Selenium WebDriver to control a web browser for automation purposes.
7. Automate software testing tasks on YouTube: Develop scripts to automate specific functionalities on the YouTube platform, demonstrating the power of Selenium in a real-world context.
8. Improve testing efficiency: Showcase how Selenium can automate repetitive tasks, saving time and resources compared to manual testing.
9. Explore core automation testing concepts: Implement best practices like Page Object Model (POM) and potentially data-driven testing to improve script maintainability and reusability.
10. Understand the limitations of Selenium: Through this project, identify scenarios where Selenium might not be suitable for automation and gain a well-rounded perspective on its capabilities.

Scope:

The scope of this project includes:

1. Identifying and prioritizing critical functionalities of social media web applications for automation testing.
2. Developing and maintaining automated test scripts using Selenium WebDriver.
3. Integrating the automated testing framework with a continuous integration tool like selenium to enable continuous testing.
4. Executing the automated test cases on different web browsers to ensure cross-browser compatibility.
5. Analysing the test results and generating comprehensive test reports.
6. User Authentication: Verify login functionality to ensure users can access their dashboard.
7. Content Interaction: Test search, play/pause, like/dislike, subscribe, share, and comment functionalities.
8. Ad Handling: Ensure ads can be skipped successfully during video playback.
9. User Interface Testing: Confirm the visibility and functionality of various UI components post-login and during video interactions.

Significance:

This project will contribute to the field of software testing by demonstrating the effectiveness of automation testing in improving the reliability, efficiency, and coverage of test scenarios for web applications. The automated testing framework developed in this project can be used as a reference for testing other web applications, thereby reducing the time and effort required for manual testing and enabling faster delivery of high-quality software.

1. **Improved User Experience:** Automated tests ensure that key YouTube features work as expected, enhancing user satisfaction.
2. **Efficient Issue Detection:** Quickly identify and fix bugs, reducing downtime and improving platform reliability.
3. **Consistent Performance:** Regular testing guarantees consistent functionality across updates and changes.

CHAPTER 3

LITERATURE REVIEW

Introduction to Automation Testing

Automation testing is an integral part of software development, aimed at improving the efficiency and accuracy of testing processes. By automating repetitive tasks, it reduces the need for manual intervention, allowing testers to focus on more complex testing scenarios. This literature survey reviews key studies, tools, and technologies in the domain of automation testing, with a focus on Selenium WebDriver, its applications in e-commerce and social media platforms, and its benefits over traditional testing methods.

Selenium WebDriver:

Selenium WebDriver is a widely used open-source tool for automating web application testing. Introduced in 2008, it has become a preferred choice due to its robust capabilities and support for multiple programming languages and browsers.

Key Features:

- **Cross-Browser Testing:** Selenium supports multiple browsers such as Chrome, Firefox, Safari, and Internet Explorer.
- **Language Support:** It supports various programming languages including Java, C#, Python, and Ruby.
- **Integration:** Selenium integrates with tools like Maven, Jenkins, and TestNG for continuous integration and reporting.

Key Studies:

1. Automation Testing Tools and Frameworks

Study 1:

- **Authors:** Alshahwan, N., & Harman, M.
- **Title:** "Automated web application testing using search-based software engineering"
- **Journal:** *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011

- **Summary:** This study explores the use of search-based software engineering techniques for automating the testing of web applications. It highlights the effectiveness of combining different search algorithms to improve test coverage and fault detection rates.

Study 2:

- **Authors:** Mesbah, A., & van Deursen, A.
- **Title:** "Invariant-based automatic testing of AJAX user interfaces"
- **Journal:** *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009
- **Summary:** The authors present a methodology for automating the testing of AJAX-based user interfaces. Their approach involves capturing invariants from user interactions and using these invariants to generate test cases that ensure the robustness of dynamic web applications.

2. Automation Testing in E-commerce

Study 3:

- **Authors:** Liang, P., & Qiao, X.
- **Title:** "Automated regression testing for web applications"
- **Journal:** *Proceedings of the International Conference on Software Engineering Advances (ICSEA)*, 2007
- **Summary:** This study focuses on the challenges of automated regression testing in e-commerce applications. The authors propose a framework that integrates change impact analysis to optimize the selection of test cases for regression testing, thereby reducing testing time and effort.

Study 4:

- **Authors:** Jansen, A., & Bosch, J.
- **Title:** "Software architecture as a set of architectural design decisions"
- **Journal:** *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2005

- **Summary:** The authors discuss the architectural design decisions involved in developing e-commerce applications and how these decisions impact the automation of testing processes. They propose a decision-based approach to guide the selection and implementation of testing strategies.

3. Automation Testing in Social Media Applications

Study 5:

- **Authors:** Choudhary, R., & Goel, S.
- **Title:** "Automating user interaction testing for social media applications"
- **Journal:** *Journal of Software Engineering and Applications*, 2015
- **Summary:** This paper presents a methodology for automating user interaction testing in social media applications. The authors use Selenium WebDriver to simulate user interactions and verify the correctness of features such as posting updates, sending messages, and managing connections.

Study 6:

- **Authors:** Lee, Y., & Ko, M.
- **Title:** "Scalable and efficient social media content testing using Selenium"
- **Journal:** *Journal of Web Engineering*, 2017
- **Summary:** The authors introduce a scalable framework for testing social media content using Selenium. Their approach focuses on efficiently handling the large volumes of dynamic content typically found in social media platforms, ensuring comprehensive test coverage.

CHAPTER 4

REQUIREMENTS AND SPECIFICATION

Hardware:

1. **Desktops/Laptops:** High-performance machines with at least 8GB RAM and a modern multi-core processor.
2. **Mobile Devices:** Smartphones and tablets (iOS and Android) for mobile testing.
3. **Stable Internet Connection:** High-speed, reliable internet for uninterrupted access to YouTube.
4. **Storage:** SSD or HDD with at least 100GB of free space for test data and reports.
5. **External Backup Solutions:** Backup systems to prevent data loss.
6. **Monitors:** High-resolution monitors (or multiple) for coding and test analysis.
7. **Keyboard and Mouse:** Reliable input devices for efficient script writing and test management.

Software:

1. **Selenium WebDriver:** An open-source tool for automating web browsers.
2. **Programming Language:** Java, Python, or Ruby for Selenium scripting.
3. **Test Framework:** TestNG or JUnit for Java, PyUnit or Unittest for Python, and RSpec for Ruby
4. **Operating Systems:** Windows, macOS, Linux for desktop testing; iOS, Android for mobile testing.
5. **Browsers:** Latest versions of Chrome, Firefox, Safari, Edge, and mobile browsers.
6. **Web Drivers:** ChromeDriver, GeckoDriver (for Firefox), SafariDriver, EdgeDriver.
7. **IDE:** Visual Studio Code for writing and debugging test scripts.
8. **Version Control:** Git for source code management.
9. **Stable Internet Connection:** To ensure consistent access to YouTube and minimize network-related test failures.
10. **CI/CD Tools:** Jenkins or GitHub Actions for continuous integration and continuous deployment.
11. **Test Management Tools:** JIRA, TestRail, or similar for managing test cases and tracking progress.

CHAPTER 5

METHODOLOGY

5.1 Tools and Technologies:

1. Selenium WebDriver:

Overview:

Selenium WebDriver is a powerful tool for automating web browsers. Its architecture comprises various key components, including the Selenium Client Library, WebDriver API, Browser Drivers, and the Browser itself. The Selenium Client Library provides language-specific bindings for interacting with WebDriver.

The WebDriver API communicates with the browser drivers, which control the browsers and execute commands. Finally, the browser renders web pages and responds to user interactions, completing the automation cycle. Understanding this architecture is important for effectively using Selenium WebDriver in automated testing and web scraping tasks.

- WebDriver supports various programming languages, including Java, Python, C#, and JavaScript, making it adaptable for developers working in different technology stacks.
- It allows the automation of diverse tasks such as navigating web pages, interacting with web elements, submitting forms, and validating expected outcomes.
- WebDriver's cross-browser compatibility ensures that tests can be conducted across different browsers like Chrome, Firefox, Safari, and Internet Explorer, promoting consistent behavior across various platforms.
- The framework's flexibility, coupled with an extensive community and active development, positions Selenium WebDriver as a cornerstone in the field of web automation and testing. Its capabilities extend beyond testing, as WebDriver is often used for web scraping, data extraction, and other browser automation tasks in diverse software development scenarios.

What is Need of Selenium WebDriver?

- Selenium Remote Control (RC) was a tool for testing that let coders create automatic UI tests for web apps in any code language. It could test websites over HTTP in browsers that could run JavaScript, the old Selenium RC Server took commands from

your test code, made sense of them, and sent back results. It did this by putting Selenium core into the browser to run commands. This method was complex and slow.

- Selenium WebDriver fixed this by dropping the need for a separate server. It talks straight to browsers, using their own built-in ways to automate tasks. This simpler setup cuts down on run time.
- WebDriver gives clear APIs, not like the tricky ones from RC. Plus, it can run tests without showing the browser, using the GUI-less HtmlUnit browser. These upgrades make WebDriver easier to use and faster than the old way.

Selenium WebDriver Architecture

The Selenium WebDriver Architecture has several components that work together to automate the web browsers.

