Systemintegration

Eksamenssynopsis

CI/CD

Afvejning af kompleksitet mod fordele

Mikkel Bak Markers

17. december 2025

Indhold

En kort note om referencer og terminologi

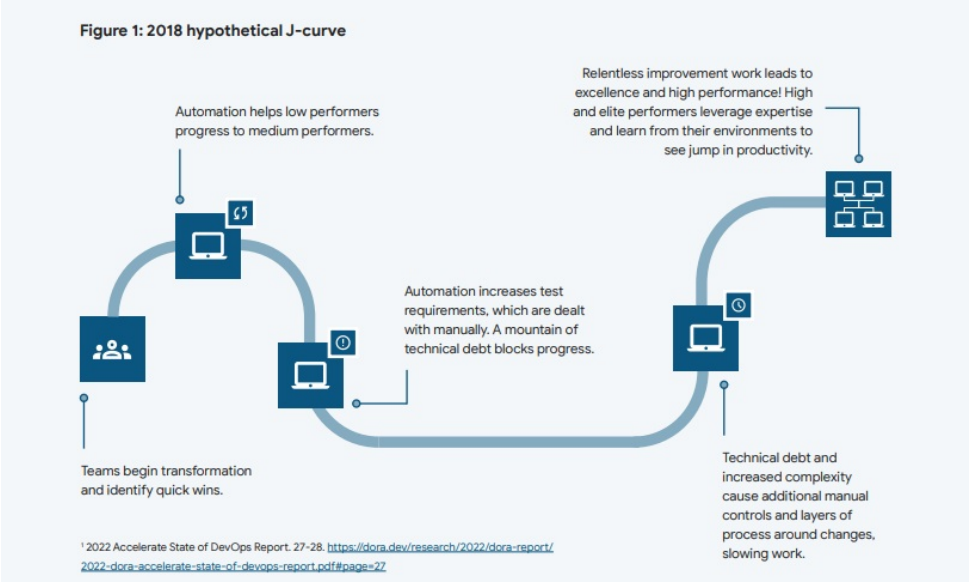
Denne synopsis anvender en række termer, som måske kræver afklaring. Når jeg refererer til *CI/CD*, mener jeg praksisserne Continuous Integration (kontinuerlig integration) og Continuous Deployment/Delivery (kontinuerlig deployment/levering). Continuous Integration refererer til praksis med hyppig integration af kodeændringer i et delt repository, hvor automatiserede builds og tests køres[1]. Continuous Deployment/Delivery refererer til praksis med automatisk deployment af kodeændringer til produktions- eller stagingmiljøer efter beståede tests[2]. Et andet punkt er referencerne til eksisterende filer og kodeuddrag. Da dette dokument sandsynligvis leveres separat, er det måske ikke klart, hvor disse filer kan findes. Jeg vil gøre mit bedste for at inkludere relevante elementer i bilagene, men for fuld kontekst inkluderes et link til koderepositoriet her:

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

1 Indledning / Motivation

I moderne softwareudvikling er Continuous Integration og Continuous Deployment (refereret til som CI/CD) blevet essentielle praksisser for at sikre hurtig og pålidelig levering af applikationer[3]. I sit arbejde om kontinuerlig integration understreger Martin Fowler vigtigheden af at integrere kode hyppigt for at opdage problemer tidligt og sikre samhørighed mellem teammedlemmer. Ligeledes er principperne om at automatisere build/deployment-processen og gøre builds selvtestende afgørende for at opretholde software af høj kvalitet og reducere tiden til markedet for nye features[1].

Værktøjer som Docker og GitHub Actions er blevet populære valg til implementering af CI/CD-pipelines, der tilbyder henholdsvis containerisering og automatiseringskapaciteter[4], [5]. Adoptionen af disse værktøjer og praksisser introducerer dog et niveau af umiddelbar kompleksitet, som kan virke skræmmende for udviklingsteams. Denne kompleksitet opstår fra behovet for at konfigurere og vedligeholde yderligere infrastruktur, samt indlæringskurven forbundet med nye teknologier. Forskning fra DORA's State of DevOps-rapporter indikerer, at praksisser som CI/CD kan føre til betydelige forbedringer i deployment-frekvens og generel stabilitet/konsistens, men fremhæver også de indledende udfordringer såsom store stigninger i testkrav og setup-tid. Dette er blevet visualiseret ved "2018 hypothetical J-curve" (se Figur 1), en ikke-lineær graf der viser faldet og stigningen i performance over tid når man engagerer sig i CI/CD og pålidelighedspraksisser generelt[3].



