Univerzitet Donja Gorica

Fakultet za informacione sisteme i tehnologije

Podgorica

Automatizacija isporuke softvera primjenom CI/CD pipeline-a i kontejnerizacije

Diplomski rad

Student: Armin Ramusović

Broj dosijea: 22/125

Podgorica, februar 2026. godine

Fakultet za informacione sisteme i tehnologije

Podgorica

Automatizacija isporuke softvera primjenom CI/CD pipeline-a i kontejnerizacije

Diplomski rad

Mentor: Prof. dr Tomo Popović

Komentor: dr Stevan Čakić

Student: Armin Ramusović

Broj dosijea: 22/125

Podgorica, februar 2026. godine

Apstrakt

Ovaj rad istražuje uticaj automatizacije isporuke softvera primjenom CI/CD *pipeline*-a i kontejnerizacije na efikasnost i pouzdanost procesa implementacije. Istraživanje je sprovedeno kroz dva kontrolisana eksperimenta nad višeservisnom aplikacijom razvijenom u Java Spring Boot okruženju i kontejnerizovanom pomoću Docker tehnologije. U prvom eksperimentu poređen je ručni i automatizovani CI/CD proces isporuke, pri čemu je automatizacija smanjila prosječno vrijeme isporuke sa 185 na 65,8 sekundi (smanjenje od 64,4%). Drugi eksperiment poredio je Recreate i Blue/Green strategije implementacije, pokazujući da Blue/Green pristup smanjuje period nedostupnosti sistema za 44% i omogućava 75% brži povratak na prethodnu verziju. Rezultati potvrđuju da integracija CI/CD *pipeline*-a i kontejnerizacije značajno unapređuje brzinu, pouzdanost i konzistentnost isporuke softvera.

Ključne riječi: CI/CD, kontejnerizacija, Docker, automatizacija, strategije implementacije, Blue/Green, DevOps, GitHub Actions, kontinuirana integracija, kontinuirana isporuka

Abstract

This thesis investigates the impact of software delivery automation using CI/CD pipelines and containerization on the efficiency and reliability of the deployment process. The research was conducted through two controlled experiments on a multi-service application developed in the Java Spring Boot framework and containerized using Docker technology. The first experiment compared manual and automated CI/CD delivery processes, where automation reduced the average delivery time from 185 to 65.8 seconds (a 64.4% reduction). The second experiment compared Recreate and Blue/Green deployment strategies, demonstrating that the Blue/Green approach reduces downtime by 44% and enables 75% faster rollback. The results confirm that the integration of CI/CD pipelines and containerization significantly improves the speed, reliability, and consistency of software delivery.

Keywords: CI/CD, containerization, Docker, automation, deployment strategies, Blue/Green, DevOps, GitHub Actions, continuous integration, continuous delivery

Sadržaj

Apstrakt 3
Abstract 3
1. Uvod 6
  1.1. Predmet i cilj rada 6
  1.2. Hipoteza rada 7
  1.3. Opis eksperimentalnog istraživanja 7
2. Teorijski okvir 8
  2.1. CI/CD koncept 8
  2.2. Kontejnerizacija sa Docker-om 9
  2.3. Automatizovano testiranje 9
  2.4. Strategije implementacije 10
  2.5. Metrike kvaliteta isporuke softvera 11
3. Metodologija istraživanja 13
  3.1. Opis eksperimentalnog okruženja 13
  3.2. Alati i tehnologije 14
  3.3. Dizajn eksperimenata 14
    3.3.1. Ručni vs CI/CD proces isporuke 14
    3.3.2. Recreate vs Blue/Green 15
  3.4. Definisanje metrika i način mjerenja 16
4. Implementacija CI/CD pipeline-a i kontejnerizacije 18
  4.1. Arhitektura sistema i testna aplikacija 18
    4.1.1. Struktura api-service komponente 19
    4.1.2. Testovi i kvalitet koda 20
  4.2. Kontejnerizacija aplikacije 20
    4.2.1. Traefik konfiguracija 21
    4.2.2. Docker Compose za Blue/Green scenario 21
  4.3. Konfiguracija CI/CD pipeline-a 22
  4.4. Organizacija repozitorijuma i workflow fajlova 24
  4.5. Implementacija eksperimenata 25
5. Eksperimentalni rezultati i analiza 29
  5.1. Ručni vs CI/CD proces isporuke 29
  5.2. Recreate vs Blue/Green 31
  5.3. Uporedna analiza i diskusija 35
6. Zaključak 38
Literatura 40

