The Impact of Implementing Containerization into CI/CD Testing Pipelines

Or Brener[1140102] email obrener@uoguelph.ca,
Danindu Marasinghe[1093791] email dmarasin@uoguelph.ca, and
Eric Morse[1141504] email emorse@uoguelph.ca

University of Guelph, 50 Stone Rd E, Guelph, ON, Canada, N1G 2W1

Abstract. This paper explores the effects of containerization into Continuous Integration/Continuous Deployment (CI/CD) frameworks to enhance test automation processes. The study investigates how containerization impacts the efficiency and scalability of testing workflows while identifying the challenges it introduces. Through an analysis of CI/CD tools integrated with Docker containers, this research presents both the benefits and complexities associated with implementing containerized environments into CI/CD pipelines.

Keywords: Software Testing · Continuous integration · DevOps · Agile Software Development

1 Introduction

The need for rapid and reliable software releases has intensified with modern development practices, making CI/CD pipelines an essential component in many development environments. CI/CD automates the integration, testing, and deployment of code changes, allowing teams to streamline workflows and reduce the time-to-market. Recent advancements have introduced containerization technologies, such as Docker and Kubernetes, into CI/CD, further enhancing the capabilities of automated test environments by providing isolated and consistent execution contexts. This research aims to examine the impact of containerization on CI/CD test automation, focusing on the benefits of increased testing speed, scalability, and reliability, as well as challenges such as start-up investment, resource management, and integration complexities.

2 Background

Integrating Continuous Integration/Continuous Deployment (from here on out shortened to CI/CD) practices with containerization has revolutionized modern software development practices. By automating and standardizing the building, testing, and deployment process of application development, the industry has been able to foster an environment of faster, and more reliable software delivery.

Here, we will discuss foundational concepts of CI/CD as well as containerization, setting the stage for the investigation of their combined impact on test automation frameworks.

1. Understanding CI/CD
   CI/CD is a set of software development practices which aim to improve the speed, reliability, and consistency of application delivery. It automates the software life cycle stages, from code integration to deployment, ensuring rapid feedback and reducing manual intervention.
   - Continuous Integration (CI):
     CI involves the frequent integration of code changes into a shared repository. This process is automated, with each code commit triggering a build and a series of tests to validate the changes. CI emphasizes early detection of integration issues, and helps ensure a stable code-base.
     Key benefits of CI include:
     - Immediate feedback on code quality through automated testing.
     - Faster resolution of integration conflicts.
     - A stable foundation for subsequent deployment processes.
   - Continuous Deployment (CD):
     CD extends the principles of CI by automating the deployment of code changes to production or, a staging environment. Every code change that passes every test, is staged and automatically released. This ensures a consistent and rapid delivery cycle.
     Key benefits of CD include:
     - Accelerated time-to-market by eliminating manual deployment steps.
     - Enhanced reliability through consistent and repeatable deployment processes.
     - Reduced risk of errors during deployment.

   CI/CD has become a cornerstone of DevOps practices. It fosters a community of collaboration among development, operations, and QA teams. By automating repetitive tasks, and ensuring code stability, CI/CD enables organizations to respond quickly to market demands while maintaining high-quality standards. Through supporting concepts such as permission to fail and rapid feedback cycles, CI/CD is perfect for teams working in agile methodology.

2. The Role of Containerization
   Containerization is lightweight virtualization approach that encapsulates applications and their dependencies into portable, isolated units called containers. Unlike a traditional Virtual Machine (VM), containers share the host system's operating system kernel, making them extremely efficient in terms of resource utilization and startup times.
   - Key Features of Containers:

  - Isolation: each container operates in its own isolated environment, ensuring that applications run consistently across different systems.
  - Portability: Containers can run on any system with a compatible container runtime, for example Docker.
  - Resource Efficiency: Containers are lightweight compare to virtual machines, enabling higher density and faster scalability.