Figur 1: Den hypotetiske J-kurve fra 2018 fra DORA's State of DevOps Report, der illustrerer det indledende fald i performance efterfulgt af betydelige forbedringer ved adoption af pålidelighed- og CI/CD-praksisser[3].

Dette projekt undersøger hvorvidt de langsigtede fordele ved CI/CD, såsom reducerede deployment-fejl og forbedret operationel effektivitet, retfærdiggør de indledende kompleksitetsomkostninger[2].

2 Problemformulering

"Selvom implementering af CI/CD med GitHub Actions og Docker øger den indledende udviklingsmæssige kompleksitet, reducerer det langsigtede operationel kompleksitet og deployment-fejl i microservice-systemer"

3 Metode

Følgende beskriver den tilgang jeg vil tage for at analysere problemformuleringen. [1]

3.1 Applikationen

For at analysere dette problem vil jeg konstruere en lille, microservice-baseret applikation. Denne applikation vil blive deployeret ved hjælp af to hovedtilgange:

- Manuel deployment uden CI/CD-værktøjer
- Deployment ved hjælp af CI/CD-værktøjer (GitHub Actions og Docker)
- Deployment til Docker lokalt, ved hjælp af automatisering via scripts

For hver deployment-metode vil jeg måle tiden til at bygge, deploye, og antallet af fejl der opstår under disse. Derudover vil jeg vurdere setup-kompleksiteten af hver metode, diskutere antallet af trin, filer og kodelinjer der kræves for at konfigurere hver proces, samt min egen subjektive vurdering af kompleksitet. Den tredje metode, der bruger scripts til at automatisere lokal Docker deployment, er en mellemvej mellem fuldstændig manuel deployment og fuld CI/CD-automatisering. Dette skal hjælpe med at illustrere fordelene ved automatisering, men uden at bruge fulde CI/CD-værktøjer. Årsagen til tilføjelsen af denne tilgang er, at de begge er lokal deployment, og derfor ignorerer netværkskompleksiteten ved deployment til en cloud-udbyder. Der vil selvfølgelig være forskelle i deployment-miljøerne, men dette vil hjælpe med at isolere fordelene ved automatisering fra fordelene ved at bruge CI/CD-værktøjer specifikt.

3.2 Måletilgang

Repositoriet indeholder deployment- og testscripts i `Scripts/`-mappen der automatiserer den lokale Docker deployment-proces. Disse scripts tjener dobbelte formål: at måle tiden der kræves til automatiseret lokal deployment, og at etablere en baseline for sammenligning af lokal automatiseringskompleksitet mod fuld CI/CD-pipeline implementering. Scriptsene guider brugeren gennem manuelle deployment-trin mens de registrerer tidsstempler, hvilket muliggør direkte sammenligning mellem menneskedrevne og fuldt automatiserede tilgange. Det skal absolut bemærkes, at disse scripts blev skrevet med omfattende assistance fra GitHub Copilot, da opsætning af integrationstests og deployment-scripts ikke er en overvældende overhead i kompleksitet, men sammenlignet med tiden givet til dette projekt, ville det have taget alt for lang tid at skrive dem alle fra bunden. I samme ånd har der været assistance til at skrive workflow-filer, især måle-flowet, som registrerer tiden der tages for hvert trin, og outputter det som en del af workflow-kørselslogs. Måleartefakten kan hentes for hver action-kørsel.

4 Analyse & Resultater

At bygge og deploye en applikation manuelt er ligetil, men processen kan være utrolig tidskrævende og fejlbehæftet. Til dette projekt byggede jeg en simpel microservice-applikation; den består af en order service og en inventory service, og de kommunikerer via RabbitMQ. Når en ordre placeres, tjekker den inventory service for tilgængelighed, og hvis varen er på lager, bekræfter den ordren og reducerer varelageret tilsvarende. Den manuelle deployment-proces involverede flere trin:

- Opsætning af miljøet

[1] Det skal bemærkes at hele dette project, inkl. research, kodeimplementering, testscripts, og den originale skrivning af denne synopsis, blev udført på engelsk, i overensstemmelse med kursusmaterialer og industristandard. Visse tekniske nuancer og fagtermer kan gå tabt i oversættelsen til dansk. Især Metode-afsnittet er mærket af dette, da det beskriver specifikke implementeringer og værktøjer.

