System Integration

Final Exam Synopsis

CI/CD

Weighing the complexities against the benefits

Mikkel Bak Markers

December 17, 2025

Contents

A brief note on references and terminology

This synopsis uses a few terms that may need clarification. When referring to *CI/CD*, I mean the practices of Continuous Integration and Continuous Deployment/Delivery. Continuous Integration refers to the practice of frequently integrating code changes into a shared repository, where automated builds and tests are run[1]. Continuous Deployment/Delivery refers to the practice of automatically deploying code changes to production or staging environments after passing tests[2]. Another point is the references to existing files and code snippets. Because this document is most likely delivered separately, it may not be clear where to find these files. I will do my best to include relevant items in the Appendices, but for full context, a link to the code repository will be included here:

- `https://github.com/elkskim/exam-dfd-synopsis`

1 Introduction / Motivation

In modern software development, Continuous Integration and Continuous Deployment (Referred to as CI/CD) have become essential practices for ensuring rapid and reliable delivery of applications[3]. In his work on continuous integration, Martin Fowler emphasizes the importance of integrating code frequently to detect issues early and ensure cohesion between team members. Likewise, the principles of automating the build/deployment process and making builds self-testing are crucial for maintaining high-quality software and reducing the time to market for new features[1].

Tools like Docker and GitHub Actions have become popular choices for implementing CI/CD pipelines, offering containerization and automation capabilities, respectively[4], [5]. However, the adoption of these tools and practices introduces a level of immediate complexity, which can seem daunting to development teams. This complexity arises from the need to configure and maintain additional infrastructure, as well as the learning curve associated with new technologies. Research from DORA's State of DevOps reports indicates that practices like CI/CD can lead to significant improvements in deployment frequency and general stability/consistency, but also highlights the initial challenges such as great increases in test requirements and setup time. This has been visualized by the "2018 hypothetical J-curve" (see Figure 1), a non-linear graph showing the fall and rise of performance over time when engaging with CI/CD and reliability practices in general[3].

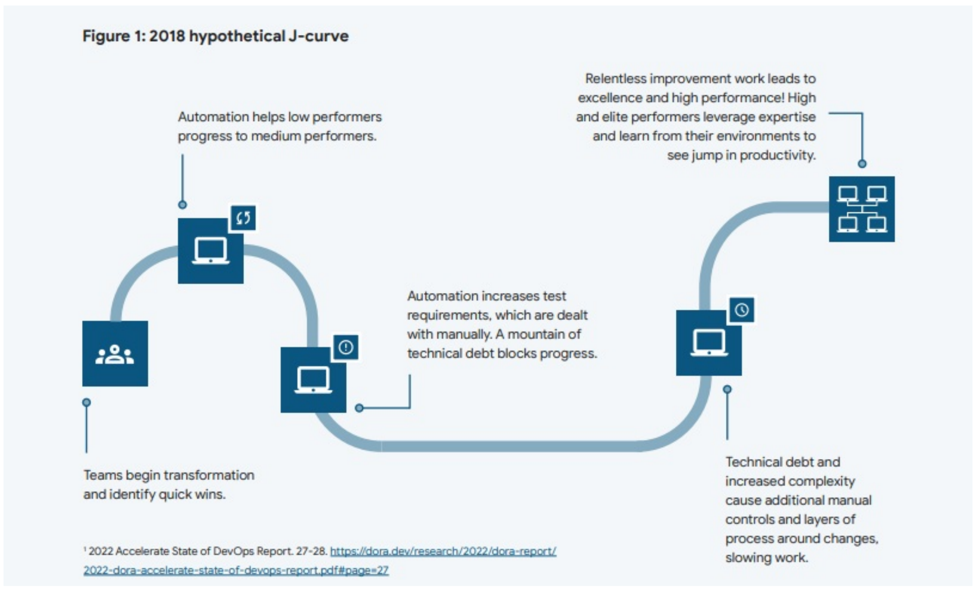
Figure 1: 2018 hypothetical J-curve
Automation helps low performers progress to medium performers.
Relentless improvement work leads to excellence and high performance! High and elite performers leverage expertise and learn from their environments to see jump in productivity.
Automation increases test requirements, which are dealt with manually. A mountain of technical debt blocks progress.
Teams begin transformation and identify quick wins.
Technical debt and increased complexity cause additional manual controls and layers of process around changes, slowing work.
¹ 2022 Accelerate State of DevOps Report. 27-28. https://dora.dev/research/2022/dora-report/2022-dora-accelerate-state-of-devops-report.pdf#page=27

