INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING &TECHNOLOGY (IJCET)

ISSN Print: 0976-6367
ISSN Online: 0976-6375

Publishers of High Quality Peer Reviewed Refereed Scientific, Engineering & Technology, Medicine and Management International Journals

PUBLISHED BY

IAEME

IAEME Publication

Chennai, India

https://iaeme.com/Home/journal/IJCET

International Journal of Computer Engineering and Technology (IJCET)
Volume 16, Issue 4, July-August 2025, pp. 12-29, Article ID: IJCET_16_04_002
Available online at https://iaeme.com/Home/issue/IJCET?Volume=16&Issue=4
ISSN Print: 0976-6367; ISSN Online: 0976-6375; Journal ID: 5751-5249
Impact Factor (2025): 18.59 (Based on Google Scholar Citation)
DOI: https://doi.org/10.34218/IJCET_16_04_002

INTEGRATING PLAYWRIGHT WITH JENKINS FOR SCALABLE AUTOMATION TESTING IN CI/CD PIPELINES

Twinkle Joshi

IQGeo, Canada.

ABSTRACT

The evolution of modern web applications has necessitated sophisticated testing strategies that can keep pace with rapid deployment cycles while ensuring comprehensive coverage and security. Automation testing has emerged as a critical component of the software quality assurance process, providing validation of complete user workflows and system integrations. However, the integration of automation testing into CI/CD pipelines presents unique challenges, particularly in areas of scalability, security, and observability.

Microsoft Playwright has gained significant traction as a powerful automation testing framework, offering cross-browser testing capabilities, robust API handling, and advanced automation features. When combined with Jenkins, one of the most widely adopted CI/CD platforms, organizations can create powerful automated testing workflows. However, this integration introduces complex considerations around security, particularly regarding credential management, test data isolation, and report security.

This article addresses three critical aspects of Playwright-Jenkins integration: efficient pipeline implementation with parallelization strategies, comprehensive

security frameworks for credential management, and advanced reporting mechanisms that provide real-time insights into test execution and results. This article presents practical implementation guidelines to secure, scalable, and observable test automation. By creating an integrated ecosystem, development teams can achieve higher test coverage, faster feedback loops, and improved collaboration between testing and development teams.

Keywords: End-to-End Testing, Automation Testing, Playwright, Jenkins, CI/CD, Secure Test Automation, Parallel Test Execution, Credential Management, Environment Variables, Custom Reporting, Playwright HTML Reporter.

Cite this Article: Twinkle Joshi. (2025). Integrating Playwright with Jenkins for Scalable Automation Testing in CI/CD Pipelines. *International Journal of Computer Engineering and Technology (IJCET)*, 16(4), 12–29.

https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_16_ISSUE_4/IJCET_16_04_002.pdf

1. Introduction

Modern software development demands fast, reliable, and secure testing processes integrated seamlessly into Continuous Integration and Continuous Delivery (CI/CD) pipelines. Tools like **Playwright** and **Jenkins** have emerged as powerful allies in achieving this goal. While each tool brings its own strengths, their integration enables a robust and scalable test automation ecosystem that addresses several critical operational and technical challenges faced by development teams.

1.1. Understanding the Tools

1.1.1. Playwright

Playwright is an open-source browser automation framework designed for end-to-end testing and web scraping. It supports multiple browsers—Chromium, Firefox, and WebKit—via a single API, enabling developers to simulate real user behavior (clicking, navigating, typing, etc.) across various platforms. Playwright is optimized for speed, especially in headless mode, which is ideal for server-side execution.