- Konfigurering af RabbitMQ
- Bygning og deployment af hver microservice

Gentagelsen af disse trin drev mig allerede til vanviddets rand, så jeg endte med at lave et script[2] der, når det køres, guider mig gennem hvert trin for at lindre smerten, men stadig holder mig åben for menneskelige fejl. Overraskende gjorde dette meget lidt for at reducere tid og fejl, da indtastning af kommandoer og konfigurering af services (f.eks. at sikre at rabbitmq kørte/stoppede korrekt) Testresultaterne er inkluderet i Tabel 1.

4.1 Resultater af Lokal Deployment-test

Resultaterne fra deployment-testene er opsummeret i Tabel1. Den automatiserede deployment-proces, der benytter Docker og scripts, demonstrerede en betydelig reduktion i deployment-tid sammenlignet med den manuelle deployment-tilgang.

Tabel 1: Sammenligning af Deployment-tid[3]

Forsøg	**Manuel (sek)**	**Automatiseret (sek)**	**Forskel (sek)**	**% Forbedring**
1	262	56	206	78.7%
2	157	53	104	66.4%
3	170	53	117	68.9%
Gennemsnit	**196**	**54**	**142**	**72.5%**

Selvom den manuelle deployment var håndterbar for en enkelt deployment, afslørede gentagelse af processen dens ineffektiviteter. Ikke inkluderet i tabellen, men observeret under test, var flere fejl der opstod under både manuel og automatiseret deployment. En testkørsel havde fejl spredt ud ujævnt i de manuelle deployments, men meget jævnt i de automatiserede deployments; Dette blev forårsaget af en fejlkonfiguration i RabbitMQ-opsætningen, hvilket betyder at mens jeg nogle gange (selv ubevidst) fixede problemet under manuel deployment, var den automatiserede deployment plaget af dens bedste kvalitet: konsistens. Af samme grund har jeg kun inkluderet den sidste af de komparative testkørsler i Bilag A, da den indeholder det mest komplette datasæt, men den første testkørsel blev gjort helt ubrugelig, da en fejl i konfigurationen også påvirkede den automatiserede logging, hvilket gjorde det umuligt at skelne dataene. Dette fremhæver igen *afvejningen* - automatisering sikrer konsistens, hvad enten det er konsistent fejl, eller en konstant sejr.

4.2 GitHub Actions CI/CD Deployment

Nu hvor vi har dækket automatisering med Docker på en lokal maskine, lad os dykke direkte ned i hovedretten - at bruge GitHub Actions til CI/CD. Inden for projekt-repositoriet er en workflow-fil blevet oprettet. Med denne, genopbygger hvert push til main-branchen imaget, kører det i docker compose, og kører tests mod det. Dette gør det muligt for os at simulere en fuld CI/CD-pipeline, hvor kodeændringer automatisk bygges, testes og deployes. Performance-målingerne fra GitHub Actions workflow er inkluderet i Bilag B. Tiden registreret af workflow'et selv var i alt cirka 85 sekunder, og tiden vist for den totale varighed af workflow-kørslen var 97 sekunder. Dette er lidt længere end den lokale automatiserede deployment-tid på 54 sekunder, på grund af overhead fra cloud-baseret virtualisering, container registry-operationer, og netværks-latency. Dog repræsenterer de 43 sekunders forskel prisen for at opnå flere kritiske kapaciteter

[2]Jeg anerkender igen assistance fra min gamle ven, mit blege månelys; GitHub Copilot hvis vejledning hjalp med at skabe og forfine scriptsene.

[3]Rå testresultater er gemt i `Ymyzon/deployment-results/`: `comparison-20251212_142409.txt` (Forsøg 1), `comparison-20251212_142553.txt` (Forsøg 2), `comparison-20251212_145400.txt` (Forsøg 3).

fraværende fra lokal automatisering: fjerneksekvering uafhængig af udvikler-maskiner, automatisk triggering ved kodeændringer, og standardiserede deployment-miljøer der eliminerer "works on my machine"-scenarier. GitHub Actions-tilgangen validerer at fuld CI/CD-automatisering forbliver konkurrencedygtig med lokal scripting med hensyn til performance, mens den tilbyder den ekstra fordel ved at afkoble deployment fra individuelle udvikler-workstations.