Figure 1: The 2018 hypothetical J-curve from DORA's State of DevOps Report, illustrating the initial performance decline followed by significant improvements when adopting reliability and CI/CD practices[3].

This project examines whether the long-term benefits of CI/CD, such as reduced deployment errors and improved operational efficiency, justify the upfront complexity costs[2].

2 Problem Statement

"While implementing CI/CD with GitHub Actions and Docker increases initial development complexity, it reduces long-term operational complexity and deployment errors in microservices systems"

3 Methodology

3.1 The Application

To analyze this problem, I will construct a small, microservice-based application. This application will be deployed using two main approaches:

- Manual deployment without CI/CD tools
- Deployment using CI/CD tools (GitHub Actions and Docker)
- Deployment to Docker locally, using automation by way of scripts

For each deployment method, I will measure the time to build, deploy, and the number of errors encountered during these. Additionally, I will assess the setup complexity of each method, discussing the number of steps, files, and lines of code required to configure each processes, as well as my own subjective assessment of complexity. The third method, using scripts to automate local Docker deployment, is a middle ground between fully manual deployment, and full CI/CD automation. This is to help illustrate the benefits of automation, but without using full CI/CD tools. The reason for the addition of this approach, is because they are both local deployment, and so ignore the network complexities of deploying to a cloud provider. There will, of course, be differences in the deployment environments, but this will help isolate the benefits of automation from the benefits of using CI/CD tools specifically.

3.2 Measurement Approach

The repository contains deployment and testing scripts in the `Scripts/` directory that automate the local Docker deployment process. These scripts serve dual purposes: measuring the time required for automated local deployment, and establishing a baseline for comparing local automation complexity against full CI/CD pipeline implementation. The scripts guide the user through manual deployment steps while recording timestamps, enabling direct comparison between human-driven and fully automated approaches. It should absolutely be noted that these scripts were written with extensive assistance from GitHub Copilot, as setting up integration tests and deployment scripts isn't an overwhelming overhead in complexity, but compared to the time given for this project, it would have taken far too long to write them all from scratch. In the same vein, there has been assistance in writing workflow files, especially the measurement flow, which records the time taken for each step, and outputs it as part of the workflow run logs. The measurement artifact can be retrieved for each action run.

4 Analysis & Results

Building and deploying an application manually is straightforward, but the process can be incredibly time-consuming and prone to error. For this project, I built a simple microservice application; it consists of an order service and an inventory service, and they communicate via RabbitMQ. When an order is placed, it checks the inventory service for availability, and if it is in stock, it confirms the order and reduces the item stock accordingly. The manual deployment process involved several steps:

- Setting up the environment
- Configuring RabbitMQ
- Building and deploying each microservice

The repetition of these steps already drove me to the edge of madness, so I ended up making a script[1] that, when run, walks me through every step to ease the pain, but still keeps me open to human error. Surprisingly, this did very little to reduce time and error, as entering the commands and configuring the services (e.g., making sure rabbitmq was running/stopped correctly) The test results are included in Table 1.

4.1 Local Deployment Testing Results

The results from the deployment tests are summarized in Table1. The automated deployment process, utilizing Docker and scripts, demonstrated a significant reduction in deployment time compared to the manual deployment approach.