- Docker and Kubernetes:
  Docker is one of the most widely used containerization platforms, providing tools to build, ship and run containers. Kubernetes is a orchestration platform, which extends Docker's capabilities by automating the deployment, scaling,and management of containerized applications.
  Together, these tools are able to address critical challenges often faced in software development:
  - Environment Consistency: Using containers, we are able to ensure that applications behave identically in development, testing, and production environments, reducing the "works on my machine" problem.
  - Scalability: Containerization orchestrations platforms, like Kubernetes, enable dynamic scaling of applications based on precised demand.
  - Rapid Deployment: Containers allow for faster creation of environments, accelerating testing and deployment cycles.
- Containerization in CI/CD:
  Containers are being increasingly integrated into CI/CD pipelines to enhance their automation and scalability. By running tests in a containerized environment, CI/CD workflows can achieve:
  - Parallel execution of tests in isolated containers, therefore reducing overall runtime.
  - Consistent test results across environments, improving reliability and communication across teams.
  - Streamlined deployments with container images, serving as the deployment artifact.

Despite its advantages, containerization can also introduces challenges such as security concerns, resource overheads, and management complexity. Addressing these issues requires careful planning and expertise in container orchestration.

CI/CD and containerization represent two trans-formative technologies in modern software development. While CI/CD automates and streamlines the software life cycle, containerization ensures consistency and scalability across environments. Their integration creates powerful synergies, enabling faster, more reliable software delivery. What we have described above, should provide the conceptual foundation for exploring their combined impact on test automation frameworks.

3 Methodology

This research employs a mixed-methods approach, combining both qualitative and quantitative analysis to assess the benefits and challenges of implementing containerization within CI/CD test automation.

3.1 Literature Review

The literature reviewed in this study explores the intersection of containerization and CI/CD practices, focusing on their application in automated test frameworks. The primary goal is to understand how these technologies enhance efficiency, scalability, and consistency while addressing challenges in their implementation. This section synthesizes insights from key research sources, offering a foundation for the present study.

Containerization and its Role in Modern DevOps Practices :
Containerization, particularly through tools like Docker and Kubernetes, has emerged as a transformative technology in software development. Raj's research in Containerization and Its Impact on DevOps Practices highlights its ability to standardize application deployment and enhance scalability by isolating applications in lightweight containers. These containers encapsulate application dependencies, providing a consistent environment across development, testing, and production stages. This consistency addresses a common problem in software development—discrepancies between local and production environments, colloquially known as the "works on my machine" issue[3].

The research emphasizes containerization's seamless integration with DevOps practices, particularly in enabling automation and micro-service architectures. By supporting independent scaling and deployment of services, containerization aligns with agile methodologies, offering faster iteration cycles. However, the paper also notes significant challenges, including security vulnerabilities in container images, operational complexity, and resource overheads. Raj's study forms a foundation for understanding containerization's transformative role and the trade-offs involved.

CI/CD Pipelines and Their Integration with Containerized Environments :
Mustyala's work, CI/CD Pipelines in Kubernetes: Accelerating Software Development and Deployment, delves into how CI/CD pipelines benefit from Kubernetes' orchestration capabilities. CI/CD automation ensures continuous integration and testing of code changes, improving collaboration among development, operations, and QA teams. By leveraging Kubernetes, these pipelines

gain the ability to scale dynamically, automate deployments, and ensure system reliability through self-healing mechanisms[2]. The paper outlines the critical role of Kubernetes in ensuring scalability and consistency during deployments. Kubernetes' declarative configuration files, such as Helm charts and YAML manifests, ensure that application deployments are consistent and reproducible. Additionally, Mustyala highlights the benefits of blue-green and canary deployment strategies enabled by Kubernetes, which minimize downtime and reduce risk during software roll outs.