4.3 Return on Investment

Den indledende investering i at skabe automatiseringsinfrastrukturen (Docker-konfigurationer, deployment-scripts og GitHub Actions workflows) tog cirka 3,5-4 timers udviklingstid. Med en gennemsnitlig tidsbesparelse på 142 sekunder per deployment, indtræffer break-even-punktet efter cirka 90-100 deployments. For et team der deployer 2-3 gange om dagen, repræsenterer dette en tilbagebetalingsperiode på cirka 5-6 uger. Ud over dette punkt sparer hver deployment over 2 minutters udviklertid, mens den også giver fordelene ved øget konsistens og reducerede menneskelige fejl. Ikke kun det, men efterhånden som ændringer til applikationerne foretages, og den skaleres langs X/Y/Z-akserne, vil ændringerne der kræves til workflows og deployment (f.eks. Docker, Kubernetes, osv.) være betydeligt mindre end hvis pipelines skulle udføres manuelt hver gang. Kompleksiteten af pipelinen i dette projekt er dog relativt lav. Dette skyldes kun at have to meget simple microservices, hvilket lægger meget lidt pres på både skrivning af Dockerfiles og GitHub Actions workflow. I et virkeligheds-scenarie ville der selvfølgelig være flere elementer at overveje, det være sig deployment af specifikke services med Docker Swarm/Kubernetes, eller at have flere miljøer (staging, produktion, osv.). Dette ville øge den indledende kompleksitet og tidsinvestering, selvom det fundamentale princip forbliver: infrastructure-as-code skalerer mere gunstigt end manuelle processer efterhånden som systemkompleksiteten vokser.

Automatiseringsinfrastrukturen der blev skabt, inklusive Dockerfiles, GitHub Actions workflows og deployment-scripts, repræsenterer en engangsomkostning der fortsætter med at give værdi gennem hele applikationens livscyklus.

5 Diskussion

Der er stadig flere punkter der skal diskuteres, og flere begrænsninger sat af dette projekts omfang. Hvor overrasket var jeg over resultaterne? Ville det mindre omfang af applikationen allerede have sat begrænsninger på mine forventninger? Hvilke tilføjelser kunne laves til projektet for yderligere at validere hypotesen, og hvilke tilføjelser kunne have været lavet for endnu mere at automatisere processen? Hvad gør vores udviklings-/cloud-miljø med hensyn til at påvirke de værktøjer vi kunne have brugt?

5.1 Forventning & Virkelighed

Som forventet; jo mindre omfanget af projektet er, desto mindre er de potentielle gevinster fra automatisering. Det er klart at for en applikation af denne størrelse, overhead'et ved at opsætte CI/CD-pipelines ikke er retfærdiggjort af tiden sparet ved at automatisere bygning/deployment alene. Dog er fordelene ved konsistens og fejlreduktion tydelige selv i denne skala. Jeg er ikke særligt overrasket over resultaterne, da de stemmer godt overens med etableret litteratur om CI/CD-fordele[2], [3], og omfanget er lille nok til at alle edge cases kan tælles på én hånd.

5.2 Begrænsninger & Mulige Tilføjelser

Det begrænsede omfang af dette projekt begrænser generaliseringsbarheden af resultaterne. Virkelige produktionssystemer involverer typisk database-migrationer, secret management, multi-miljø deployments (dev/staging/produktion), og rollback-kapaciteter. Hvert af disse elementer

ville øge både manuel og automatiseret deployment-kompleksitet, selvom kløften mellem dem sandsynligvis ville udvides yderligere til automatiseringens fordel.

Applikationens enkelhed udelukkede også brugen af mere sofistikeret tooling der bliver nødvendig ved større skala. Docker Swarm eller Kubernetes orkestrering, kommercielle cloud-platforme (AWS, Azure, GCP), og Infrastructure-as-Code værktøjer som Terraform ville tilføje betydelig indledende kompleksitet men give kapaciteter essentielle for produktions-deployments: auto-skalering, self-healing, og geografisk distribution. Disse værktøjer repræsenterer det næste niveau af CI/CD-modenhedskurven, hvor J-kurvens dyk bliver stejlere, men de eventuelle gevinster bliver mere betydelige.