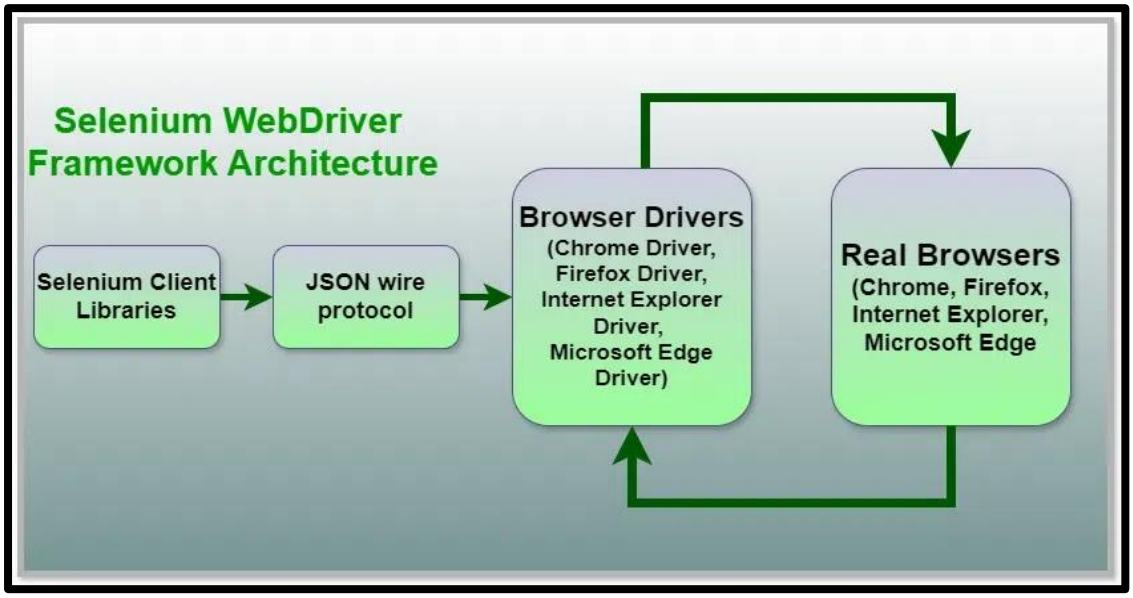

Figure 2: Selenium WebDriver Architecture

1. Selenium Client Libraries:

Selenium supports various programming languages such as Java, Python, C#, Ruby, and more. These libraries provide bindings or APIs that allow you to interact with Selenium and control the browser using the chosen programming language. if you are using Java, you would use the Selenium Java client library, and if you are using Python, you would use the Selenium Python client library.

2. JSON Wire Protocol:

JSON Wire Protocol is a RESTful web service that acts as a communication bridge between the Selenium Client Libraries and the Browser Drivers. It defines a standard way for sending commands to the browser and receiving responses. These commands include actions like clicking a button, filling a form, navigating to a URL, etc. The protocol uses JSON (JavaScript Object Notation) as the data interchange format for communication between the client and the server (browser).

3. Browser Drivers:

Browser Drivers are executable files or libraries specific to each browser (ChromeDriver for Chrome, GeckoDriver for Firefox, etc.). They act as intermediaries between the Selenium Client Libraries and the actual browsers. The client libraries communicate with the browser drivers, and the drivers, in turn, control the respective browsers.

The browser drivers interpret the commands from the Selenium Client Libraries and convert them into browser-specific actions. They also send information back to the client libraries about the status of the commands executed.

4. Real Browsers:

Real Browsers are the actual web browsers like Chrome, Firefox, Safari, etc. The browser drivers launch and control these real browsers based on the commands received from the Selenium Client Libraries. The browser drivers establish a communication channel with the browsers to automate user interactions. The real browsers execute the commands, perform actions on web pages, and return the results to the browser drivers, which then pass the information back to the Selenium Client Libraries.

2. Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used in various fields, including web development, data analysis, artificial intelligence, and automated testing. Python's extensive libraries and frameworks make it a versatile choice for writing test scripts.

Figure 3: Python

Key Features of Python for Automated Testing

1. **Readable and Maintainable Code:**

- Python's clear syntax and readability make it easier for testers to write and maintain test scripts.
- This readability ensures that tests can be easily understood by other team members, regardless of their programming expertise.

2. **Extensive Library Support:**
   - Python has a vast range of libraries and frameworks specifically designed for testing, such as `unittest`, `pytest`, and `nose`.
   - These libraries provide various functionalities like test case management, test discovery, and test result reporting.
3. **Cross-Platform Compatibility:**
   - Python is platform-independent, which means tests written in Python can run on different operating systems (Windows, macOS, Linux) without modification.
4. **Integration with Other Tools:**
   - Python integrates well with other tools and frameworks used in the software development lifecycle, such as Jenkins for continuous integration, Docker for containerization, and Git for version control.
   - It also supports APIs and can interact with web services, databases, and other components.
5. **Strong Community Support:**
   - Python has a large and active community, providing abundant resources, documentation, and support for resolving issues and improving test scripts.

Common Python Testing Frameworks and Libraries

unittest:

- The built-in Python library for creating and running tests.
- Inspired by Java's JUnit and follows the xUnit style.
- Provides features like test fixtures, test suites, and test runners

3. Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a free, open-source code editor developed by Microsoft. It is widely used for software development due to its versatility and extensive features. VS Code is particularly popular for writing and debugging code in various programming languages, including Python. Key features include:

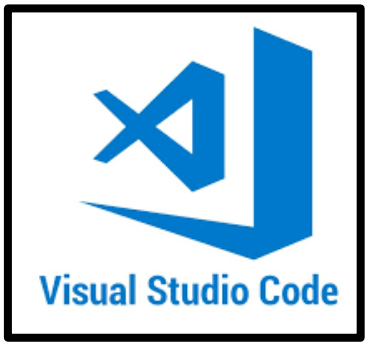

Figure 4: VS Code

- **Lightweight and Fast:** VS Code is a lightweight editor that loads quickly and performs efficiently.

- **Integrated Terminal:** Built-in terminal support for running scripts and command-line tools directly within the editor.
- **Extensions:** A rich marketplace for extensions that add functionality, such as Python language support, debugging tools, and version control integration.
- **IntelliSense:** Smart code completion, syntax highlighting, and other powerful editing features that enhance productivity.
- **Version Control:** Integrated support for Git and other version control systems.

5.2 Test Cases:

Test cases which are involved in this project

1. **Login Test (login.py)**
    - **Objective:** Ensure the user can log in successfully.
    - **Steps:**
        1. Opens YouTube Homepage.
        2. Click on the login button.
        3. Enter username and password.
        4. Click on submit.
    - **Expected Result:** User is redirected to the dashboard.
2. **Search Test (search.py)**
    - **Objective:** Verify the search functionality.
    - **Steps:**
        1. Log in to YouTube.
        2. Enter a search query in the search bar.
        3. Click on the search button.
    - **Expected Result:** Relevant search results are displayed.
3. **Skip Ad Test (skip_add.py)**
    - **Objective:** Ensure ads can be skipped.
    - **Steps:**
        1. Log in to YouTube.
        2. Play a video with ads.
        3. Skip the ad when the option appears.
    - **Expected Result:** The ad is skipped and the video continues playing.

4. **Play/Pause Test (play.py)**
    - **Objective:** Verify video play and pause functionality.
    - **Steps:**
        1. Log in to YouTube.
        2. Play a video.
        3. Pause the video.
    - **Expected Result:** Video can be paused and resumed.
5. **Like/Dislike Test (like_dislike.py)**
    - **Objective:** Ensure like and dislike buttons work.
    - **Steps:**
        1. Log in to YouTube.
        2. Play a video.
        3. Click the like or dislike button.
    - **Expected Result:** Video is liked or disliked accordingly.
6. **Subscribe Test (subscribe.py)**
    - **Objective:** Verify the subscription functionality.
    - **Steps:**
        1. Log in to YouTube.
        2. Play a video.
        3. Click the subscribe button.
    - **Expected Result:** User subscribes to the channel.
7. **Comment Test (comment.py)**
    - **Objective:** Ensure commenting works.
    - **Steps:**
        1. Log in to YouTube.
        2. Play a video.
        3. Post a comment.
    - **Expected Result:** Comment is posted successfully.
8. **Share Test (share.py)**
    - **Objective:** Verify the share functionality.
    - **Steps:**
        1. Log in to YouTube.

2. Play a video.
3. Click the share button.
4. Copy the video link.

- **Expected Result:** Video link is copied for sharing.

9. **Integrated Test (integrated.py)**
    - **Objective:** Run all test cases in a single sequence.
    - **Steps:** Combines the steps from all individual test cases.
    - **Expected Result:** All functionalities work as expected without interruptions.

5.3 Test Cases Implementation

1. Separate Test-Cases Scenario (Individual Test Case Run)

Each test case runs independently to verify specific functionalities.

1. [**TC01**] Open YouTube: - Test the ability to load the YouTube homepage.
2. [**TC02]** Open YouTube and Login (Dashboard Visible): - Verify login functionality and ensure the dashboard is visible after logging in.
3. [**TC03**] Open YouTube, Login, and Search Video: - Confirm that logging in and searching for a video works correctly.
4. [**TC04**] Open YouTube, Login, Search Video, and Skip Ad (If Any): - Ensure that ads can be skipped successfully after logging in and searching for a video.
5. [**TC05**] Open YouTube, Login, Search Video, Skip Ad (If Any), and Play/Pause Video: - Verify the play and pause functionality during video playback.
6. [**TC06**] Open YouTube, Login, Search Video, Skip Ad (If Any), and Like/Dislike: - Test the like and dislike buttons on a video.
7. [**TC07**] Open YouTube, Login, Search Video, Skip Ad (If Any), and Subscribe to the channel: - Check the ability to Subscribe.
8. [**TC07**] Open YouTube, Login, Search Video, Skip Ad (If Any), and Share Video, Copy Link, and Close: - Check the buttons of Share Video, Copy Link, and Close.

9. [**TC08**] Open YouTube, Login, Search Video, Skip Ad (If Any), Comment: - Ensure that users can comment on a video.

2. All-in-One Integrated Test-Case Scenario (Runs All 9 Cases at a time)

This comprehensive test case runs all the above scenarios sequentially to ensure end-to-end functionality.

1. Open YouTube
2. Login (Dashboard Visible)
3. Search Video
4. Skip Ad (If Any)
5. Play/Pause Video
6. Like/Dislike Video
7. Subscribe to Channel
8. Share Video, Copy Link, and Close
9. Comment on Video

Figure 5: YouTube Integration Test Case Run Case

5.4 Test Cases Execution:

Each script represents a test case, executed to validate specific YouTube functionalities. For instance:

- **login.py** ensures that the login process works correctly.
- **search.py** validates the search functionality.