Key features include:

- Multi-language support (JavaScript, TypeScript, Python, .NET, Java)
- Built-in test runner (@playwright/test) with support for parallel execution, automatic waiting, assertions, and rich reporting

- Diagnostic artifacts such as screenshots, video recordings, and execution traces for enhanced debugging

1.1.2. Jenkins

Jenkins is a leading open-source automation server that enables continuous integration and delivery. It allows teams to define and automate every stage of their CI/CD pipeline via Jenkinsfiles, making tasks like building, testing, and deploying repeatable and consistent.
Core capabilities include:

- Plugin-based architecture for high customizability
- Pipeline-as-code support for defining CI/CD workflows
- Distributed execution on agent nodes or cloud environments
- Secure credential storage and environment variable injection

1.1.3. Why Integrate Playwright with Jenkins?

When used together, Playwright and Jenkins provide a continuous testing setup that improves quality, speeds up feedback cycles, and reduces manual intervention. This integration empowers teams to:

- Automatically trigger cross-browser UI tests on each code commit or pull request
- Identify bugs early through reliable, repeatable test executions
- Improve team efficiency with faster and more actionable test feedback
- Maintain consistent, isolated, and secure test environments
- Generate in-depth diagnostic reports for faster issue resolution

1.2. Addressing CI/CD Test Automation Challenges with Playwright and Jenkins

1.2.1. Managing Test Execution Time with Parallelization

As test suites grow, execution times increase, slowing down feedback and deployment cycles.

Solution:

- Playwright enables parallel test execution across isolated workers, reducing total runtime without sacrificing reliability.
- Jenkins pipelines can be configured to run test jobs concurrently on different agents or containers (via Docker/Kubernetes), improving overall efficiency.
- Result aggregation plugins help consolidate outputs from parallel jobs for unified visibility.

1.2.2. Ensuring Efficient Resource Management and Test Isolation

Parallel execution can cause resource contention, especially on shared infrastructure, leading to test flakiness.

Solution:

- Playwright ensures test isolation by running each test in a separate browser context or instance.
- Jenkins allows resource control through node labeling and integration with cloud-based provisioning tools (e.g., Kubernetes), ensuring that tests run in appropriately sized and isolated environments.
- Dynamic resource allocation ensures optimal usage and scalability.

1.2.3. Securing Credentials in CI/CD Environments

Storing sensitive data like API keys in test code or config files poses a major security risk.

Solution:

- Jenkins offers secure credential management, allowing secrets to be safely injected into jobs at runtime.
- Playwright tests can access these credentials via environment variables, avoiding hardcoded values.

1.2.4. Simplifying Result Interpretation with Advanced Reporting

Basic test logs often lack the information needed to quickly diagnose failures in complex applications.

Solution:

- Playwright automatically captures screenshots, execution traces, error logs, and video recordings during test runs.
- These artifacts are archived through Jenkins and visualized using plugins like HTML publisher, providing a rich UI for result analysis.
- This level of detail enables developers to resolve bugs faster and with greater confidence.

The integration of Playwright and Jenkins offers solutions to the key challenges of test automation within CI/CD pipelines. This powerful combination allows development teams to:

- **Reduce test execution time** with intelligent parallelization strategies

- **Ensure reliable, isolated test environments** through advanced resource and agent management
- **Maintain strong security posture** by safeguarding sensitive credentials
- **Accelerate bug diagnosis and resolution** with in-depth, visual test reports

2. Parallel Test Execution with Jenkins and Playwright in CI/CD Pipeline

As modern test suites expand—either through increased test coverage or support for multiple browsers—executing them sequentially can become inefficient. To optimize execution time and CI/CD efficiency, both Playwright and Jenkins support various forms of parallel test execution.