While the integration of CI/CD and Kubernetes accelerates the development process, the study also emphasizes challenges, including the steep learning curve of Kubernetes and the operational overhead of managing complex containerized systems. These findings are critical for framing the scalability and reliability aspects of containerized CI/CD pipelines.

Parallel Testing in Dockerized CI/CD Frameworks :

Majumder's thesis, Maximizing Efficiency: Automated Software Testing with CI/CD Tools and Docker Containerization for Parallel Execution, provides an in-depth analysis of how Docker containers enhance test automation in CI/CD pipelines. The study evaluates the execution of regression tests in a containerized GitLab CI/CD environment, demonstrating significant reductions in test execution times through parallelization[1].

The research details a pipeline architecture that builds Docker images, creates configurations for child pipelines, and triggers parallel execution. The use of multiple containers ensures that tests run in isolated environments, eliminating the need for shared resource cleanup between tests. This approach enables the execution of hundreds of tests concurrently, reducing the time required for regression testing from hours to minutes.

Majumder also discusses the automatic retry mechanisms incorporated to address transient test failures, such as network issues or memory constraints. While the study highlights the advantages of parallel testing, it also identifies challenges, such as managing system resources and ensuring compatibility across different operating systems used in containers.

Challenges and Limitations of Containerization in CI/CD :

Despite its benefits, containerization introduces unique challenges in CI/CD workflows. Raj's study identifies security concerns as a major limitation, with vulnerabilities in container images and runtime environments requiring continuous monitoring. Additionally, the resource overhead of managing large-scale containerized systems can offset the efficiency gains achieved through parallel execution[3].

Another significant issue is the skill gap in containerization technologies. As noted in multiple studies, the rapid evolution of tools like Docker and Kubernetes has created a demand for specialized expertise, which many organizations lack. This gap can hinder the adoption and effective management of containerized environments.

Moreover, vendor lock-in remains a concern, particularly with proprietary features of container orchestration tools. Organizations relying heavily on specific platforms may face challenges in migrating to alternative solutions. This issue underscores the need for open standards and best practices to ensure interoperability and flexibility in CI/CD workflows.

Best Practices for Implementing Containerized CI/CD Pipelines :

Several studies propose best practices for implementing containerized CI/CD pipelines. Mustyala recommends adopting declarative configurations for CI/CD pipelines and Kubernetes resources to ensure reproducibility. Tools like Helm simplify the deployment of complex applications, while automated rollbacks and canary deployments minimize risk during updates[2].

Raj emphasizes the importance of security in containerized pipelines. Regularly scanning container images, implementing role-based access control (RBAC), and defining network policies are crucial steps to mitigate vulnerabilities. Additionally, leveraging monitoring tools like Prometheus and Grafana enables organizations to gain insights into performance and resource utilization, helping optimize pipelines over time[3].

Majumder highlights the value of parallel execution for improving test efficiency, particularly for large test suites. However, the study advises caution in scaling parallel tests to avoid overloading system resources. Configuring CI/CD tools to manage resource allocation dynamically is critical for maintaining system stability[1].

The Evolution of CI/CD with Containerization :

The convergence of containerization and CI/CD practices represents a significant evolution in software development. Containers not only enhance the automation of testing and deployment but also enable organizations to embrace micro-service architectures effectively. By decoupling services and their dependencies, containerization fosters modularity and independent scaling, which are crucial for modern software systems.

The reviewed studies collectively underscore that while containerized CI/CD pipelines offer unparalleled efficiency, scalability, and consistency, their implementation requires careful planning. Organizations must address challenges related to security, resource management, and skill gaps to fully realize the potential of these technologies.

The literature demonstrates the trans-formative impact of containerization on CI/CD practices, particularly in test automation frameworks. While containers enhance efficiency and scalability, they also introduce complexities that demand robust solutions. The insights from these studies inform the design and evaluation of the proposed framework, guiding this research in addressing key challenges and optimizing the integration of containerization into CI/CD environments.