LISTA SLIKA I TABELA

Tabela 1. Dizajn eksperimenata 16

Slika 1. Arhitektura sistema 19

Slika 2. Logički prikaz kontejnerske organizacije 22

Slika 3. Tok CI/CD procesa 24

Slika 4. Recreate implementacija 26

Slika 5. Blue/Green implementacija sa prebacivanjem saobraćaja 27

Slika 6. Povratak na prethodnu verziju u Blue/Green modelu 28

Tabela 2. Pojedinačna mjerenja vremena isporuke (u sekundama) 29

Tabela 3. Sumirani rezultati Eksperimenta 1 29

Slika 7. Uspješno izvršen CI/CD pipeline u GitHub Actions interfejsu 31

Tabela 4. Mjerenja za Recreate strategiju (u sekundama) 32

Slika 8. Log recreate eksperimenta u Github Actions 32

Tabela 5. Mjerenja za Blue/Green strategiju (u sekundama) 33

Slika 9. Eksterno mjerenje Blue/Green implementacije putem dva paralelna terminala 34

Tabela 6. Sumirani rezultati Eksperimenta 2 34

1. Uvod

Savremeni razvoj softvera karakterišu brze promjene zahtjeva, česte nadogradnje sistema i potreba za pouzdanom isporukom aplikacija u kratkim vremenskim intervalima. U takvom okruženju, tradicionalni ručni procesi izgradnje i instalacije softvera često dovode do povećanog broja grešaka, neusklađenosti okruženja i produženog vremena isporuke.

Kao odgovor na ove izazove razvijene su DevOps prakse, CI/CD *pipeline*-i i kontejnerizacija. Automatizacijom procesa izgradnje, testiranja i implementacije povećava se dosljednost i smanjuje mogućnost ljudske greške, dok kontejnerizacija omogućava pokretanje aplikacija u standardizovanom i ponovljivom okruženju. Ove tehnologije danas su široko zastupljene u modernom razvoju softvera.[1]

Značaj automatizacije potvrđuju i višegodišnja istraživanja u oblasti DevOps praksi. Inicijativa DevOps Research and Assessment (DORA) kroz kontinuirane analize performansi tehnoloških organizacija ukazuje na povezanost između primjene DevOps praksi i unapređenja brzine i stabilnosti isporuke softvera.[2] Ovi nalazi dodatno potvrđuju aktuelnost i relevantnost teme kojom se rad bavi.

Iako se prednosti automatizacije često ističu u literaturi i praksi, konkretni efekti na vrijeme isporuke, period nedostupnosti sistema (*downtime*) i brzinu vraćanja na prethodnu verziju (*rollback*) zahtijevaju jasno i mjerljivo ispitivanje. Upravo potreba za kvantitativnom analizom u kontrolisanim uslovima predstavlja osnovnu motivaciju ovog rada.

1.1. Predmet i cilj rada

Predmet ovog diplomskog rada jeste analiza i eksperimentalno ispitivanje uticaja primjene CI/CD *pipeline*-a i kontejnerizacije na proces isporuke softvera. Cilj rada je da se kroz praktično istraživanje i uporedne eksperimente utvrdi u kojoj mjeri ove tehnologije doprinose smanjenju vremena isporuke softvera, povećanju pouzdanosti sistema, smanjenju broja grešaka tokom procesa implementacije, te poboljšanju efikasnosti mehanizama za oporavak i smanjenju perioda nedostupnosti sistema.

[1] Kim, G., Humble, J., Debois, P. & Willis, J. (2016). *The DevOps Handbook*. IT Revolution Press.

[2] DevOps Research and Assessment (DORA). (2024). *Accelerate State of DevOps Report*. Google Cloud.

1.2. Hipoteza rada

Glavna hipoteza rada glasi:

Primjena CI/CD pipeline-a u kombinaciji sa kontejnerizacijom značajno unapređuje brzinu i pouzdanost isporuke softvera u poređenju sa tradicionalnim ručnim pristupom.

Pomoćne hipoteze:

- Automatizacija procesa implementacije smanjuje broj grešaka izazvanih razlikama između okruženja i ljudskim faktorom.
- Kontejnerizacija omogućava stabilnije i ponovljivo izvršavanje aplikacija.
- Vrijeme oporavka sistema nakon greške značajno se smanjuje primjenom naprednih strategija implementacije poput Blue/Green pristupa.

1.3. Opis eksperimentalnog istraživanja

Da bi se hipoteze provjerile, rad se zasniva na dva eksperimenta:

Eksperiment 1: Ručni vs CI/CD proces isporuke

Poredi se tradicionalni ručni proces isporuke sa potpuno automatizovanim CI/CD pristupom. Mjeri se ukupno vrijeme isporuke nove verzije aplikacije, od početka procesa do potvrde dostupnosti servisa.

Eksperiment 2: Strategije implementacije (Recreate vs Blue/Green)

Poredi se recreate strategija (zaustavljanje stare i pokretanje nove verzije) sa Blue/Green strategijom (paralelno pokretanje nove verzije i prebacivanje saobraćaja). Mjere se period nedostupnosti i vrijeme povratka na prethodnu verziju u oba scenarija, uz simulaciju greške u aplikaciji.