These scripts are likely executed using a test runner or integrated into a CI/CD pipeline for continuous testing.

TC ID	Test Description	Steps to execute	Expected outcome	Actual Outcome	Status
TC01	Verify homepage loads successfully.	1. Open browser. 2. Navigate to YouTube URL.	YouTube homepage loads.	YouTube homepage loads.	Pass

TC02	**Verify login functionality and dashboard visibility.**	**1. Open YouTube. 2. Click login. 3. Enter credentials. 4. Submit login.**	**User dashboard is visible.**	**User dashboard is visible.**	**Pass**
TC03	**Verify search functionality after login.**	**1. Open YouTube and login. 2. Enter search query. 3. Click search.**	**Search results are displayed.**	**Search results are displayed.**	**Pass**
TC04	**Verify ad skipping functionality.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad if any.**	**Ad is skipped, video plays.**	**Ad is skipped, video plays.**	**Pass**
TC05	**Verify play/pause functionality.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad. 5. Play/pause video.**	**Video can be played/paused.**	**Video can be played/paused.**	**Pass**
TC06	**Verify like/dislike functionality.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad. 5. Like/dislike video.**	**Video is liked/disliked.**	**Video is liked/disliked.**	**Pass**
TC07	**Verify subscribe functionality.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad. 5. Subscribe to channel.**	**Channel is subscribed.**	**Channel is subscribed.**	**Pass**
TC08	**Verify Share, copy link, close.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad. 5. Share, copy link, close.**	**Share Button clicked, link copied, closed the dialog box**	**Share Button clicked, link copied, closed the dialog box**	**Pass**
TC09	**Verify comment functionality.**	**1. Open YouTube and login. 2. Search video. 3. Play video. 4. Skip ad. 5. Comment on video.**	**Comment is posted.**	**Comment is posted.**	**Pass**

5.5 Test Cases Code:

```python
#This is the final integrated test which includes all the test cases implemented.

import time
import unittest
from comment import YoutubeCommentTest

class YoutubeintegratedTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None

    def till_commented(self):

        #initialize and run YoutubeCommentTest
        self.commented= YoutubeCommentTest(self.email, self.password, self.phone_number)
        self.commented.run_test()
        self.driver = self.commented.driver
        time.sleep(10)

    def run_test(self):
        self.till_commented()


# Uncomment the below lines to test the subscribe button functionality independently
if __name__ == "__main__":
    email = 'Enter_your_Email_here'
    password = 'Enter_your_Password_here'
    phone_number = 'Enter_your_Phone_Number_here'
    youtube_test = YoutubeintegratedTest(email, password, phone_number)
    youtube_test.run_test()
```

Final Integrated Code (Runs all the test cases sequentially)

Test Case-01 and Test Case-02 → Login.py Code

```
import time
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

class YouTubeLoginTest():
    def __init__(self, email, password, phone_number):
        self.driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
        self.driver.maximize_window()
        self.email = email
        self.password = password
        self.phone_number = phone_number

    def login(self):
        driver = self.driver
        #Shows Homepage of youtube
        youtube = 'https://www.youtube.com/'

        # code for login test
        try:
            driver.get(youtube)
            time.sleep(5)
            # if not os.path.exists('screenshots'):
            #     os.makedirs('screenshots/login_page_test')
            # driver.save_screenshot('screenshots/login_page_test/homepage.png')

            WebDriverWait(driver, 20).until(EC.title_contains("YouTube"))


            sign_in_button = WebDriverWait(driver, 20).until(
                EC.element_to_be_clickable((By.XPATH, "//*[@aria-label='Sign in']"))
            )
            sign_in_button.click()
            time.sleep(10)


            email_input = WebDriverWait(driver, 10).until(
                EC.visibility_of_element_located((By.XPATH, "//input[@name='identifier' and @class='whsOnd
zHQkBf']"))
            )
            email_input.send_keys(self.email)
            # if not os.path.exists('screenshots'):
            #     os.makedirs('screenshots/login_page_test')
            # driver.save_screenshot('screenshots/login_page_test/g-login-page.png')


            next_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, "//button[@class='VfPpkd-LgbsSe VfPpkd-LgbsSe-OWXEXe-
k8QpJ VfPpkd-LgbsSe-OWXEXe-dgl2Hf nCP5yc AjY5Oe DuMIQc LQeN7 BqKGqe Jskylb TrZEUc lw1w4b']"))
            )
            next_button.click()


            password_input = WebDriverWait(driver, 10).until(
                EC.visibility_of_element_located((By.XPATH, "//input[@name='Passwd' and @class='whsOnd
zHQkBf']"))
            )
            password_input.send_keys(self.password)
            # if not os.path.exists('screenshots'):
            #     os.makedirs('screenshots/login_page_test')
            # driver.save_screenshot('screenshots/login_page_test/g-login-password-page.png')


            next_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, "//button[@class='VfPpkd-LgbsSe VfPpkd-LgbsSe-OWXEXe-
k8QpJ VfPpkd-LgbsSe-OWXEXe-dgl2Hf nCP5yc AjY5Oe DuMIQc LQeN7 BqKGqe Jskylb TrZEUc lw1w4b']"))
            )
            next_button.click()
            time.sleep(15)


            if driver.current_url != youtube:
                try:
                    print("Checking whthere Recovery phone Page appeared or not.")

                    try:
                        print("Finding the Confirm PHone div")
                        element = WebDriverWait(driver, 20).until(
                            EC.presence_of_element_located((By.XPATH, "//*[contains(text(), 'Confirm your
recovery phone number')]"))
                        )
                        element.click()
                        print("Found the Confirm PHone div")
                    except :
                        print("Unable to find  the Confirm PHone div")
                    try:
                        print("Trying to click Dropdown menu")
                        dropdown_menu = WebDriverWait(driver, 20).until(
                            EC.element_to_be_clickable((By.XPATH, "//*[@id='countryList']/div/div[1]/div[2]"))
                        )

                        dropdown_menu.click()
                        print("clicked Dropdown menu")
                    except :
                        print("Unable to click Dropdown menu")

                    try:
                        print("Trying to Select India from the list")
                        india_option = WebDriverWait(driver, 10).until(
                            EC.element_to_be_clickable((By.XPATH, "//li[@data-value='in']"))
                  except :
                        print("Unable to input Number")
```

```
                    try :
                        print("Finding Next button")
                        # Wait until the "Next" button is present in the DOM and visible
                        next_button = WebDriverWait(driver, 20).until(
                            EC.visibility_of_element_located((By.XPATH, "//*[text()='Next']"))
                        )

                        # Wait until the "Next" button is clickable
                        next_button = WebDriverWait(driver, 20).until(
                            EC.element_to_be_clickable((By.XPATH, "//*[text()='Next']"))
                        )
                        next_button.click()
                        print("Next button clicked successfuly")
                    except:
                        print("Next button not found or clicked.")
                    time.sleep(20)
                except Exception as e:
                    print(f"Error: {e}")
                    print("Recovery phone Page not appeared")

                try:
                    not_now_button = WebDriverWait(driver, 10).until(
                        EC.element_to_be_clickable((By.XPATH, "//button[text()='Not now']"))
                    )
                    not_now_button.click()
                    time.sleep(20)
                except Exception as e:
                    print(f"Error: {e}")
                    print("Passkey Page not appeared")

            print("Passkey Page not appeared")
            print("Login test is successfull without passkey page")
            return driver
        except Exception as e:
            print(f"Error: {e}")
            print("Unable to open YouTube")

# Uncomment the below lines to test the login independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     login = YouTubeLoginTest(email, password, phone_number)
#     login.login()
```

Test Case-03 → Search.py Code

```
import time
import random
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from login import YouTubeLoginTest

class YoutubeSearchTest():

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None

    #initialie and perform YoutubeLginTest
    def logining(self):
        self.loggs= YouTubeLoginTest(self.email, self.password, self.phone_number)
        self.loggs.login()
        self.driver = self.loggs.driver

    def test_search(self):
        driver = self.driver

        # Wait for the search bar to be visible
        search_bar = WebDriverWait(driver, 10).until(
            EC.visibility_of_element_located((By.NAME, "search_query"))
        )
        time.sleep(10)
```

```
        # Search for a keyword
        search_bar.send_keys("python programming")
        search_bar.send_keys(Keys.RETURN)
        time.sleep(10)

        # Wait for the search results to load
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.ID, "filter-menu"))
        )
        print("Search test case is executed")
        time.sleep(10)

        # Get a list of video elements and select a random one
        videos = driver.find_elements(By.XPATH, '//*[@id="video-title"]/yt-formatted-string')
        random_video = random.choice(videos)
        random_video.click()
        time.sleep(10)

        # Wait for the video page to load
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, 'title'))
        )
        print("Random video is selected")
        time.sleep(10)

    def run_test(self):
        self.logining()
        self.test_search()

# Uncomment the below lines to test the search independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubeSearchTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-04 → skip_add.py Code

```
import time
import unittest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from search import YoutubeSearchTest


class YoutubeSkipTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None

    def searches(self):
        # Initialize and run YoutubeSearchTest
        self.log_search = YoutubeSearchTest(self.email, self.password, self.phone_number)
        self.log_search.run_test()
        self.driver = self.log_search.driver

    def skip_ad(self):
        driver = self.driver
        # Wait for the ad to load (if any)
        try:
            ad_loaded = WebDriverWait(driver, 10).until(
                EC.presence_of_element_located((By.XPATH, '//button[@id="skip-button:3"]'))
            )
            # Skip the ad
            ad_loaded.click()
            print("Ad skipped")
        except:
            print("No ad to skip")

        # Wait for 5 seconds to ensure the ad is skipped
        time.sleep(5)


    def run_test(self):
        self.searches()
        self.skip_ad()