3.2 Empirical Analysis

This study investigates the benefits and challenges of implementing containerization with CI/CD test automation frameworks through empirical testing. The focus is on evaluating key performance metrics that reflect the efficiency, scalability, and reliability of containerized test workflows. This section outlines the experimental design, data collection process, and evaluation metics used in the study.

Experimental Design :
The empirical testing was conducted using two configurations:

1. Non-Containerized CI/CD Workflow:
   A baseline pipeline running tests sequentially in a traditional CI/CD environment without containerization. This configuration uses GitHub actions as the testing environment, executing all test cases in a single, shared runtime.
2. Containerized CI/CD Workflow:
   A pipeline configured with a `Dockerfile` and container to enable scalability and ease of execution of tests. Each test suite runs within isolated containers, leveraging a lightweight Python image environment.

The experiment compared the two configurations using identical test cases to ensure consistency.

Data Collection Process

1. Pipeline Setup:
   - The pipelines were set up in a controlled environment to minimize external variability.
   - Docker containers were built with predefined base images and dependencies to ensure consistent test environments across iterations
2. Script Development:
   - A python script `scientific_calculator.py` was developed, which includes a number of small python methods with apparent expected outputs.
3. Test Suite:
   - The test suite in the file named `test_scientific_calculator.py` included units tests to validate core functionalities of our application.
4. Execution:
   - For each configuration, the pipeline was executed 25 times to gather statistically significant data.
   - Results were logged for total workflow runtime, test runtime, and pass/-failure rates.

Evaluation of Metrics :

The performance of the two configurations was evaluated based on the following metrics:

1. Total Workflow Runtime:
   - This metrics measures the total time taken form the initiation of the CI/CD pipeline to the delivery of the final test results. This includes, but is not limited to, startup time, build time, and test time.
   - It provides a comprehensive view of how containerization impacts overall efficiency, including setup, execution, and reporting stages.
2. Test Runtime:
   - This metric focuses specifically on the duration of test execution, excluding setup and tear-down times.
   - It highlights the impact running a containerized instance might have on testing times.
3. Pass/Failure Rate:
   - Pass/Failure rates for individual tests were tracked to ensure accuracy and consistency of results between the two configurations.
   - A mismatch in results would indicate issues with environmental parity or test dependence.

Statistical Analysis :

- Descriptive statistics (mean, median, and mode) were computed for each metric to summarize performance trends.
- Inferential statistics, including paired t-tests, were used to assess whether the observed differences between configurations were statistically significant.
- Anomalous data points, such as outlier runtime or unexpected test failures, were identified and analyzed to determine their root causes. Although none were found.

Reliability Measures :

- To ensure reliability, each pipeline was monitored for environmental consistency, including resource allocation (CPU and memory) and network stability.
- The containerized pipeline was test for environmental isolation by introducing minor variations in test environments to confirm that the results remained unaffected.
- Logs and performance data were stored for reproducibility and audit purposes.

The empirical approach outlined provided a robust framework for evaluating the efficiency and reliability of containerized CI/CD workflows. By focusing on workflow runtime, test runtime, and pass/failure rates, the study identifies key performance trade-offs and highlights the practical implications of containerization for automated test environments.

4 Findings

The integration of Docker containers within CI/CD pipelines brings several distinct advantages to test automation:

4.1 Benefits

Efficiency in Test Execution Containers facilitate parallel execution of test suites, significantly reducing total execution time compared to sequential testing. For instance, using Dockerized CI/CD pipelines led to faster completion times by enabling multiple tests to run concurrently within isolated environments[Containerization-and-its-impact-on-DevOps-practices].

Scalability Containers allow easy scaling of testing environments, especially when used with Kubernetes orchestration, enabling efficient resource management and load distribution during peak testing periods.

Consistency and Portability Containers maintain consistency across different testing environments, mitigating the “works on my machine” problem. This consistency enables seamless transitions from development to production.