Metrike koje se mjere:

- Ukupno vrijeme isporuke nove verzije
- Period nedostupnosti sistema (*downtime*)
- Vrijeme povratka na prethodnu verziju (*rollback*)
- Stabilnost aplikacije nakon isporuke

2. Teorijski okvir

Prije detaljnog opisa eksperimentalne postavke, neophodno je definisati teorijske osnove na kojima se rad zasniva. Pojmovi poput kontinuirane integracije, kontejnerizacije i strategija implementacije imaju precizna značenja u kontekstu softverskog inženjerstva, a njihovo razumijevanje je preduslov za interpretaciju rezultata eksperimenata. U nastavku su objašnjeni ključni koncepti i principi koji čine temelj automatizovanog procesa isporuke softvera. Ovo poglavlje daje teorijsku osnovu za razumijevanje CI/CD pristupa, kontejnerizacije i strategija implementacije koje su korišćene u eksperimentalnom dijelu rada.

2.1. CI/CD koncept

CI/CD predstavlja skup praksi i tehnika koje imaju za cilj automatizaciju procesa izgradnje, testiranja i isporuke softvera. Termin se sastoji iz dva dijela: kontinuirane integracije (Continuous Integration – CI) i kontinuirane isporuke ili implementacije (Continuous Delivery/Deployment – CD).[3]

Kontinuirana integracija podrazumijeva često spajanje promjena u zajedničku granu repozitorijuma uz automatsko pokretanje procesa izgradnje i testiranja.[4] Cilj je da se greške otkriju u ranoj fazi razvoja, čime se smanjuje rizik od složenih problema u kasnijim fazama projekta. Svaka promjena u kodu pokreće automatizovani proces provjere ispravnosti, što povećava stabilnost glavne grane projekta.

Kontinuirana isporuka nadovezuje se na CI i omogućava da svaka uspješno testirana verzija softvera bude spremna za implementaciju u produkcijsko okruženje. U naprednijem obliku, kontinuirana implementacija omogućava automatsko postavljanje koda bez ručne intervencije, čime se skraćuje vrijeme od izmjene koda do dostupnosti nove verzije korisnicima.

CI/CD *pipeline* predstavlja tehničku realizaciju ovih principa. Tipičan *pipeline* sastoji se od nekoliko sekvencijalnih faza: pokretanje procesa nakon izmjene koda (*trigger*), izgradnja aplikacije (build), automatsko testiranje, pakovanje artefakta ili kontejnerske slike, i na kraju implementacija u ciljano okruženje.[5]

[3] Humble, J. & Farley, D. (2010). *Continuous Delivery*. Addison-Wesley Professional.

[4] Duvall, P.M., Matyas, S. & Glover, A. (2007). *Continuous Integration*. Addison-Wesley Professional.

[5] Shahin, M., Babar, M.A. & Zhu, L. (2017). *Continuous Integration, Delivery and Deployment*. IEEE Access.

Važan element automatizovanog *pipeline*-a je i *health check*, mehanizam kojim sistem automatski provjerava da li je servis funkcionalan i spreman za primanje zahtjeva nakon implementacije. U kontekstu CI/CD procesa, *health check* predstavlja posljednji korak koji potvrđuje uspješnost isporuke prije nego što se *pipeline* označi kao završen.

2.2. Kontejnerizacija sa Docker-om

Kontejnerizacija predstavlja pristup pakovanja aplikacije zajedno sa svim njenim zavisnostima u izolovanu izvršnu jedinicu – kontejner.[6] Za razliku od tradicionalne instalacije softvera, gdje aplikacija zavisi od konfiguracije operativnog sistema i instaliranih biblioteka, kontejner omogućava da se aplikacija izvršava u kontrolisanom i ponovljivom okruženju.

Docker je najčešće korišćen alat za kontejnerizaciju.[7] Osnovni elementi Docker ekosistema su: Dockerfile (fajl koji definiše način izgradnje kontejnerske slike), Docker image (rezultat procesa izgradnje koji sadrži aplikaciju i sve potrebne komponente), Docker container (pokrenuta instanca slike) i Registry (repozitorijum za čuvanje i verzionisanje slika).

Jedna od najvažnijih prednosti kontejnerizacije jeste eliminisanje problema neusklađenosti okruženja, poznatog kao situacija u kojoj aplikacija funkcioniše na razvojnoj mašini, ali ne i u produkciji. Kontejner omogućava da ista verzija aplikacije bude izvršena u identičnim uslovima u svim fazama – od razvoja do produkcije.