2.1. Parallelism at the Playwright Level

A. Multiple Workers (Concurrency at Test Level):

Playwright's test runner allows concurrent execution of tests using worker threads. By default, Playwright selects the number of workers based on CPU cores, but this can be overridden in the `playwright.config.ts` or via CLI using `--workers`. This method is effective on powerful CI agents but requires tuning to avoid test flakiness due to resource contention. For example:

```
export default defineConfig({
  workers: 3,
  retries: process.env.CI ? 2 : 0,
});
```

B. Browser-Level Parallelism Using Projects:

Tests can also be parallelized across multiple browsers by defining separate "projects" within the configuration file. Each browser project runs its test suite in isolation and can utilize its own worker pool:

This configuration enables cross-browser testing in parallel, improving test efficiency and coverage.

```
    projects: [
      { name: 'Chromium', use: { browserName: 'chromium' } },
      { name: 'Firefox', use: { browserName: 'firefox' } },
    ]
```

2.2. Parallelism at the Jenkins Pipeline Level

A. Parallel Pipeline Stages:

Jenkins allows execution of parallel stages within the pipeline. Separate stages can be used to run different browser tests concurrently. Each stage can be configured to use different agents or environments to avoid resource conflicts. Example:

```
parallel {
  stage('Chrome Tests') {
    steps {
      sh 'npx playwright test --browser=chromium'
    }
  }
  stage('Firefox Tests') {
    steps {
      sh 'npx playwright test --browser=firefox'
    }
  }
}
```

This method helps reduce overall test execution time without depending solely on Playwright's internal parallelism.

B. Dynamic Sharding Across Multiple Agents:

Playwright also supports test sharding using the `--shard=<index>/<total>` flag, which can be integrated into Jenkins pipelines by dynamically generating stages. This allows large test suites to be divided across multiple agents, enabling horizontal scalability:

```
npx playwright test --shard=1/5
```

With Jenkins scripted pipelines, it's possible to programmatically spawn multiple shard-based parallel stages, making this suitable for extensive, resource-intensive test environments.

2.3. Considerations for Effective Parallel Execution

- **Test Independence:**

  Ensure tests do not share state or dependencies that can lead to race conditions or resource conflicts. Isolate outputs, ports, and environments when running in parallel.

- **Resource Constraints:**

  Higher degrees of parallelism increase system load. Monitor CPU, memory, and I/O usage on Jenkins nodes. For large-scale testing, consider distributing execution across multiple Jenkins agents.

3. Secure Credential Management in Jenkins and Playwright

3.1. Jenkins Credential Store Integration

Jenkins provides structured and secure ways to handle credentials in Pipelines using built-in methods and best practices. This includes support for secret text, username/password pairs, secret files, and other credential types such as SSH keys or certificates. Each type is accessed using the `credentials()` helper method within the `environment` directive or via the `withCredentials` step.

A. Handling Secret Text Credentials

Secret text credentials can be injected into environment variables using the `credentials()` method inside the `environment` block. For example:

```
environment {
   AWS_ACCESS_KEY_ID = credentials('jenkins-aws-secret-key-id')
   AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
}
```

These environment variables are then used securely during pipeline execution. Jenkins masks secret values in logs with `****`, helping reduce accidental exposure. However, it does not completely prevent malicious misuse; therefore, avoid using trusted credentials in untrusted Pipelines.

B. Handling Username and Password Credentials

Username/password pairs can also be handled via `credentials()`, which automatically sets three environment variables:

- `VAR` (e.g., `BITBUCKET_COMMON_CREDS`) – contains `username:password`
- `VAR_USR` – username only
- `VAR_PSW` – password only

```
environment {
    BITBUCKET_CREDS = credentials('jenkins-bitbucket-common-creds')
}
```

These variables are scoped to the block or stage where they are defined. Logging or exposing them is similarly masked, but users must avoid malicious jobs accessing them.

C. Handling Secret Files

For credentials stored as files (e.g., kubeconfigs or binary credentials), Jenkins injects a temporary path to the file into an environment variable:

```
environment {
    MY_KUBECONFIG = credentials('my-kubeconfig')
}
```

This value can be used in shell commands, such as `kubectl`, during pipeline execution.

D. Using withCredentials for Complex Credential Types

For SSH keys, certificates, and other non-standard credentials, Jenkins provides the `withCredentials` step. It allows temporary binding of secure variables within a specific code block. This method is often used when dealing with multiple credentials or when fine-grained scoping is required.