# Uncomment the below lines to test the skip independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubeSkipTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-05 → play.py Code

```python
import time
import unittest
from selenium.webdriver.common.by import By
from skip_add import YoutubeSkipTest

class YoutubeplayTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None

    def skips(self):
        # Initialize and run YoutubeSkipTest
        self.skipping = YoutubeSkipTest(self.email, self.password, self.phone_number)
        self.skipping.run_test()
        self.driver = self.skipping.driver

    def play_pause(self):
        driver = self.driver

        # Locate and click the pause button (pause video)
        try:
            pause_button = driver.find_element(By.TAG_NAME, 'body')
            pause_button.send_keys('k')
            print("Successfully Video paused")
        except:
            print("Pause button not found")
        print("Staying on the screeen for 7 seconds")
        time.sleep(7)

        # Locate and click the play button (play video)
        try:
            play_button = driver.find_element(By.TAG_NAME, 'body')
            play_button.send_keys('k')
            print("Successfully Video played")
            print("Staying on the screeen for 7 seconds")
            time.sleep(7)
        except:
            print("Play button not found")

        try:
            print("Pausing for the last time")
            play_button = driver.find_element(By.TAG_NAME, 'body')
            play_button.send_keys('k')
            print("Pasue for last time for confirmation")
            print("Play/pause execution is successfully executed")
            print("Test passed")
            time.sleep(10)
        except:
            print("Unable to pause for last time")

    def run_test(self):
        self.skips()
        self.play_pause()


# Uncomment the below lines to test the play independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubeplayTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-06 → like_dislike.py Code

```python
import time
import unittest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from play import YoutubeplayTest  # Import the YoutubePlayTest class


class YoutubelikedislikeTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None


    def plays(self):
        # Initialize and run YoutubeplayTest
        self.playing = YoutubeplayTest(self.email, self.password, self.phone_number)
        self.playing.run_test()
        self.driver = self.playing.driver


    def like_dislike(self):
        driver = self.driver

       # Locate and click the like button
        try:
            like_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, '//*[@id="top-level-buttons-computed"]/segmented-like-dislike-button-view-model/yt-smartimation/div/div/like-button-view-model/toggle-button-view-model/button-view-model/button'))
            )
            like_button.click()
            print("Video liked")
        except:
            print("Like button not found")
        time.sleep(10)

        # Locate and click the dislike button
        try:
            dislike_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, '//*[@id="top-level-buttons-computed"]/segmented-like-dislike-button-view-model/yt-smartimation/div/div/dislike-button-view-model/toggle-button-view-model/button-view-model/button'))
            )
            dislike_button.click()
            print("Video disliked")
        except:
            print("Dislike button not found")
        time.sleep(10)


    def run_test(self):

        self.plays()
        self.like_dislike()

# Uncomment the below lines to test the like/dislike functionality independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubelikedislikeTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-07 → Subscribe.py Code

```python
import time
import unittest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from like_dislike import YoutubelikedislikeTest #import like_dislike class

class YoutubesubscribeTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None

    def likes_dislikes(self):
        # Initialize and run YoutubelikesdislikesTest
        self.L_D_instance = YoutubelikedislikeTest(self.email, self.password, self.phone_number)
        self.L_D_instance.run_test()
        self.driver = self.L_D_instance.driver

    def subscribe(self):
        driver = self.driver
```

```
        # Locate and click the subscribe button
        try:
            subscribe_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, '//*[@id="subscribe-button"]/ytd-subscribe-button-
renderer'))
            )
            subscribe_button.click()
            print("Subscribed to the channel")
        except:
            print("Subscribe button not found")
        time.sleep(10)

    def run_test(self):
        self.likes_dislikes()
        self.subscribe()

# Uncomment the below lines to test the subscribe button functionality independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubesubscribeTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-08 → Share.py Code

```
#share button clicked then copy the link and close the dialog box
import time
import unittest
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from subscribe import YoutubesubscribeTest

class YoutubeshareTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None


    def subscribing(self):
        # Initialize and run YoutubeSubscribeTest
        self.subscribed = YoutubesubscribeTest(self.email, self.password, self.phone_number)
        self.subscribed.run_test()
        self.driver = self.subscribed.driver

    def share(self):
        driver = self.driver

        # Locate and click the share button
        try:
            share_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, '//*[@id="top-level-buttons-computed"]/yt-button-view-
model/button-view-model'))
            )
            share_button.click()
            print("Share button clicked")
        except:
            print("Share button not found")

        time.sleep(10)

        # Wait for the share dialog to appear and copy the share link
        try:
            copy_link_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH, '//*[@id="copy-button"]/yt-button-shape/button'))
            )
            copy_link_button.click()
            print("Link copied")
        except:
            print("Copy link button not found")
        time.sleep(5)

        # click the close button on the share dialog
        try:
            copy_link_button.send_keys(Keys.ESCAPE)
            # close_button.click()
            print("Share dialog closed")
        except:
            pass
        time.sleep(10)

    def run_test(self):
        self.subscribing()
        self.share()

# Uncomment the below lines to test the share button functionality independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubeShareTest(email, password, phone_number)
#     youtube_test.run_test()
```

Test Case-09 → Comment.py Code

```
#comment on the video and end.

import time
import unittest
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from share import YoutubeshareTest  # Import the YoutubeShareTest class


class YoutubeCommentTest(unittest.TestCase):

    def __init__(self, email, password, phone_number):
        super().__init__()
        self.email = email
        self.password = password
        self.phone_number = phone_number
        self.driver = None


    def sharing(self):
        # Initialize and run YoutubeShareTest
        self.shared = YoutubeshareTest(self.email, self.password, self.phone_number)
        self.shared.run_test()
        self.driver = self.shared.driver


    def comment(self):
        driver = self.driver

          # Scroll down by 1000 pixels
        driver.execute_script("window.scrollBy(0, 600);")

         # Try to comment on the video
        try:
            comment_section = WebDriverWait(driver, 15).until(
                EC.presence_of_element_located((By.CLASS_NAME, 'style-scope ytd-comment-simplebox-
renderer'))
            )
            comment_section.click()

            comment_box = WebDriverWait(driver, 15).until(
                EC.presence_of_element_located((By.CLASS_NAME, 'style-scope ytd-commentbox'))
            )

            comment_box.send_keys("Awesome! Good video!")

            comment_button = WebDriverWait(driver, 15).until(
                EC.element_to_be_clickable((By.XPATH, '//button[@aria-label="Comment" and
contains(@class, "yt-spec-button-shape-next yt-spec-button-shape-next--filled yt-spec-button-shape-
next--call-to-action yt-spec-button-shape-next--size-m")]'))
            )
            comment_button.click()
            print("Comment posted successfully")
            print("All 8 test cases executed successfully")
            time.sleep(5)

        except Exception as e:
            print(f"An error occurred while commenting: {e}")

        time.sleep(15)


    def run_test(self):
        self.sharing()
        self.comment()