Za sisteme koji se sastoje od više servisa koristi se Docker Compose. Ovaj alat omogućava definisanje kompletnog sistema (aplikacija, baza podataka, *proxy* servisi i slično) u jednom konfiguracionom fajlu. Time se omogućava jednostavno podizanje i gašenje kompletnog sistema jednom komandom, što je posebno važno u kontekstu automatizovanog procesa implementacije.

U okviru ovog rada, kontejnerizacija predstavlja ključni mehanizam standardizacije okruženja i osnove za implementaciju automatizovanog CI/CD procesa.

2.3. Automatizovano testiranje

Automatizovano testiranje predstavlja ključnu komponentu CI/CD procesa. U trenutku kada se promjena koda integriše u zajednički repozitorijum, *pipeline* automatski pokreće

[6] Mouat, A. (2016). *Using Docker*. O'Reilly Media.

[7] Merkel, D. (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. Linux Journal.

unaprijed definisane testove kako bi se provjerilo da nova verzija ne narušava postojeću funkcionalnost. Na ovaj način greške se otkrivaju rano, prije nego što kod dospije u produkcijsko okruženje.

Unit testovi su osnovna vrsta automatizovanih testova — svaki test provjerava ispravnost jedne izolovane komponente ili metode, nezavisno od ostatka sistema. Za pisanje unit testova u Java okruženju standardno se koristi JUnit 5, koji pruža anotacije i mehanizme za definisanje i pokretanje testova. Kada testirana komponenta zavisi od vanjskih servisa ili baza podataka, koristi se Mockito, biblioteka koja omogućava kreiranje lažnih objekata (mock-ova) koji simuliraju ponašanje tih zavisnosti. Na ovaj način test ostaje izolovan i ponovljiv bez potrebe za stvarnom infrastrukturom.

U okviru ovog rada, JUnit 5 i Mockito korišćeni su za testiranje ključnih komponenti oba servisa. Testovi se automatski pokreću unutar CI/CD *pipeline*-a kao dio *maven verify* komande, čime se osigurava da nijedna verzija softvera koja ne prolazi testove ne može biti isporučena u produkcijsko okruženje.[8]

2.4. Strategije implementacije

Strategija implementacije definiše način na koji se nova verzija aplikacije uvodi u produkcijsko okruženje.[9] Izbor strategije utiče na vrijeme nedostupnosti sistema, mogućnost brzog povratka na prethodnu verziju i ukupnu pouzdanost isporuke.

U ovom radu razmatraju se dvije strategije:

Recreate pristup podrazumijeva gašenje postojeće verzije aplikacije i pokretanje nove verzije. Ova strategija je jednostavna za implementaciju, ali podrazumijeva kratkotrajni prekid dostupnosti sistema. Prednost je minimalna kompleksnost, dok je glavni nedostatak potencijalni period nedostupnosti tokom tranzicije.

Blue/Green implementacija podrazumijeva istovremeno postojanje dvije verzije aplikacije: aktivne (npr. Blue) i nove verzije (Green). Nova verzija se pokreće paralelno sa postojećom, testira se u izolovanom režimu, a zatim se saobraćaj preusmjerava na novu verziju putem *reverse proxy* mehanizma.

[8] Meszaros, G. (2007). *xUnit Test Patterns*. Addison-Wesley Professional.

[9] Humble, J. & Farley, D. (2010). *Continuous Delivery*. Addison-Wesley Professional.

Ključne prednosti ove strategije su minimalan ili nikakav period nedostupnosti, kao i jednostavan povratak na prethodnu verziju koji se svodi na preusmjeravanje saobraćaja nazad na prethodnu verziju, te manji rizik pri implementaciji nove verzije.[10]

U kontekstu ovog rada, Blue/Green strategija se implementira pomoću Docker Compose konfiguracija i Traefik *reverse proxy*-a.

2.5. Metrike kvaliteta isporuke softvera

U cilju objektivne procjene uticaja CI/CD pristupa i kontejnerizacije na proces isporuke softvera, neophodno je definisati mjerljive pokazatelje performansi sistema.[11] Bez jasno definisanih metrika nije moguće donijeti pouzdane zaključke o efikasnosti određenog pristupa, niti izvršiti uporednu analizu različitih strategija implementacije.

U savremenoj DevOps literaturi poseban značaj imaju metrike koje mjere brzinu i pouzdanost isporuke. Istraživanja inicijative DevOps Research and Assessment (DORA) identifikuju ključne pokazatelje performansi organizacija, među kojima se izdvajaju vrijeme implementacije promjene (Lead Time for Changes), učestalost implementacije, stopa neuspjelih promjena i vrijeme potrebno za oporavak sistema nakon incidenta (Time to Restore Service).[12] Ove metrike omogućavaju kvantitativnu procjenu sposobnosti sistema da brzo isporuči novu funkcionalnost uz očuvanje stabilnosti.