Example:

```
withCredentials(bindings: [
  sshUserPrivateKey(credentialsId: 'jenkins-ssh-key-for-abc', keyFileVariable: 'SSH_KEY_FOR_ABC'),
  certificate(credentialsId: 'jenkins-certificate-for-xyz', keystoreVariable: 'CERTIFICATE_FOR_XYZ', passwordVariable: 'XYZ-CERTIFICATE-PASSWORD')
]) {
    // Use $SSH_KEY_FOR_ABC, $CERTIFICATE_FOR_XYZ securely
}
```

These credentials are available only within the `withCredentials` block, further isolating their use.

3.2. Test Configuration Isolation

Proper test configuration isolation ensures that sensitive configuration data remains protected while maintaining test execution flexibility.

A. Environment-Specific Configuration: It automatically selects appropriate settings based on execution context:

```
const config = {
    development: {
        baseURL: process.env.DEV_BASE_URL,
        timeout: 30000
    },
    staging: {
        baseURL: process.env.STAGING_BASE_URL,
        timeout: 45000
    },
    production: {
        baseURL: process.env.PROD_BASE_URL,
        timeout: 60000
    }
};
```

Rather than direct credential embedding, this injection pattern maintains security while providing necessary authentication capabilities.

B. Securing Credentials with Environment Variables

Managing sensitive data like API keys, passwords, and tokens securely is crucial in automated testing. Playwright supports a secure approach to handle credentials through environment variables.

- **Using a `.env` File (With `dotenv`):** Keeps sensitive data like API keys out of your codebase.
- **Steps**:

- Install `dotenv`

```
npm install dotenv
```

- Create a `.env` file containing secrets:

```
API_KEY=secure-api-key
PASSWORD=supersecretpassword
```

- Load this file in Playwright script:

```
require('dotenv').config();
await page.fill('#password', process.env.PASSWORD);
```

- **Best Practice**: Use different .env files (.env.dev, .env.prod) for each environment and load them dynamically with dotenv.config({ path: ... }).

4. Advanced Reporting with Playwright and Jenkins in CI/CD Pipeline

Efficient test reporting is vital in automated testing for diagnosing failures, tracking test coverage, and enabling collaboration. Playwright supports multiple advanced reporting mechanisms, including its native HTML reporter, integration with Allure Reports, and support for custom report generation, making it ideal for CI/CD environments like Jenkins.

4.1. Playwright Native HTML Report

- **Overview**: Playwright comes with a built-in, interactive HTML reporter.
- **Features**:
  - Displays test results with pass/fail status.
  - Captures traces, screenshots, and logs for debugging.
  - Generates reports `--reporter=html` and opens via `npx playwright show-report`.
- **Jenkins Integration**:
  - Generate the report in a build step and archive the output (`playwright-report/`).
  - Use **HTML Publisher Plugin** in Jenkins to display it as a post-build action.

4.2. Allure Reporting with Playwright

- **Overview**: Allure provides rich, customizable test reports with timelines, attachments, and historical trends.
- **Features**:

- Test result visualization and analysis capabilities that significantly enhance the value of test execution data.
- Feature-based categorization
- Risk level classification
- Execution time grouping
- Browser compatibility tagging
- Playwright's screenshot and video recording capabilities integrate seamlessly with Allure reporting, providing visual context for test failures

- **Jenkins Integration**:
  - Use **Allure Jenkins plugin**.
  - Post-build action can automatically generate and display the Allure report.

4.3. Custom Reporting

- **Overview**: Tailored for specific business needs (e.g., CSV exports, dashboard APIs).
- **Implementation guideline**:
  - Implement a custom reporter by extending Playwright's reporter API.
  - Collect events (`onTestBegin`, `onTestEnd`, etc.) and log data as needed (JSON, DB, REST API).
- **Jenkins Integration**:
  - Output files can be parsed or pushed to external systems
  - Suitable for organizations with proprietary reporting dashboards.