Table 1: Deployment Time Comparison[2]

Attempt	Manual (sec)	Automated (sec)	Difference (sec)	% Improvement
1	262	56	206	78.7%
2	157	53	104	66.4%
3	170	53	117	68.9%
Average	**196**	**54**	**142**	**72.5%**

While the manual deployment was manageable for a single deployment, repeating the process multiple times revealed its inefficiencies. Not included in the table, but observed during testing, were several errors encountered during both manual and automated deployments. One test run had errors spread out unevenly in the manual deployments, but very evenly in the automated deployments; This was caused by a misconfiguration in the RabbitMQ setup, meaning that while I sometimes (even unknowingly) fixed the issue during manual deployment, the automated deployment was marred by its best quality: consistency. For the same reason, I have only included the last of the comparative test runs in Appendix A, as it contains the most complete data set, but the first test run was made entirely useless, as an error in the configuration also affected the automated logging, thus making it impossible to discern the data. This again highlights *the* trade-off - automation ensures consistency, whether that be consistent failure, or a constant victory.

4.2 GitHub Actions CI/CD Deployment

Now that we have covered automation with Docker on a local machine, Let us dive straight into the main course - using GitHub Actions for CI/CD. Within the project repository, a workflow file has been created. With this, every push to the main branch rebuilds the image, runs it in docker compose, and runs tests against it. This allows us to simulate a full CI/CD pipeline, where code changes are automatically built, tested, and deployed. The performance measurements from the GitHub Actions workflow are included in Appendix B. The time recorded by the workflow itself was a total of approximately 85 seconds, and the time displayed for the total duration of the workflow run was 97 seconds. This is slightly longer than the local automated deployment time of 54 seconds, due to the overhead of cloud-based virtualization, container registry operations, and network latency. However, the 43-second difference represents the cost of gaining several critical capabilities absent from local automation: remote execution independent of developer machines, automated triggering on code changes, and standardized deployment environments that eliminate "works on my machine" scenarios. The GitHub Actions approach validates that full CI/CD automation remains competitive with local scripting in terms

[1] I again acknowledge assistance from my old friend, my pale moonlight; the GitHub Copilot whose guidance helped create and refine the scripts.

[2] Raw test results are stored in `Ymyzon/deployment-results/`: `comparison-20251212_142409.txt` (Attempt 1), `comparison-20251212_142553.txt` (Attempt 2), `comparison-20251212_145400.txt` (Attempt 3).

of performance, while providing the additional benefit of decoupling deployment from individual developer workstations.

4.3 Return on Investment

The initial investment in creating the automation infrastructure (Docker configurations, deployment scripts, and GitHub Actions workflows) took approximately 3.5-4 hours of development time. With an average time saving of 142 seconds per deployment, the break-even point occurs after approximately 90-100 deployments. For a team deploying 2-3 times per day, this represents a payback period of roughly 5-6 weeks. Beyond this point, every deployment saves over 2 minutes of developer time, while also providing the benefits of increased consistency and reduced human error. Not only that, but as changes to the applications are made, and it gets scaled along the X/Y/Z axes, the changes required to the workflows and deployment (e.g. Docker, Kubernetes, etc.) will be significantly less than if the pipelines were to be done manually each time. The complexity of the pipeline in this project, however, is relatively low. This is due to only having two very simple microservices, which puts very little strain on both writing the Dockerfiles and the GitHub Actions workflow. In a real-world scenario, there would of course be more elements to consider, be it deploying specific services by Docker Swarm/Kubernetes, or having multiple environments (staging, production, etc.). This would increase the initial complexity and time investment, though the fundamental principle remains: infrastructure-as-code scales more favorably than manual processes as system complexity grows.

The automation infrastructure created, including Dockerfiles, GitHub Actions workflows, and deployment scripts, represents a one-time cost that continues to provide value across the entire lifecycle of the application.