U kontekstu ovog rada fokus je stavljen na metrike koje su direktno povezane sa procesom implementacije i oporavkom sistema, a koje se mogu precizno izmjeriti u kontrolisanom eksperimentalnom okruženju.

Prva posmatrana metrika jeste ukupno vrijeme isporuke nove verzije aplikacije. Ova metrika obuhvata period od pokretanja procesa implementacije do trenutka kada je nova verzija sistema dostupna korisnicima. Vrijeme isporuke predstavlja pokazatelj agilnosti sistema i efikasnosti automatizacije, jer kraće vrijeme implementacije omogućava brže reagovanje na promjene zahtjeva i ispravke grešaka.

Druga metrika je period nedostupnosti sistema, koji predstavlja vremenski interval tokom kojeg servis nije dostupan korisnicima. Period nedostupnosti je ključni pokazatelj

[10] Fowler, M. (2010). *Blue-Green Deployment*. martinfowler.com.

[11] Forsgren, N., Humble, J. & Kim, G. (2018). *Accelerate*. IT Revolution Press.

[12] DevOps Research and Assessment (DORA). (2024). *Accelerate State of DevOps Report*. Google Cloud.

pouzdanosti sistema, jer direktno utiče na korisničko iskustvo i kontinuitet poslovanja. Smanjenje perioda nedostupnosti jedan je od glavnih ciljeva naprednih strategija implementacije, posebno u distribuiranim sistemima.

Treća metrika odnosi se na vrijeme vraćanja sistema na prethodnu stabilnu verziju. Ova metrika je analogna DORA pokazatelju „Time to Restore Service“, jer mjeri sposobnost sistema da se brzo oporavi nakon detekcije greške. U praksi, brz i pouzdan mehanizam za povratak na prethodnu verziju predstavlja važan element otpornosti sistema, naročito u scenarijima gdje nova verzija sadrži aplikativne nedostatke koji nisu detektovani osnovnim provjerama dostupnosti.

3. Metodologija istraživanja

U cilju provjere postavljenih hipoteza, istraživanje u ovom radu zasnovano je na eksperimentalnoj uporednoj analizi tradicionalnog i automatizovanog procesa isporuke softvera. Metodologija je koncipirana tako da omogući objektivno mjerenje vremena isporuke, pouzdanosti procesa implementacije i efikasnosti mehanizama za povratak na prethodnu verziju.

Eksperimenti su realizovani u kontrolisanom okruženju na virtualnom privatnom serveru, uz ponovljiva mjerenja i jasno definisane metrike. Posebna pažnja posvećena je eliminisanju spoljašnjih varijabli kako bi rezultati bili dosljedni i uporedivi.

3.1. Opis eksperimentalnog okruženja

Eksperimenti su realizovani na virtualnom privatnom serveru (VPS) pružaoca Hetzner, sa operativnim sistemom Ubuntu 24.04 LTS. Server je konfigurisan sa 4 vCPU jezgra i 8 GB RAM memorije, što predstavlja realistično produkcijsko okruženje za aplikacije srednje veličine.

Softversko okruženje obuhvata Java Development Kit (JDK 21), Spring Boot 3.x uz Maven kao alat za izgradnju projekta, Docker Engine i Docker Compose za kontejnerizaciju i orkestraciju servisa, Traefik v2.11 kao *reverse proxy*, PostgreSQL 16 kao sistem za upravljanje bazom podataka, GitHub Actions kao CI/CD sistem, te GitHub Container Registry (GHCR) za skladištenje Docker slika.

Eksperimentalna aplikacija sastoji se od dva Spring Boot servisa i jedne baze podataka:

- **api-service** (port 8080) – glavni REST servis koji upravlja poslovnom logikom, pristupa PostgreSQL bazi podataka putem JPA/Hibernate, i izlaže *endpoint*-e uključujući /actuator/health za provjeru zdravlja servisa i /notifycheck za provjeru dostupnosti notify servisa.
- **notify-service** (port 8081) – pomoćni servis koji služi za testiranje stabilnosti i scenarija povratka na prethodnu verziju. Sadrži dva *endpointa*: /health koji uvijek vraća HTTP 200, i /health-buggy koji uvijek vraća HTTP 500, čime se simulira greška u servisu.
- **PostgreSQL 16** – relaciona baza podataka za trajno skladištenje podataka.

Servisi su povezani putem interne Docker mreže, dok Traefik obavlja funkciju *reverse proxy*-a i upravlja usmjeravanjem saobraćaja ka aktivnoj verziji aplikacije.