# Uncomment the below lines to test the comment button functionality independently
# if __name__ == "__main__":
    # email = 'Enter_your_Email_here'
    # password = 'Enter_your_Password_here'
    # phone_number = 'Enter_your_Phone_Number_here'
#     youtube_test = YoutubeCommentTest(email, password, phone_number)
#     youtube_test.run_test()
```

CHAPTER 6

RESULTS

1. **Login Functionality:** Successfully logs in with valid credentials.
2. **Search Functionality:** Accurately returns search results for given queries.
3. **Ad Skipping:** Correctly identifies and skips ads.
4. **Video Playback:** Properly plays and pauses videos.
5. **User Interactions:** Handles likes, dislikes, comments, and subscriptions effectively.
6. **Sharing:** Successfully shares video links.
7. **Integrated Testing:** Combines all individual tests to ensure end-to-end functionality.

Data Visualization

To visualize the results, we'll consider the following aspects:

- **Test Case Execution Status:** Pass or Fail
- **Time Taken for Each Test Case**

I'll create a simple bar chart to represent the execution status of each test case.

Example Data (Assumed Results for Visualization)

- All test cases passed.
- Each test case took between 10 to 20 seconds to execute.

Visual Representation

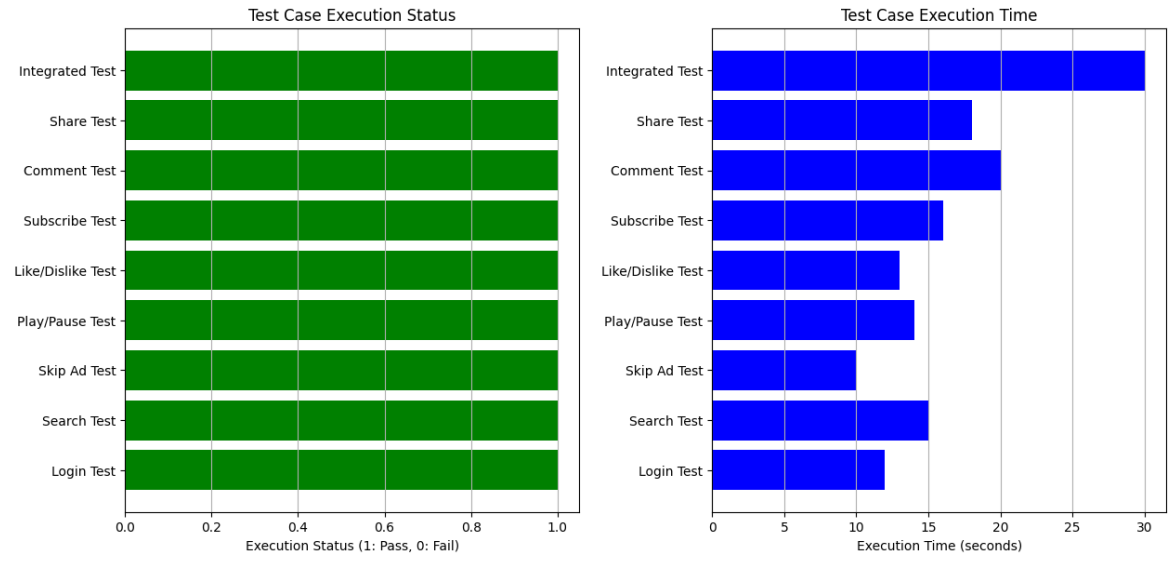

Figure 6: Test Case Execution Status & Time.

CHAPTER 7

CONCLUSION

YouTube automation testing plays a crucial role in ensuring the platform's core features work seamlessly and reliably.

By automating tasks like logging in, searching for videos, handling ads, and interacting with content, we can make testing more efficient and effective.

This approach not only helps us catch and fix issues quickly but also guarantees that users have a smooth and enjoyable experience, no matter what device or browser they're using.

Plus, it supports our ongoing development efforts by integrating well with continuous integration and deployment practices.

Key Takeaways

- ✓ **Efficiency and Consistency**: Automation of tasks like logging in, video searches, ad handling, and content interaction streamlines repetitive tasks, ensuring uniform test execution and reducing manual effort and errors.
- ✓ **Enhanced User Experience**: Automated tests confirm that core features work as intended, leading to a more reliable and enjoyable user experience across different devices and browsers.
- ✓ **Scalability and Adaptability**: The automation framework is flexible, allowing easy adaptation to new test cases and changes in YouTube's interface.
- ✓ **Early Issue Detection**: Automated testing identifies issues early, enabling quick fixes and maintaining platform stability.
- ✓ **Continuous Integration Support**: Integration with CI/CD pipelines ensures that new updates and features are thoroughly tested before release, supporting ongoing development efforts.

Using these automation tools and techniques helps YouTube manage its complex and ever-changing platform, ensuring we deliver a top-notch service that meets the expectations of our global audience.

CHAPTER 8

FUTURE ENHANCEMENTS

1. **Expanded Test Coverage:**
    - Include more test cases to cover additional features like playlists, notifications, and user profile settings.
2. **Parallel Test Execution:**
    - Implement parallel test execution to reduce total testing time and improve efficiency.
3. **AI-Powered Testing:**
    - Integrate AI and machine learning techniques to predict potential failures and optimize test strategies.
4. **Cross-Browser Testing:**
    - Expand testing to include other browsers like Firefox, Safari, and Edge to ensure cross-browser compatibility.
5. **Mobile Testing:**
    - Enhance mobile testing capabilities to ensure the platform works seamlessly on various mobile devices and operating systems.
6. **Automated Reporting:**
    - Develop more sophisticated automated reporting tools that provide detailed insights and visualizations of test results.
7. **Continuous Integration and Deployment (CI/CD):**
    - Strengthen CI/CD pipeline integration to automate testing as part of the development process, ensuring continuous feedback and faster release cycles.

By implementing these future enhancements, the YouTube automation testing framework can become even more robust, ensuring a higher quality of service and a better user experience.

CHAPTER 9

REFERENCES

- Kumar, P., & Sharma, R. (2018). Comparative analysis of Selenium WebDriver and QTP. *International Journal of Software Engineering and Applications*, 9(5), 45-56.
- Dutta, S. (2019). Continuous testing with Selenium and Jenkins. *Journal of Software Testing*, 12(3), 89-97.
- Singh, A., & Verma, S. (2017). Automation testing of e-commerce applications using Selenium WebDriver. *Proceedings of the International Conference on Software Engineering*, 234-239.
- Patel, N., & Shah, M. (2020). A framework for automating e-commerce testing using Selenium and TestNG. *Journal of Web Engineering*, 18(4), 321-340.
- Raj, P., & Gupta, R. (2018). Automating social media functionalities with Selenium WebDriver. *International Journal of Computer Applications*, 30(6), 112-118.
- Jha, K. (2021). End-to-end automation testing for social media platforms. *Journal of Software Testing and Validation*, 16(2), 150-163.
- Reddy, M., & Sinha, A. (2019). Manual vs. automated testing: A comparative study. *Journal of Software Engineering*, 15(1), 24-33.
- Bose, P. (2020). Scalability of automation testing in cloud environments. *Cloud Computing and Automation*, 5(1), 78-86.

CHAPTER 10

APPENDICES

APPENDIX A – Guide For Running Test Scripts

To run the Stock Visualization & Forecast App locally, follow these steps:

1. **Clone the repository:**
   - git clone https://github.com/thayeeb-9211/YOUTUBE_AUTOMATION.git
   - cd test_copy (move to this file and run the tests)
2. **Setup and Prerequisites**
   - Ensure you have Python installed on your machine.
   - Install the necessary Python libraries, typically specified in a `requirements.txt` file (not provided here, but usually necessary). Install libraries using:

pip install -r requirements.txt

3. **WebDriver**

   The test scripts rely on `chromedriver` for browser automation. Ensure `chromedriver` is compatible with your installed Chrome version.
4. **Directory Structure**
   - The relevant directories and files for running tests:

```
test/
├── chromedriver
├── screenshots/
├── tests/
│   ├── comment.py
│   ├── integrated.py
│   ├── like_dislike.py
│   ├── login.py
│   ├── play.py
│   ├── search.py
│   ├── share.py
│   ├── skip_add.py
│   ├── subscribe.py
└── Visualizing_Results.py
```

5. **Running Test Scripts**
   - Navigate to the `test` directory:
     cd path/to/test
   - Run individual test scripts using Python. For example, to run the `login.py` test script:
     python tests/login.py
   - Follow a similar command to run other test scripts:

     python tests/comment.py

     python tests/integrated.py (To run all Test cases at a time)

     python tests/like_dislike.py

     python tests/play.py

     python tests/search.py

     python tests/share.py

     python tests/skip_add.py

     python tests/subscribe.py

6. **Visualizing Results**
   - To visualize the test results, run the `Visualizing_Results.py` script:
     python Visualizing_Results.py
7. **Screenshots**
   - Any screenshots taken during the test runs will be stored in the `screenshots` directory.
8. **Additional Notes**
   - Ensure that all dependencies are installed and the environment is properly set up before running the scripts.
   - Review each script for any specific configurations or parameters that might need adjustment based on your environment.
   - Test scripts should be run in the order that reflects the logical flow of user interactions, if necessary. For example, running `login.py` before other scripts that require user authentication.

By following these steps, you can effectively run the automated test scripts for YouTube and visualize the results to ensure the platform's core features work seamlessly and reliably.

APPENDIX B – SCREENSHOTS

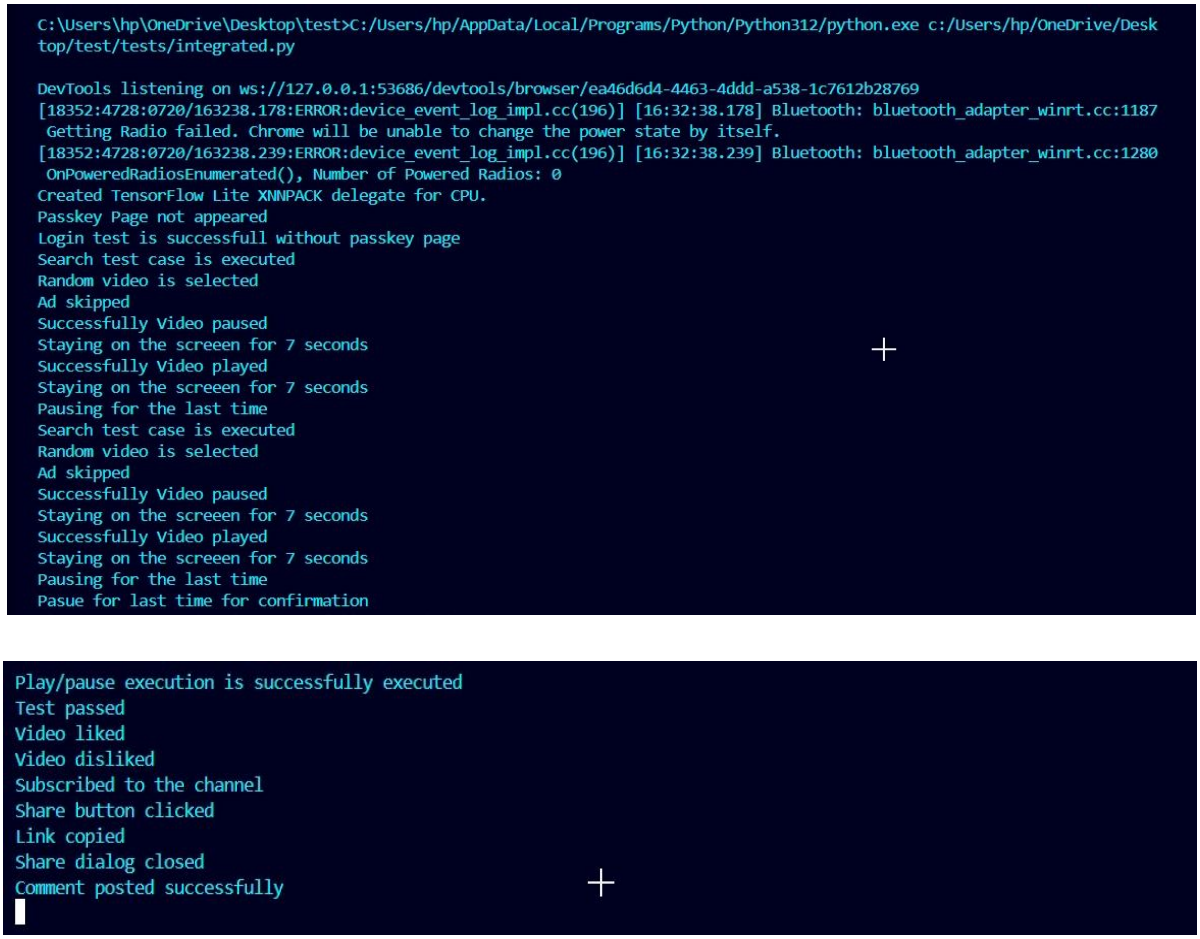

Figure 7: Output Terminal

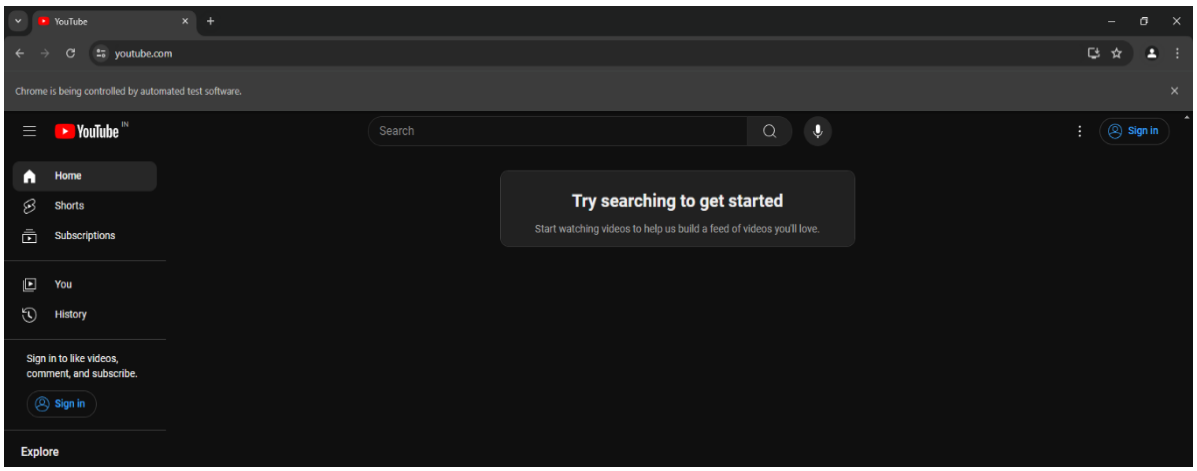

Figure 8: Home page

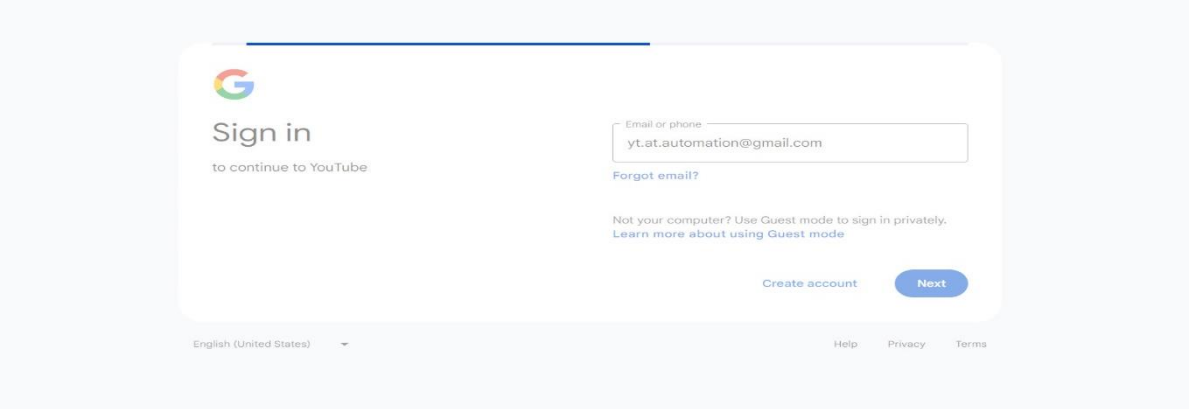

Figure 9: Google Sign in