4.2 Challenges

However, challenges were also observed:

Resource Overheads : Running multiple containers can strain system resources, particularly CPU and memory, when scaling test suites extensively.

Security and Isolation : Containers introduce security concerns, especially when handling sensitive data in shared CI/CD environments. Ensuring isolation between test containers and production systems requires careful configuration.

Complexity in Management : Setting up and maintaining containerized CI/CD systems can be complex, requiring familiarity with orchestration tools like Kubernetes and container security best practice.

4.3 Graphs and Tables

Run #	Non-Containerized Test Run Time (s)	Non-Containerized Workflow Run Time (s)
1	0.3	10
2	0.36	10
3	0.27	8
4	0.29	8
5	0.3	10
6	0.33	12
7	0.27	7
8	0.4	13
9	0.32	8
10	0.34	12
11	0.29	12
12	0.26	9
13	0.29	9
14	0.32	7
15	0.27	8
16	0.26	8
17	0.34	8
18	0.28	8
19	0.27	11
20	0.26	8
21	0.26	7
22	0.27	7
23	0.28	8
24	0.31	9
25	0.33	6

Table 1. Raw data from non-containerized runs

Run #	Containerized Test Run Time (s)	Containerized Workflow Run Time (s)
1	0.26	20
2	0.25	14
3	0.31	13
4	0.27	13
5	0.24	16
6	0.26	12
7	0.25	18
8	0.25	17
9	0.25	14
10	0.24	17
11	0.3	15
12	0.3	15
13	0.25	15
14	0.28	13
15	0.27	13
16	0.27	13
17	0.3	16
18	0.29	17
19	0.25	12
20	0.24	13
21	0.24	14
22	0.27	18
23	0.25	14
24	0.26	15
25	0.25	17

Table 2. Raw data from containerized runs

Run #	Non-containerized Test Run Time / Non-Containerized Workflow Run Time
1	3.00%
2	3.60%
3	3.38%
4	3.63%
5	3.00%
6	2.75%
7	3.86%
8	3.08%
9	4.00%
10	2.83%
11	2.42%
12	2.89%
13	3.22%
14	4.57%
15	3.38%
16	3.25%
17	4.25%
18	3.50%
19	2.45%
20	3.25%
21	3.71%
22	3.86%
23	3.50%
24	3.44%
25	5.50%

Table 3. Test time percentage of total workflow time for non-containerized runs

Run #	Containerized Test Run Time / Containerized Workflow Run Time
1	1.30%
2	1.79%
3	2.38%
4	2.08%
5	1.50%
6	2.17%
7	1.39%
8	1.47%
9	1.79%
10	1.41%
11	2.00%
12	2.00%
13	1.67%
14	2.15%
15	2.08%
16	2.08%
17	1.88%
18	1.71%
19	2.08%
20	1.85%
21	1.71%
22	1.50%
23	1.79%
24	1.73%
25	1.47%