3.2. Alati i tehnologije

Za implementaciju sistema korišćeni su sljedeći alati:

Spring Boot korišćen je za razvoj oba servisa, dok je Maven korišćen za upravljanje zavisnostima i procesom izgradnje. **Maven** omogućava standardizovan proces izgradnje aplikacije, uključujući kompilaciju, pokretanje testova (JUnit 5 sa Mockito) i generisanje izvršnog JAR artefakta.

Docker je korišćen za kontejnerizaciju aplikacija, dok je **Docker Compose** korišćen za definisanje i pokretanje kompletnog sistema jednom komandom.[13] Korišćena su dva tipa Dockerfile-a: *multi-stage* Dockerfile za ručnu implementaciju (Maven build + runtime u jednoj slici) i pojednostavljeni Dockerfile za CI/CD (samo kopiranje gotovog JAR artefakta u runtime sliku).[14]

Traefik v2.11 korišćen je kao *reverse proxy* i *load balancer*.[15] Njegova uloga je usmjeravanje HTTP saobraćaja ka aktivnoj verziji aplikacije. U kontekstu Blue/Green strategije, Traefik omogućava prebacivanje saobraćaja između verzija izmjenom dinamičke konfiguracije. Konfiguracija se definiše u YAML fajlu sa opcijom za automatsko učitavanje promjena.

GitHub Actions korišćen je za implementaciju CI/CD *pipeline*-a.[16] Automatizovani proces uključuje izgradnju, testiranje, izradu Docker slika, objavljivanje na GHCR sa Git *commit* SHA tagom, te automatsku implementaciju na VPS putem SSH konekcije. Za svaki eksperiment kreiran je poseban *workflow* fajl sa odgovarajućom logikom.

3.3. Dizajn eksperimenata

Eksperimentalni dio rada sastoji se od dva eksperimenta koji su direktno povezani sa automatizacijom isporuke softvera.

3.3.1. Ručni vs CI/CD proces isporuke

U ovom eksperimentu porede se dva pristupa isporuci softvera:

---

[13] Docker Documentation. https://docs.docker.com

[14] Turnbull, J. (2014). *The Docker Book*. James Turnbull.

[15] Traefik Documentation. https://doc.traefik.io/traefik

[16] GitHub Actions Documentation. https://docs.github.com/en/actions

Ručni proces uključuje sljedeće korake: SSH konekciju na VPS server, navigaciju do *root* direktorijuma servisa, pokretanje testova komandom *./mvnw test*, navigaciju do direktorijuma sa Docker Compose fajlom, te pokretanje *docker compose up* komande koja lokalno vrši izgradnju slike korišćenjem *multi-stage* Dockerfile-a (Maven *build* + *runtime*). Nakon pokretanja, ručno se provjeravaju /actuator/health i /notifycheck *endpoint*-i.

CI/CD proces se aktivira ručno putem GitHub Actions workflow_dispatch *trigger*-a. *Pipeline* se sastoji od tri sekvencijalna zadatka: build-and-test (*checkout* koda, *setup* JDK 21 sa Maven *cache*-om, *mvn verify* za oba servisa), docker-push (preuzimanje JAR artefakata, prijava na GHCR, izgradnja Docker slika sa pojednostavljenim Dockerfile.ci, objava sa SHA tagom), i implementacija (SSH na VPS, preuzimanje novih slika, *docker compose up*, automatski *health check* sa *retry loop*-om).

Mjeri se ukupno vrijeme isporuke – od početka procesa do potvrde dostupnosti servisa. Svaki scenario ponovljen je pet puta.

3.3.2. Recreate vs Blue/Green

U ovom eksperimentu porede se dvije strategije implementacije nove verzije: recreate i Blue/Green pristup.

U recreate scenariju *workflow* se aktivira ručno na GitHub-u. Nakon izgradnje i objave slika, na VPS-u se čita prethodni uspješan tag slike iz fajla za oporavak, zaustavlja se postojeći kontejner *(docker compose stop api-blue*), pokreće se nova verzija i mjeri se period nedostupnosti. Provjerava se /notifycheck koji vraća grešku (jer nova verzija poziva /health-buggy na notify servisu), nakon čega se pokreće povratak na prethodnu verziju i mjeri se vrijeme oporavka.

Kod Blue/Green pristupa nova verzija (api-green) se pokreće paralelno sa postojećom (api-blue) korišćenjem Docker Compose profila. Interno se provjerava zdravlje green instance. Zatim se Traefik dinamička konfiguracija mijenja (sed komandom) sa api-blue na api-green i restartuje se Traefik. Provjerava se /notifycheck, i ako vrati grešku, vrši se oporavak vraćanjem konfiguracije na api-blue.

Posebno je važno napomenuti da api-service sadrži *custom* HTTP *header* X-Service-Version sa vrijednošću blue ili green, koji omogućava identifikaciju aktivne instance tokom mjerenja.