Derudover målte projektet ikke alle fire kernemålinger fra DORA[3]: mens deployment-tid blev målt, blev mean time to recovery (MTTR), change failure rate, og deployment-frekvens under load ikke sporet. Den observerede 35x forbedring i konsistens antyder at disse kvalitetsmålinger ville vise favorable tendenser, da litteraturen foreslår at automatisering reducerer fejlrater, udvikler-tilfredshed og recovery-tider, men jeg kan ikke pålideligt kvantificere fejl og recovery uden mere omfattende test, og at lave lektier introducerer en vis bias omkring emnet udvikler-tilfredshed.

6 Konklusion

Dette projekt undersøgte afvejningerne mellem indledende kompleksitet og langsigtede fordele ved implementering af CI/CD-praksisser ved hjælp af GitHub Actions og Docker. Ved at konstruere en microservice-applikation og sammenligne manuel deployment med automatiserede tilgange, har jeg demonstreret at selvom CI/CD introducerer indledende kompleksitet, giver det betydelige langsigtede fordele med hensyn til deployment-hastighed, konsistens og fejlreduktion.

De kvantitative resultater er overbevisende: automatiseret deployment reducerede deployment-tid med over 70% i gennemsnit og demonstrerede 35x bedre konsistens (3 sekunders varians mod 105 sekunder). Den indledende setup-investering på 3,5-4 timer når break-even efter cirka 90-100 deployments, hvilket repræsenterer en tilbagebetalingsperiode på 5-6 uger for teams der deployer regelmæssigt.

Selvom det begrænsede omfang begrænser generaliserbarhed, stemmer resultaterne overens med etableret CI/CD-litteratur og validerer det fundamentale princip: automatisering transformerer inkonsistente manuelle processer til pålidelig, reproducerbar infrastruktur. Hypotesen er bekræftet; de langsigtede fordele ved CI/CD opvejer de indledende omkostninger, især når konsistens og fejlreduktion værdsættes sideløbende med rå tidsbesparelser.

For moderne microservice-udvikling er CI/CD ikke blot en bekvemmelighed, men et nødvendigt fundament for bæredygtig, skalerbar softwareleverance.

Referencer

[1] M. Fowler. "Continuous Integration." Accessed: 2024-12-17, hentet 17. dec. 2024. webadr.: https://martinfowler.com/articles/continuousIntegration.html

[2] Atlassian. "Continuous integration vs. delivery vs. deployment." Accessed: 2024-12-17, hentet 17. dec. 2024. webadr.: 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, tekn. rapp., 2023, Accessed: 2024-12-17. webadr.: https://dora.dev/research/2023/dora-report/

[4] Docker Inc. "Docker Documentation: Use containers to Build, Share and Run your applications," hentet 17. dec. 2024. webadr.: https://docs.docker.com/

[5] GitHub. "GitHub Actions Documentation," hentet 17. dec. 2024. webadr.: https://docs.github.com/en/actions

A Resultater af Lokal Deployment-test

Dette bilag indeholder de detaljerede resultater fra de komparative deployment-tests udført for at evaluere manuelle versus automatiserede deployment-tilgange.

A.1 Testkørsel 3 - Omfattende Analyse

Følgende resultater repræsenterer den mest komplette testkørsel, inklusive detaljerede tidsmålinger, fejltælling og ROI-analyse.

```
========================================
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 Sammenfatning af Alle Testkørsler

Alle tre testkørsler demonstrerede konsekvent at automatiseret deployment via Docker og scripts markant reducerede deployment-tid og forbedrede konsistens:

- **Test 1:** Manuel 262s mod Automatiseret 56s (78,7% forbedring)
- **Test 2:** Manuel 157s mod Automatiseret 53s (66,4% forbedring)
- **Test 3:** Manuel 170s mod Automatiseret 53s (68,9% forbedring)

Nøgleresultat: Automatiseret deployment-varians var 3 sekunder sammenlignet med manuel deployment-varians på 105 sekunder, hvilket demonstrerer 35x forbedring i konsistens.

B GitHub Actions Deployment Testresultater

Dette bilag indeholder dokumentationen hentet fra en af performance-målings workflow-kørslerne i 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
```

Selvom workflow-actions kan inspiceres direkte i repositoriet, giver denne log et øjebliksbillede af performance-målingerne opnået under CI/CD deployment-processen. Tiden registreret for denne kørsel fra den 15. december kl. 12:44 GMT+1, var cirka 1 minut og 37 sekunder, eller 97 sekunder.