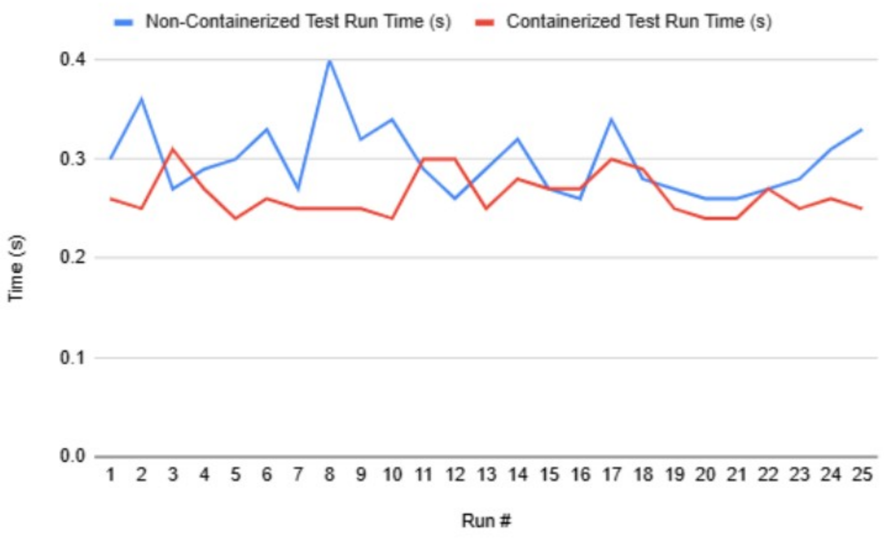

Fig. 1. Graph of test run times

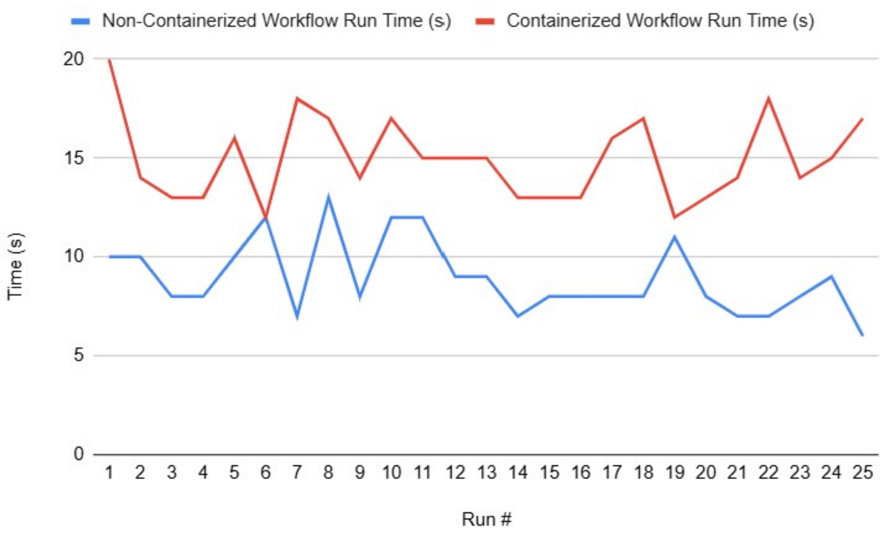

Fig. 2. Graph of workflow times

4.4 Statistical Analysis

We collected 4 columns of data:

1. Non-Containerized Test Run Time (s)
   – Mean = 0.2988s
   – Median = 0.29s
   – Mode = 0.27s
2. Non-Containerized Workflow Run Time (s)
   – Mean = 8.92s
   – Median = 8s
   – Mode = 8s
3. Containerized Test Run Time (s)
   – Mean = 0.264s
   – Median = 0.26s
   – Mode = 0.25s
4. Containerized Workflow Run Time (s)
   – Mean = 14.96s
   – Median = 15s
   – Mode = 13s

Descriptive Statistical Analysis The descriptive statistics (mean, median, and mode) suggest that the containerized tests ran faster on average than the non-containerized tests, and that the containerized workflow time was significantly different from the non-containerized workflow time. These suggestions will be further supported by our inferential statistical analysis.

Inferential Statistics We ran 2 statistical comparisons:

1. Non-Containerized Test Run Time (s) vs. Containerized Test Run Time (s)
2. Non-Containerized Workflow Run Time (s) vs Containerized Workflow Run Time (s)

For the 1st comparison, we ran a paired t-test using a p-value of 0.5 and the following hypotheses:

– Null Hypothesis: Non-containerized test time is not longer than containerized test time.
– Alternative Hypothesis: Non-containerized test time is longer than containerized test time.