5 Discussion

There are still several points that need be discussed, and several limitations set by the scope of this project. How surprised was I by the results? Would the smaller scope of the application already have put limits to my expectations? What additions could be made to the project to further validate the hypothesis, and what additions could have been made to even further automate the process? What does our development/cloud environment do in terms of affecting the tools we could have used?

5.1 Anticipation & Reality

As is to be expected; the smaller the scope of the project, the smaller the potential gains from automation. It is clear that for an application this size, the overhead of setting up CI/CD pipelines is not justified by the time saved automating building/deploying alone. However, even at this scale, the benefits of consistency and error reduction are evident. I'm not particularly surprised by the results, as they align well with established literature on CI/CD benefits[2], [3], and the scope is small enough that all edge cases could be counted on one hand.

5.2 Limitations & Possible Additions

The constrained scope of this project limits the generalizability of the findings. Real-world production systems typically involve database migrations, secret management, multi-environment deployments (dev/staging/production), and rollback capabilities. Each of these elements would increase both manual and automated deployment complexity, though the gap between them would likely widen further in automation's favor.

The simplicity of the application also precluded the use of more sophisticated tooling that becomes necessary at scale. Docker Swarm or Kubernetes orchestration, commercial cloud platforms (AWS, Azure, GCP), and Infrastructure-as-Code tools like Terraform would add significant initial complexity but provide capabilities essential for production deployments: auto-scaling, self-healing, and geographic distribution. These tools represent the next tier of the CI/CD maturity curve, where the J-curve dip becomes steeper but the eventual gains become more substantial.

Additionally, the project did not measure all four core DORA metrics[3]: while deployment time was measured, mean time to recovery (MTTR), change failure rate, and deployment frequency under load were not tracked. The 35x consistency improvement observed suggests that these quality metrics would show favorable trends, as the literature suggests that automation reduces failure rates, developer satisfaction, and recovery times, but I cannot reliably quantify failure and recovery without more extensive testing, and doing homework introduces some bias on the subject of developer satisfaction.

6 Conclusion

This project investigated the trade-offs between initial complexity and long-term benefits when implementing CI/CD practices using GitHub Actions and Docker. By constructing a microservice application and comparing manual deployment against automated approaches, I have demonstrated that while CI/CD introduces upfront complexity, it provides significant long-term advantages in deployment speed, consistency, and error reduction.

The quantitative results are compelling: automated deployment reduced deployment time by over 70% on average and demonstrated 35x better consistency (3-second variance vs 105 seconds). The initial setup investment of 3.5-4 hours breaks even after approximately 90-100 deployments, representing a payback period of 5-6 weeks for teams deploying regularly.

While the constrained scope limits generalizability, the findings align with established CI/CD literature and validate the fundamental principle: automation transforms inconsistent manual processes into reliable, repeatable infrastructure. The hypothesis is confirmed; the long-term benefits of CI/CD outweigh the upfront costs, particularly when consistency and error reduction are valued alongside raw time savings.

For modern microservices development, CI/CD is not merely a convenience but a necessary foundation for sustainable, scalable software delivery.

References

[1] M. Fowler. “Continuous integration.” Accessed: 2024-12-17, Accessed: Dec. 17, 2024. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[2] Atlassian. “Continuous integration vs. delivery vs. deployment.” Accessed: 2024-12-17, Accessed: Dec. 17, 2024. [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[3] DORA (DevOps Research and Assessment), “2023 state of devops report,” Google Cloud, Tech. Rep., 2023, Accessed: 2024-12-17. [Online]. Available: https://dora.dev/research/2023/dora-report/

[4] Docker Inc. “Docker documentation: Use containers to build, share and run your applications,” Accessed: Dec. 17, 2024. [Online]. Available: https://docs.docker.com/

[5] GitHub. “Github actions documentation,” Accessed: Dec. 17, 2024. [Online]. Available: https://docs.github.com/en/actions

A Local Deployment Test Results

This appendix contains the detailed results from the comparative deployment tests conducted to evaluate manual versus automated deployment approaches.