5. Setting Up Jenkins Pipeline: A Step-by-Step Guide

Jenkins pipeline automates the process of executing Playwright automation tests in parallel and publishes rich HTML reports. Below is a detailed breakdown of each stage and its purpose:

5.1. Set Up a Jenkins Pipeline Job

In the Jenkins dashboard:

- Click **"New Item"** (or **"New Job"**).
- Give your job a name, such as `Playwright-Pipeline`.
- Select **Pipeline** as the job type and click **OK**.
- On the configuration screen, scroll to the **Pipeline** section.
- Choose **Pipeline script.**

5.2. Pipeline script implementation logic

```
pipeline {

  agent any

  environment {
    PLAYWRIGHT_IMAGE = 'mcr.microsoft.com/playwright:v1.52.0-noble'
    WORK_DIR = '/app'
  }

  stages {
    stage('Safe Clean Workspace') {
      steps {
        sh '''
          docker run --rm -v "$PWD":/app -w /app alpine sh -c "rm -rf * .??*"
        '''
      }
    }

    stage('Checkout') {
      steps {
        deleteDir()
        git url: 'https://example-url.git', branch: 'main'
      }
    }

    stage('Install Dependencies') {
      steps {
        sh """
          docker pull ${PLAYWRIGHT_IMAGE}
          docker run --rm \
            -v "${env.WORKSPACE}:${WORK_DIR}" \
            -w ${WORK_DIR} \
            ${PLAYWRIGHT_IMAGE} bash -c '
              npm ci
              npx playwright install
            '
        """
      }
    }

    stage('Run Tests in Parallel') {
      steps {
        script {
          def testFolders = sh(
            script: 'find src/tests -mindepth 1 -maxdepth 1 -type d',
            returnStdout: true
          ).trim().split('\n').findAll { it?.trim() }
```

```
                if (testFolders.isEmpty()) {
                    error "No test folders found in src/tests"
                }

                def parallelStages = [:]

                for (f in testFolders) {
                    def folder = f.trim()
                    def folderName = folder.replaceFirst('^src/tests/', '').replaceAll('/', '-')
                    def reportDir = "playwright-report-${folderName}"

                    parallelStages[folderName] = {
                        stage("Test: ${folderName}") {
                            catchError(buildResult: 'FAILURE', stageResult: 'FAILURE') {
                                withCredentials([
                                    usernamePassword(
                                        credentialsId: 'app-test-user',
                                        usernameVariable: 'TEST_USERNAME',
                                        passwordVariable: 'TEST_PASSWORD'
                                    ),
                                    string(credentialsId: 'api-key', variable: 'API_KEY')
                                ]) {
                                    sh """
                                        docker run --rm \
                                            -v "${env.WORKSPACE}:${WORK_DIR}" \
                                            -w ${WORK_DIR} \
                                            -e JENKINS=true \
                                            -e PLAYWRIGHT_HTML_REPORT_NAME="${reportDir}" \
                                            ${PLAYWRIGHT_IMAGE} bash -c '
                                                npx cross-env NODE_ENV=test playwright test ${folder} --
project=chromium --reporter=html
                                            '
                                    """
                                }
                            }
                        }
                    }
                }

                parallel parallelStages
            }
        }
    }
    stage('Publish HTML Reports') {
        steps {
            script {
                def reportDirs = sh(
                    script: 'find . -type d -name "playwright-report-*"',
                    returnStdout: true
                ).trim().split('\n').findAll { it?.trim() }
```

```
                for (reportDir in reportDirs) {
                    def folderName = reportDir.replaceFirst(/^.*playwright-report-/, '')

                    publishHTML target: [
                        reportName: "Report: ${folderName}",
                        reportDir: reportDir,
                        reportFiles: 'index.html',
                        keepAll: true,
                        alwaysLinkToLastBuild: true,
                        allowMissing: false,                    ]
                }
            }
        }
    }
  }
}
```