Za potrebe testiranja mehanizama za povratak na prethodnu verziju, u kodu api-servisa NotifyCheckController je konfigurisan da poziva /health-buggy *endpoint* na notify servisu umjesto /health. *Endpoint* /health-buggy je implementiran tako da uvijek vraća HTTP 500 status. Ovo simulira scenario u kojem implementacija prođe osnovni *health check* (Spring Boot actuator), ali ima aplikativni bug koji se manifestuje tek pri komunikaciji sa drugim servisom. Mjere se period nedostupnosti i vrijeme oporavka. Svaki scenario ponovljen je pet puta.

Pregled dizajna eksperimenata prikazan je u Tabeli 1.

Tabela 1 - Dizajn eksperimenata

Element	Eksperiment 1	Eksperiment 2
Cilj	Poređenje ručne i automatizovane implementacije	Poređenje strategija implementacije
Poređeni pristupi	Ručna vs CI/CD implementacija	Recreate vs Blue/Green
Metrike	Vrijeme isporuke	Period nedostupnosti, vrijeme oporavka
Način mjerenja	Štoperica / logovi GitHub Actions	Unix timestamp / eksterno HTTP mjerenje
Broj ponavljanja	5	5
Simulacija greške	Ne	Da (/health-buggy)

3.4. Definisanje metrika i način mjerenja

Radi obezbjeđivanja pouzdanosti i ponovljivosti rezultata, svi eksperimenti su sprovedeni pod kontrolisanim uslovima, uz jasno definisan način mjerenja i prikupljanja podataka. Svaki eksperimentalni scenario ponovljen je pet puta, a za konačnu analizu korišćene su prosječne vrijednosti dobijenih mjerenja. Na ovaj način umanjen je uticaj slučajnih varijacija uzrokovanih opterećenjem sistema, diskovnim operacijama ili mrežnim kašnjenjima.

Vrijeme isporuke u ručnom scenariju mjereno je korišćenjem štoperice, pri čemu je mjerenje započinjalo u trenutku kada je developer započeo proces povezivanja na server, a završavalo kada je *health endpoint* potvrdio dostupnost nove verzije. U automatizovanom

scenariju, vrijeme isporuke mjereno je iz logova GitHub Actions *workflow*-a, od početka do završetka svih zadataka.

Za recreate strategiju, period nedostupnosti sistema i povratak na prethodnu stabilnu verziju su mjereni automatski unutar GitHub Actions *workflow* skripte korišćenjem Unix *timestamp*-a u milisekundama (date +%s%3N). Period nedostupnosti se mjeri od zaustavljanja starog kontejnera do uspješnog *health check*-a novog kontejnera. Povratak na prethodnu verziju se mjeri od početka procedure oporavka do uspješnog *health check*-a sa prethodnom verzijom.

Za Blue/Green strategiju, mjerenje je rađeno eksterno pomoću dva terminala na lokalnom računaru koji šalju HTTP zahtjeve svakih 0.2 sekundi. Terminal 1 prati /actuator/health, a Terminal 2 prati /notifycheck, oba ispisujući HTTP status i X-Service-Version *header*. Period nedostupnosti se definiše kao period kada nijedan servis ne odgovara, a vrijeme povratka na prethodnu verziju kao period od prvog tajmauta pri prebacivanju do ponovnog stabilnog odgovora sa X-Service-Version: blue.

Svi prikupljeni podaci obrađeni su korišćenjem aritmetičke sredine kao centralne mjere, dok je standardna devijacija korišćena radi procjene stabilnosti i varijacije procesa. Rezultati su prikazani tabelarno radi lakše interpretacije i poređenja eksperimentalnih scenarija.

4. Implementacija CI/CD pipeline-a i kontejnerizacije

U ovom poglavlju prikazana je praktična realizacija automatizovanog sistema isporuke softvera zasnovanog na CI/CD *pipeline*-u i kontejnerizaciji. Implementacija obuhvata razvoj testne aplikacije, njenu kontejnerizaciju, konfiguraciju Docker Compose okruženja, kao i postavljanje automatizovanog procesa izgradnje i implementacije putem GitHub Actions sistema.

4.1. Arhitektura sistema i testna aplikacija

U cilju sprovođenja eksperimentalnog istraživanja razvijen je kontrolisani višeservisni sistem koji simulira realno produkcijsko okruženje, ali bez nepotrebne infrastrukturne kompleksnosti. Osnovna ideja bila je konstruisati arhitekturu dovoljno složenu da omogući testiranje različitih strategija isporuke i mehanizama oporavka, a istovremeno dovoljno jednostavnu da fokus ostane na procesu automatizacije, a ne na poslovnoj logici aplikacije.