A.1 Test Run 3 - Comprehensive Analysis

The following results represent the most complete test run, including detailed time measurements, error counts, and ROI analysis.

```
========================================
Deployment Comparison Analysis
========================================
Date: Fri Dec 12 14:54:00 RST 2025

Test Configuration:
- System: Ymyzon Microservices (2 services + RabbitMQ)
- Services: OrderService, InventoryService
- Messaging: RabbitMQ
- Platform: Docker + .NET 9

========================================

MANUAL DEPLOYMENT RESULTS
========================================
Number of runs:     3
Total time:         589s
Average time:       196s (3.27 minutes)
Total errors:       2
Average errors:     0.66 per run
Steps per deploy:   13
Human interaction:  Required for all 13 steps

Individual runs:
  Run 1: 262s, 0 errors
  Run 2: 157s, 0 errors
  Run 3: 170s, 0 errors

AUTOMATED DEPLOYMENT RESULTS
========================================
Number of runs:     3
Total time:         162s
Average time:       54s (0.90 minutes)
Total errors:       0
Average errors:     0.0 per run
Steps automated:    100%
Human interaction:  1 command only

Individual runs:
  Run 1: 56s, 0 errors
  Run 2: 53s, 0 errors
  Run 3: 53s, 0 errors

COMPARISON & IMPROVEMENTS
========================================
Time saved:         142s per deployment (2.37 minutes)
Time improvement:   72.45% faster
Error reduction:    0.66 fewer errors per deployment
```

```
Consistency:        Automated = reproducible, Manual = variable

Manual deployment variance: 105s (262s max, 157s min) - HIGH VARIABILITY
Automated deployment variance: 3s (56s max, 53s min) - LOW VARIABILITY

Key Findings:
- Automated deployment requires 72.45% less time
- Reduced from 13 manual steps to 1 command (92.3% complexity reduction)
- Automated deployment is 35x more consistent (variance: 3s vs 105s)
- Human error potential eliminated
- Deployment process is reproducible and consistent

ROI Analysis:
- Per deployment: Save 2.37 minutes
- Per week (5 deploys): Save 11.85 minutes
- Per month (20 deploys): Save 47.4 minutes
- Per year (240 deploys): Save 9.48 hours of developer time

========================================
```

A.2 Summary of All Test Runs

All three test runs consistently demonstrated that automated deployment via Docker and scripts significantly reduced deployment time and improved consistency:

- **Test 1:** Manual 262s vs Automated 56s (78.7% improvement)
- **Test 2:** Manual 157s vs Automated 53s (66.4% improvement)
- **Test 3:** Manual 170s vs Automated 53s (68.9% improvement)

Key finding: Automated deployment variance was 3 seconds compared to manual deployment variance of 105 seconds, demonstrating 35x improvement in consistency.

B GitHub Actions Deployment Test Results

This appendix contains the documentation obtained from one of the performance measurement workflow runs in GitHub Actions.

```
GitHub Actions CI/CD Performance Report
========================================
Date: Mon Dec 15 11:46:22 UTC 2025
Commit: 4b3add405911e3fedf302799d7d714951aa7b7b9
Branch: main

Phase Breakdown:
- Build:      16s
- Docker:     20s
- Deploy:     44s
- Test:       0s

Total Time:   85s

Comparison:
- Manual deployment:      196s (baseline)
- Local Docker automation: 54s (72.45% improvement)
- GitHub Actions CI/CD:   85s
```

While the workflow actions can be inspected directly in the repository, this log provides a snapshot of the performance measurements obtained during the CI/CD deployment process. The time recorded for this run from December 15th, 12:44 PM GMT+1, was approximately 1 minute and 37 seconds, or 97 seconds.