Pipeline Stages Explained

A. Safe Clean Workspace

Ensure that the workspace starts from a clean state, even if there are residual files from previous builds, including those owned by `root` or another user.

a. Steps:

i. A lightweight `alpine` Docker container is used to mount the current workspace (`$PWD`) and recursively delete all files (`*`, hidden files `.??*`, etc.).

ii. This is safer and more comprehensive than `deleteDir()` because it removes files even with restrictive permissions or ownership.

b. Justification:

i. Prevents inconsistent test results due to leftover files.

ii. Ensures a clean environment for dependency installation and test execution.

B. Checkout

Fetch the latest source code from a version control system (VCS) such as Git.

a. Steps:

i. `deleteDir()` is used to clear the Jenkins workspace.

ii. The `git` step clones the specified branch (`main`) from the remote repository.

b. Justification:

i. Ensures the pipeline works on the latest codebase.

ii. Reduces the risk of conflicts caused by previously cached or outdated code.

C. `Install Dependencies`

Set up the required environment by installing project dependencies and Playwright-specific browser binaries.

Steps:

i. Pulls the official Playwright Docker image.
ii. Mounts the workspace into the container and sets it as the working directory.
iii. Executes:
    1. `npm ci`: Installs node dependencies based on `package-lock.json` for reproducibility.
    2. `npx playwright install`: Installs necessary browser binaries (Chromium, Firefox, WebKit).

Justification:

i. Guarantees that the environment inside Jenkins is identical to that used locally or in other CI systems.
ii. Ensures cross-browser testing capabilities are available at runtime.

D. `Run Tests in Parallel`

Discover and run Playwright test suites concurrently, each isolated in its own Docker container.

Steps:

i. Discovers directories inside `src/tests` using `find`. Each subfolder is assumed to represent a separate test suite or domain.
ii. For each discovered test folder:
    1. A corresponding Jenkins stage is created dynamically.
    2. Playwright report directories are prepared.
    3. Jenkins environment variables are used to construct full URLs to build and artifact pages.
    4. Inside each dynamically generated stage:
        a. Credentials are injected securely using `withCredentials`.
        b. A Docker container runs the Playwright tests with custom environment variables for reporting.

Justification:

i. Parallelization reduces overall execution time significantly.
ii. Isolated containers ensure test reliability by eliminating shared state.

iii. Secure handling of secrets like API keys prevents leakage or misuse.

E. `Publish HTML Reports`

Generate and publish rich HTML-based playwright test reports within the Jenkins interface for each test suite.

Pre-requisite: **Install HTML Publisher Plugin**

This plugin enables Jenkins to display HTML reports—like the Playwright test report—directly on the build page. You can install it through Manage Plugins for better visual representation of test results. While it's possible to archive the report files without the plugin, viewing them within Jenkins might be restricted by its Content Security Policy unless you manually adjust those settings. Using the HTML Publisher plugin is a safer and more reliable option for styled report viewing.

Steps:

i. Discovers all playwright-reports generated.

ii. Uses the publishHTML Jenkins plugin to render the index.html report inside the web UI, with options to:

1. reportName: A friendly name shown in Jenkins UI, like Report: abc.
2. reportDir: The directory where the HTML report is located.
3. reportFiles: The file to render, usually index.html.
4. keepAll: If true, keeps reports from all builds, not just the last one.
5. alwaysLinkToLastBuild: If true, always links to the most recent build's report.
6. allowMissing: If false, the build will fail if the report is missing

Justification:

i. Playwright reports offer greater insights than plain text logs (e.g., test trace, screenshots, video playback).

ii. Having direct links to each test suite's report improves visibility for QA and dev teams.

iii. Encourages fast and effective issue resolution by presenting results in a visual format.