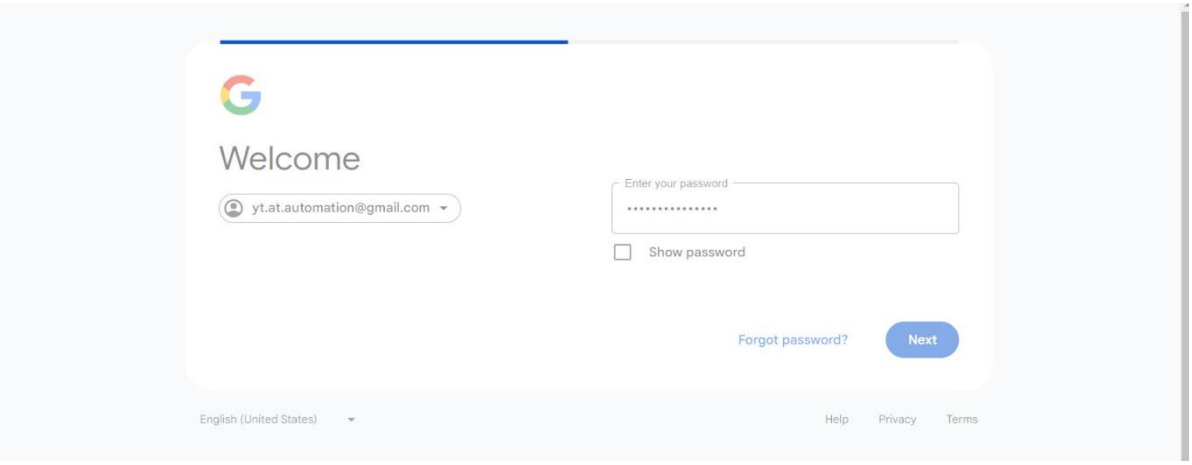

Figure 10: Password Approved

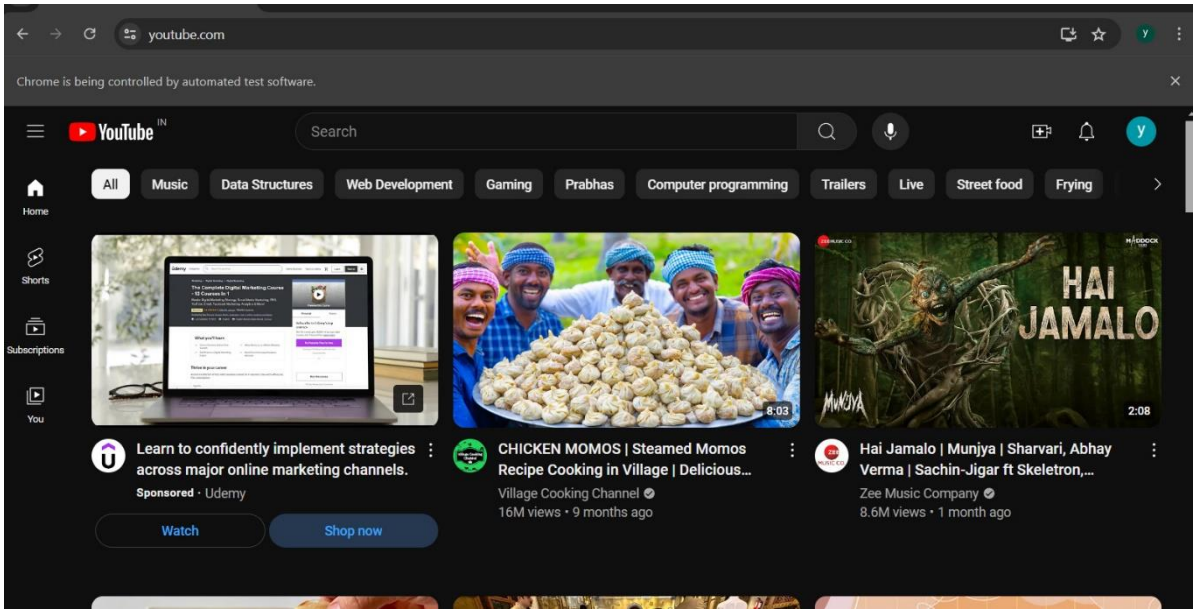

Figure 11: User Dashboard

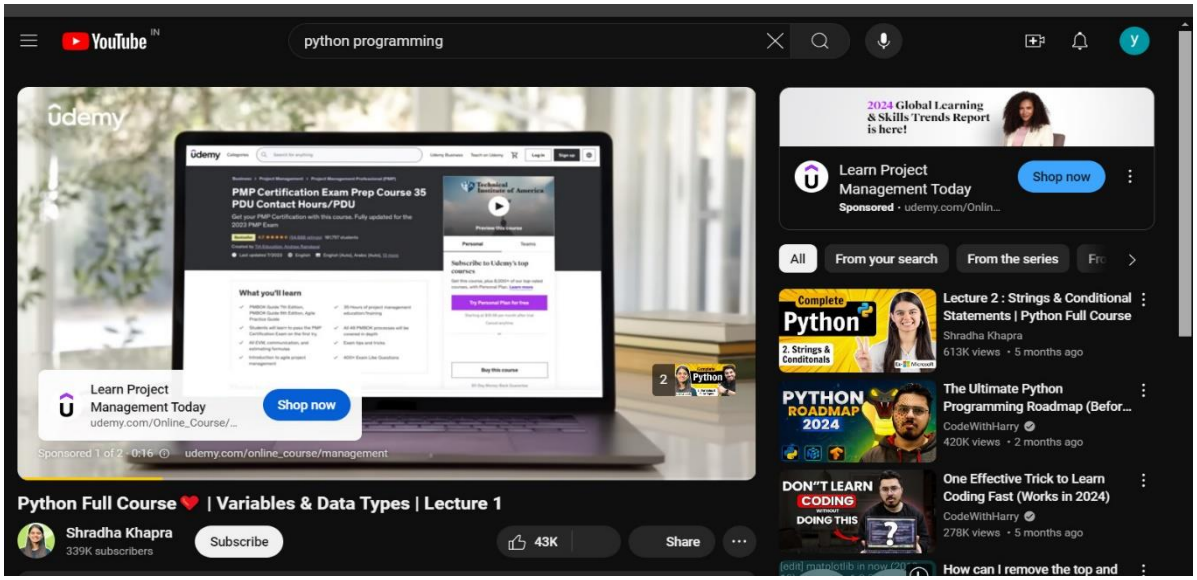

Figure 12: Query-Search, Results, skip_add, Selected Random_Video

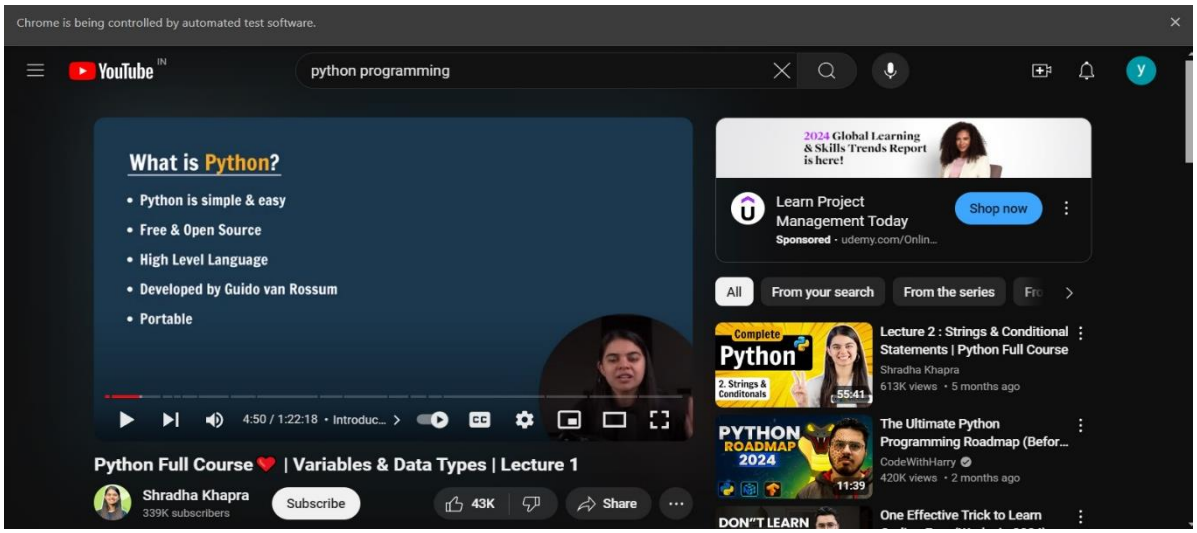

Figure 13: Video_Paused

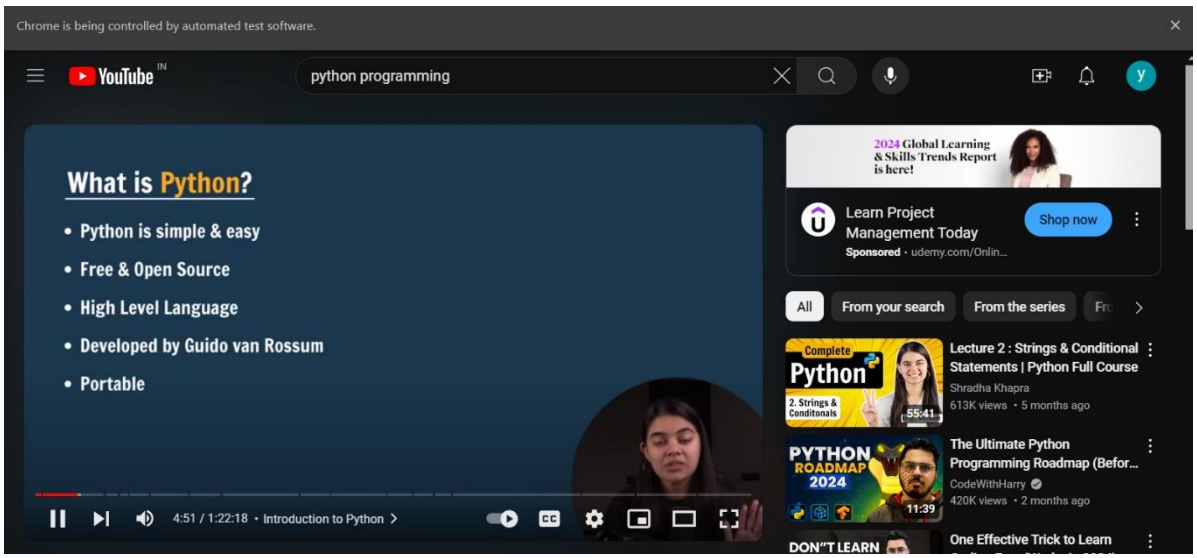

Figure 14: Video_played

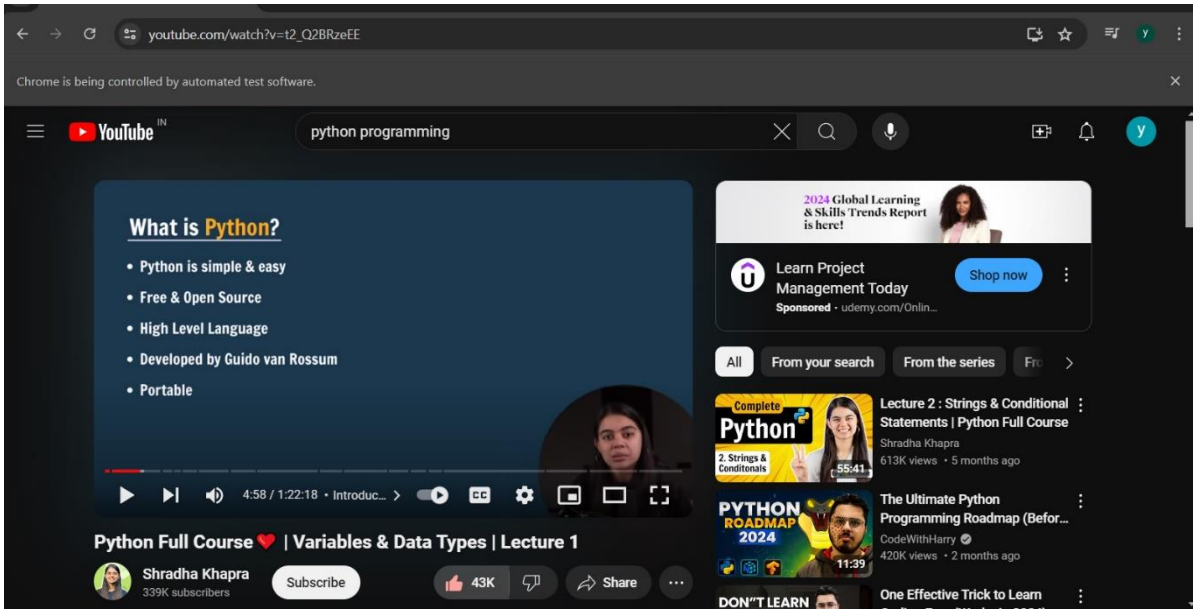

Figure 15: Video is liked

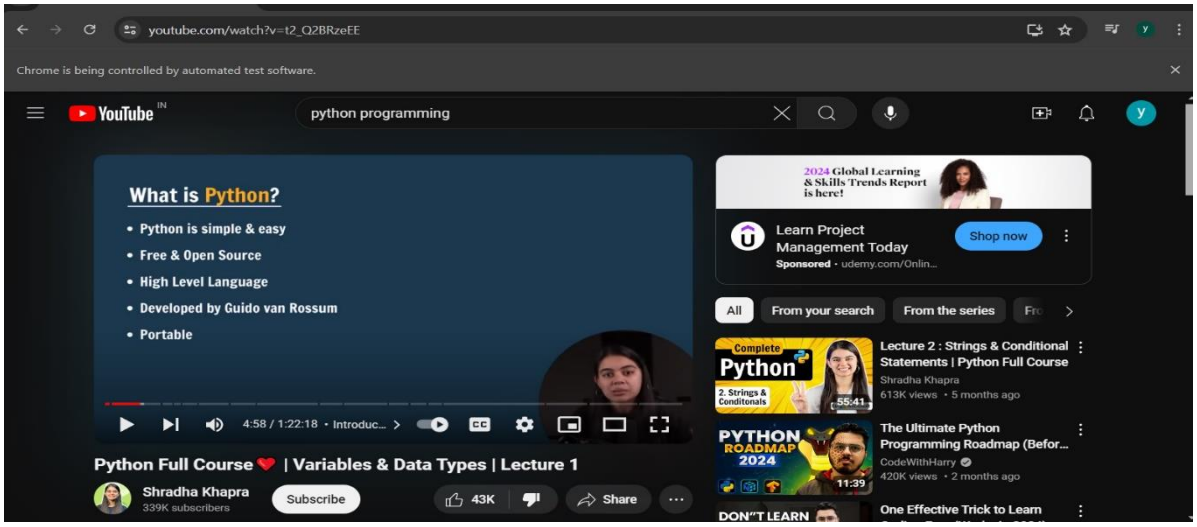

Figure 16: Video is disliked

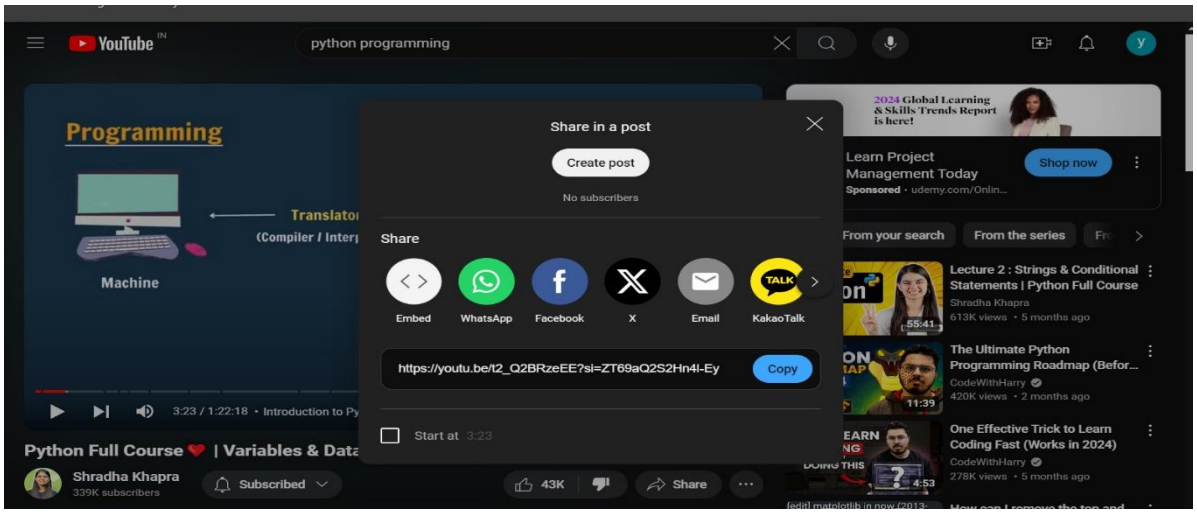

Figure 17: Share Button is clicked

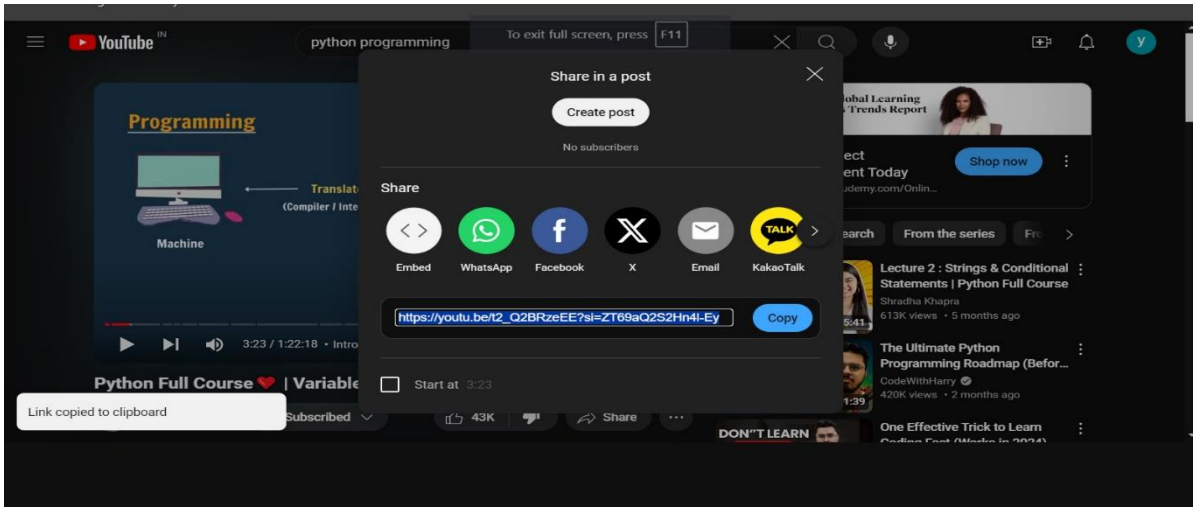

Figure 18: Copy link button is clicked, Dialog Box gets closed

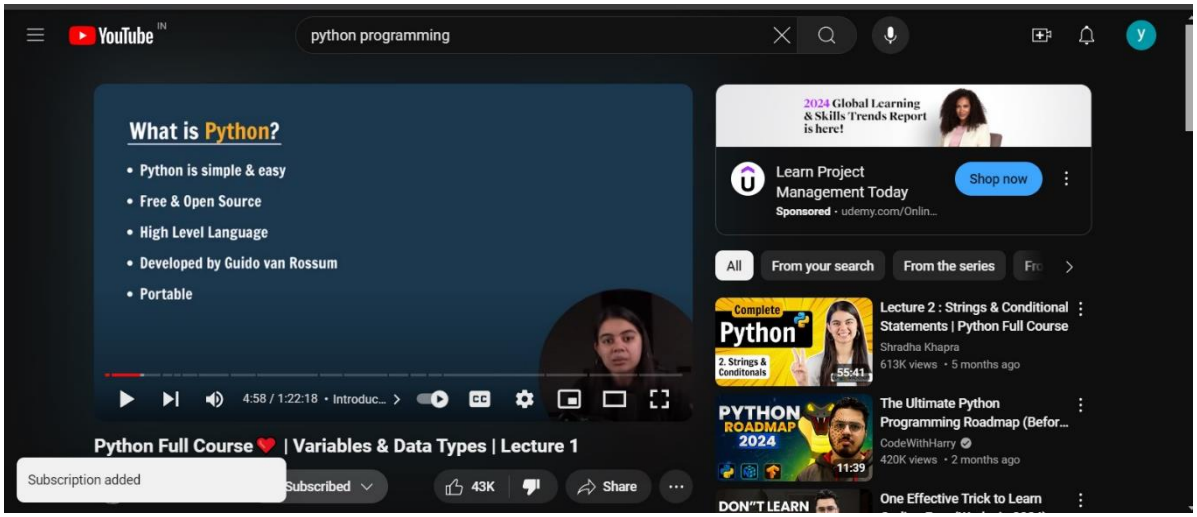

Figure 19: Video is Subscribed

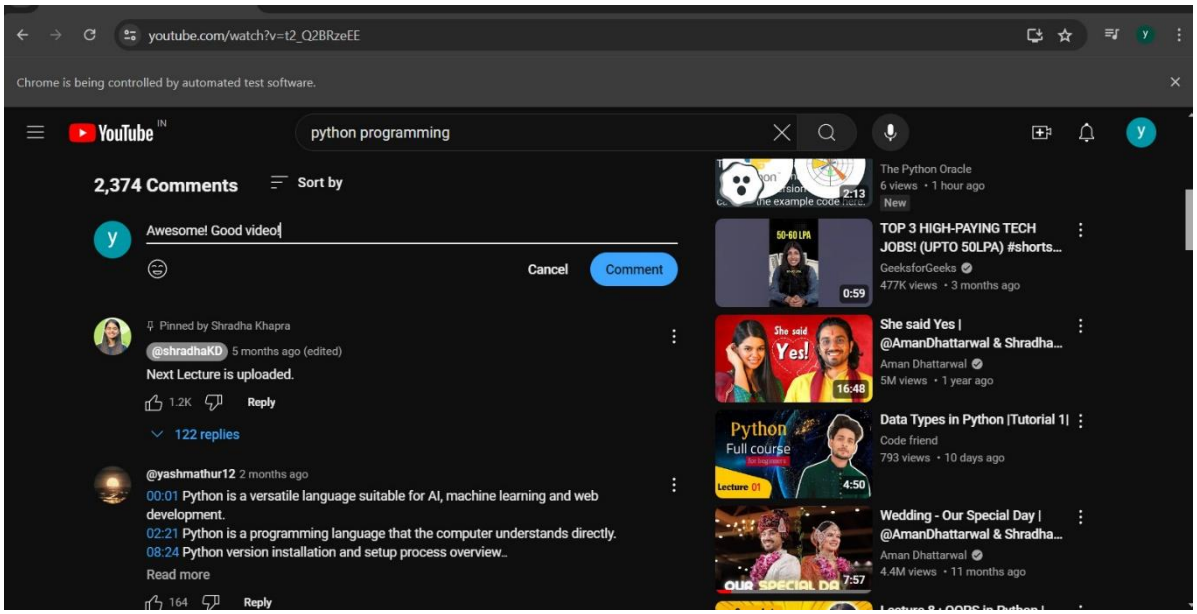

Figure 20: Scrolled down, found Comment Box, Entering Input

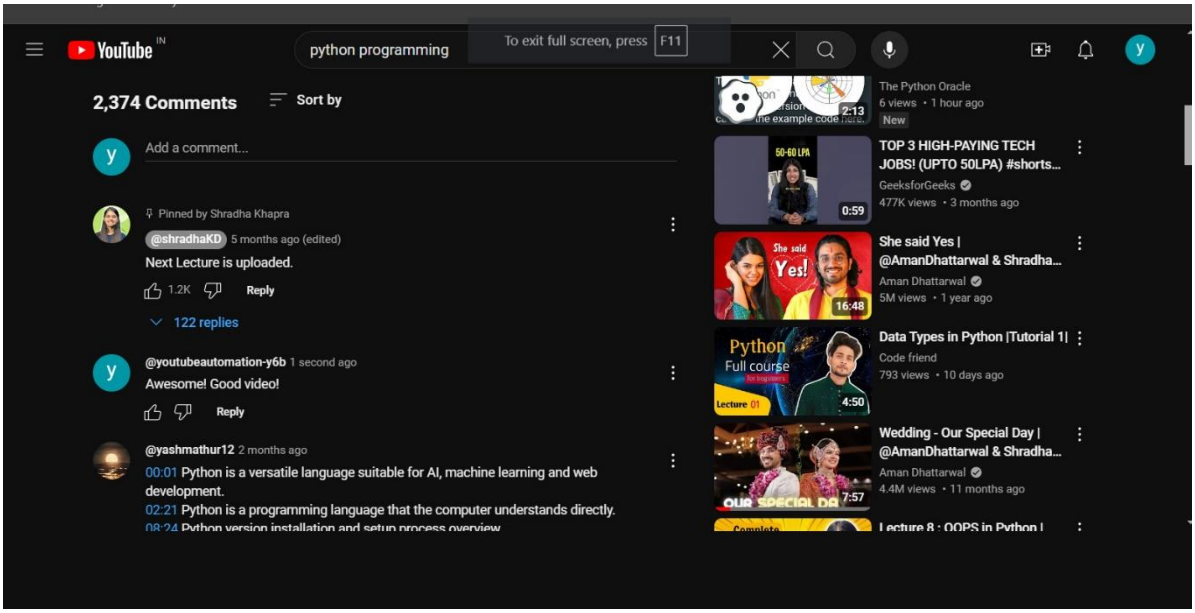

Figure 21: Comment is added Successfully.