This resulted in the following t-test output:

```
          Paired t-test

data: data$Non.Containerized.Test.Run.Time..s. and
    ↪ data$Containerized.Test.Run.Time..s.
t = 3.8343, df = 24, p-value = 4e-04
```

```
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 0.01927208 Inf
sample estimates:
mean of the differences
               0.0348
```

This output shows that p-value is far less than alpha, which was 0.05. This suggests that we can reject the null hypothesis and conclude that non-containerized test times are significantly longer than containerized test times.

For the 2nd comparison, we ran a paired t-test using a p-value of 0.05 and the following hypotheses:

– Null Hypothesis: Non-containerized workflow time is not different from the containerized workflow time.
– Alternative Hypothesis: Non-containerized workflow time is different from the containerized workflow time.

This resulted in the following t-test output:

```
       Paired t-test

data: data$Non.Containerized.Workflow.Run.Time..s. and
    ↪ data$Containerized.Workflow.Run.Time..s.
t = -10.763, df = 24, p-value = 1.148e-10
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -7.198238 -4.881762
sample estimates:
mean of the differences
                -6.04
```

This output shows that p-value is far less than alpha, which was 0.05. This suggests that we can reject the null hypothesis and conclude that non-containerized workflow times are significantly different from containerized workflow times.

5 Discussion

5.1 Discussing Results

The findings highlight the dual role of containerization in enhancing efficiency and introducing new complexities. CI/CD pipelines with Docker containers show clear efficiency gains, with reductions in test execution time and improved resource utilization through orchestration. However, implementing such a framework demands an increased focus on resource management and security protocols. The increased overhead in configuring and managing container orchestration, especially with Kubernetes, underscores the need for dedicated infrastructure expertise in maintaining a secure, high-performing CI/CD system.

The trade-offs between efficiency gains and the complexity of containerized test management underscore the importance of strategic planning in containerization efforts. Organizations must evaluate their specific testing requirements and system resources when deciding to implement containers within their CI/CD framework.

5.2 Connecting to Literature Review

Our Solution vs. Parallel Testing Our findings give a glimpse of the benefits of using containerized vs. non-containerized workflows for test automation, particularly in reducing total workflow runtime. However, our study did not incorporate parallel testing within the containerized CI/CD pipeline. Referring back to Majumder's discussion about Parallel Testing in Dockerized CI/CD Frameworks in our Literature Review, parallel execution of test suites was identified as a key advantage of containerization [1]. This change significantly reduces execution time by enabling multiple tests to run simultaneously in isolated environments. If we had implemented parallel testing in our containerized pipeline, the increase in efficiency compared to the non-containerized workflow would likely have been much more significant. The ability to execute tests concurrently could have further reduced test runtimes and improved the scalability advantages of containers. Future studies could explore the inclusion of parallel testing to fully leverage the capabilities of containerization and validate its impact on large-scale test automation workflows.

Thoughts on Resource Overhead As discussed in Section 3.1, under "Challenges and Limitations of Containerization in CI/CD", one of the challenges with implementing containerization in CI/CD workflows is limiting resource overhead. Especially in large-scale software, containerized pipelines require a great amount of resources such as CPU and memory. Modern and currently-in-development software services such as GitHub Actions help mitigate this overhead. Github Actions allow developers to configure job concurrency, set resource limits, and use caching mechanisms to reduce redundant computations. These abilities help limit the challenges of resource overhead. Ultimately, as CI/CD technology continues to improve, this will enable teams and companies with access to less resources and lower budgets to implement containerized CI/CD workflows.

Thoughts on the Skill Gap Another challenge that was discussed in Section 3.1 was the skill gap that is required in the set up and maintenance of a containerized CI/CD workflow. As CI/CD and containerizing technologies continue to evolve, a concern in the industry is that the cost of training and finding specialized developers may increase. However, tools like GitHub Actions simplify the process by removing much of the low-level work required to set up a containerized CI/CD workflow. For instance, we were able to set up a simple non-containerized CI/CD workflow in less than 30 minutes.