5.3. Save and Execute the Jenkins Pipeline

After adding logic for the pipeline script, click 'Save' and then navigate to pipeline job and click **"Build Now."** This triggers Jenkins to fetch the latest code and execute the defined pipeline steps. For the successful build playwright reports will be generated for each test suite that were executed parallelly. It will be shown as a clickable link on the build page sidebar.

6. Conclusion

Integrating Playwright with Jenkins offers a powerful solution for modern CI/CD-based test automation workflows. The synergy of Playwright's fast, cross-browser test execution with Jenkins' orchestrated and repeatable pipelines enables rapid feedback, reduced flakiness, and increased test coverage.

A key contribution of this integration lies in its ability to facilitate intelligent parallelization strategies at multiple layers. At the Playwright level, parallel execution is achieved using worker threads and test sharding mechanisms, enabling fine-grained control over execution concurrency. Jenkins pipelines can be configured to execute test directories or browser-specific test projects concurrently across distributed agents or containerized infrastructure, leveraging Docker or Kubernetes for scalability. These techniques significantly reduce the end-to-end test execution time, a critical factor in maintaining fast and reliable CI/CD feedback cycles without compromising test isolation or reproducibility.

Security remains a foundational pillar of this integration. By leveraging Jenkins' native credential store and its withCredentials binding capabilities, sensitive artifacts such as API keys, login tokens, and certificates can be securely injected at runtime while remaining protected from unauthorized access or logging exposure. This ensures compliance with enterprise-level security standards and mitigates risks associated with hard-coded secrets in shared codebases.

Equally critical is the role of observability in automated test systems. The integration supports multi-layered reporting frameworks to deliver actionable insights. Playwright's built-in HTML reporter offers trace-based visualizations, including screenshots, console logs, and network activity. Allure reporting extends this capability by providing interactive dashboards, historical test trends, and feature-based aggregation of results. Additionally, the implementation of custom reporting pipelines—such as JSON or database-backed dashboards—enables organizations to align test data visualization with specific operational KPIs or business intelligence platforms.

Collectively, these practices establish a comprehensive and adaptive test automation framework. They foster enhanced collaboration among development, QA, and DevOps stakeholders by promoting transparency, facilitating test traceability, and accelerating defect resolution. As such, the Playwright-Jenkins integration represents a critical enabler in the pursuit of continuous quality, delivering measurable improvements in software reliability, deployment velocity, and organizational agility.

References

[1] "Setting Up a Playwright Jenkins Pipeline: A Comprehensive Guide," BrowserCat, February. 20, 2025. https://www.browsercat.com/post/setting-up-playwright-jenkins-pipeline.

[2] "15 CI/CD Challenges and its Solutions," Sanghita Ganguly, BrowserStack, November 15, 2024. https://www.browserstack.com/guide/ci-cd-challenges-and-solutions.

[3] "Playwright," Microsoft, 2025. https://playwright.dev/.

[4] "Jenkins," Jenkins, 2025. https://www.jenkins.io/.

[5] "Using a Jenkinsfile," Jenkins Documentation, 2025. https://www.jenkins.io/doc/book/pipeline/jenkinsfile/#handling-credentials.

[6] "Generating a Custom Report for Playwright with Charts," Pradap Pandiyan, Medium, September 28, 2024. https://pradappandiyan.medium.com/generating-a-custom-report-for-playwright-with-charts-c4d10c17c892.

[7] "Allure Report," Qameta Software, 2025. https://allurereport.org/

Citation: Twinkle Joshi. (2025). Integrating Playwright with Jenkins for Scalable Automation Testing in CI/CD Pipelines. International Journal of Computer Engineering and Technology (IJCET), 16(4), 12–29.

Abstract Link: https://iaeme.com/Home/article_id/IJCET_16_04_002

Article Link:
https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_16_ISSUE_4/IJCET_16_04_002.pdf

✉ **editor@iaeme.com**