Of course, this process gets longer when you include containerization. However, the same argument can be made for tools like Docker. As containerization rapidly becomes more standardized in the software industry, companies like Docker will make processes that streamline the process of setting up a container and integrating it into a CI/CD workflow. We may very well see more and more templates in GitHub "Actions" that make the creation of Docker files much more straightforward. These pre-built workflows are just one of the examples of how modern software tools mitigate the challenges of the skill gap required for companies to adopt containerized CI/CD workflows.

Summary of Challenges After Findings It is clear that the sub-fields of containerization and CI/CD are rapidly growing. Thus, even though companies are discovering new challenges and limitations related to the technology, DevOps companies are continuously releasing new products and updates to old products that include fixes and improvements.

6 Conclusion

Containerizing CI/CD test automation frameworks provides significant benefits in terms of efficiency, scalability, and consistency across development environments. However, it also introduces challenges related to resource consumption, security, and operational complexity. Despite these challenges, the advantages of containerization, particularly for parallel execution and resource flexibility, suggest that containerized CI/CD systems can offer a substantial return on investment for organizations looking to streamline their development workflows. Future research should further explore optimization techniques and best practices for managing containerized CI/CD systems at scale, particularly focusing on security and automation.

References

1. Majumder, R.: Maximizing Efficiency: Automated Software Testing with CI. Master's thesis, Ohio University (2024)
2. Mustyala, A.: CI/CD pipelines in kubernetes: Accelerating software development and deployment. EPH- International Journal of Science And Engineering **8**(3) (Feb 2022). `https://doi.org/10.53555/ephijse.v8i3.238`
3. Raj, A.: Containerization and its impact on DevOps practices. International Journal of AdvancedandInnovativeResearch **10**(1) (Aug 2024)

Appendices

A.1 Dockerfile

```
# Start with a lightweight Python image
FROM python:3.10-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the requirements file into the container
COPY requirements.txt /app/

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy only the Research-paper-code directory and necessary files
    ↪ into the container
COPY Research-paper-code /app/Research-paper-code
COPY requirements.txt /app/

# Set the working directory to Research-paper-code
WORKDIR /app/Research-paper-code

# Run pytest --cov on Research-paper-code directory with coverage
CMD ["pytest", ".", "--cov"]
```

A.2 requirements.txt

```
pytest==8.3.3
pytest-cov==6.0.0
```

A.3 python-app.yaml

```
name: PyTest

on:
  push:
    branches:
      - main

permissions:
  contents: read

jobs:
  run-tests-uncontainerized:
    name: Run Tests uncontainerized

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        if [ -f requirements.txt ]; then pip install -r
            ↪ requirements.txt; fi
    - name: Test with pytest
      run: |
        pytest 'Research-paper-code/' --cov

  run-tests-docker:
    name: run tests in a docker container
    runs-on: ubuntu-latest

    steps:
    # Step 1: Check out the repository
    - name: Checkout repository
      uses: actions/checkout@v3

    # Step 2: Log in to Docker Hub (if pushing image is needed)
    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    # Step 3: Build the Docker image
    - name: Build Docker image
      run: |
        docker build -t myapp:latest .

    # Step 4: Run the Docker container
    - name: Run Docker container
      run: |
        docker run --rm myapp:latest
```

A.4 Test Results

Un-containerized:

```
---- coverage: platform linux, python 3.10.15-final-0 ----
Name Stmts Miss Cover
----------------------------------------------------------
scientific_calculator.py 69 3 96%
test_scientific_calculator.py 194 3 98%
----------------------------------------------------------
TOTAL 263 6 98%
```

Containerized:

```
---- coverage: platform linux, python 3.10.15-final-0 ----
Name Stmts Miss Cover
---------------------------------------------------
scientific_calculator.py 69 3 96%
test_scientific_calculator.py 194 3 98%
---------------------------------------------------
TOTAL 263 6 98%
```

You can see there is no difference in the results