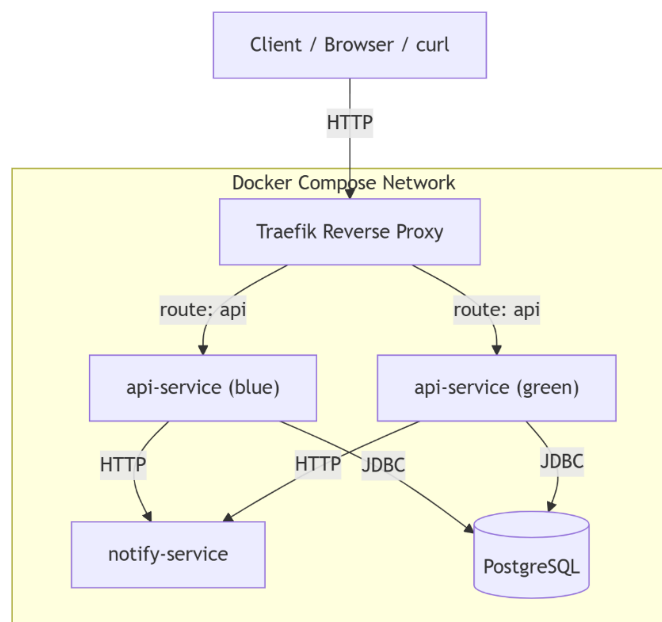
Sistem je organizovan kao skup kontejnerskih servisa koji međusobno komuniciraju putem interne mreže definisane u Docker Compose konfiguraciji. Ulaznu tačku sistema predstavlja Traefik *reverse proxy* komponenta, koja prihvata spoljašnje HTTP zahtjeve na portu 80 i prosljeđuje ih odgovarajućoj verziji aplikativnog servisa.

Centralnu komponentu sistema čini api-service implementiran korišćenjem Spring Boot 3.x okvira sa JDK 21. Ovaj servis izlaže REST interfejs za CRUD operacije nad Item entitetom, pristupa PostgreSQL bazi putem Spring Data JPA, i sadrži dva ključna *endpointa* za nadzor: /actuator/health (standardni Spring Boot *health check*) i /notifycheck (*custom endpoint* koji provjerava dostupnost notify servisa).

Drugi servis (notify-service) uveden je sa ciljem simulacije spoljne zavisnosti i kontrolisanog generisanja grešaka. Servis ima dva *endpointa*: /health koji vraća HTTP 200 i potvrdu da je servis operativan, i /health-buggy koji uvijek vraća HTTP 500. Api-service poziva jedan od ovih *endpointa* kroz NotifyCheckController, čime se omogućava simulacija scenarija neispravne verzije.

Arhitektura je od početka projektovana tako da podržava paralelno postojanje dvije verzije glavnog servisa (api-blue i api-green). Ova odluka je donesena kako bi se omogućila implementacija Blue/Green strategije implementacije. Api-service sadrži ServiceVersionFilter koji svaki HTTP odgovor obogaćuje X-Service-Version *headerom* (blue ili green), čime se

omogućava identifikacija aktivne instance tokom eksperimenata. Kompletna arhitektura sistema sa svim komponentama i tokovima komunikacije prikazana je na Slici 1.



Slika 1. Arhitektura sistema

4.1.1. Struktura api-service komponente

Api-service je implementiran kao standardna Spring Boot aplikacija sa višeslojnom arhitekturom. Glavni kontroler (ItemController) izlaže dva REST *endpointa*: GET /items za preuzimanje svih *item*-a iz baze i POST /items za kreiranje novog *item*-a. Entitet Item sadrži identifikator, naziv i *timestamp* kreiranja, a pristup bazi realizovan je putem Spring Data JPA repozitorijuma (ItemRepo).

Posebno značajna je implementacija NotifyCheckController-a, koji služi kao mehanizam za provjeru povezanosti između servisa. *Endpoint* /notifycheck šalje HTTP zahtjev ka notify servisu i na osnovu odgovora vraća status dostupnosti. U slučaju greške ili nedostupnosti, *endpoint* vraća HTTP 503 status sa opisom greške.

ServiceVersionFilter je komponenta koja dodaje X-Service-Version HTTP *header* u svaki odgovor servisa. Vrijednost *headera* čita se iz *environment* varijable SERVICE_VERSION, čija vrijednost je blue ili green u zavisnosti od aktivne instance. Ovaj *header* korišćen je kao pouzdan mehanizam za identifikaciju aktivne verzije tokom mjerenja u eksperimentima.

Aplikacija koristi standardne Spring Boot *property*-e za konfiguraciju, pri čemu su svi parametri definisani kao *environment* varijable sa podrazumijevanim vrijednostima. Na ovaj način ista Docker slika može se koristiti u različitim okruženjima bez izmjene koda, samo promjenom *environment* varijabli prilikom pokretanja kontejnera.

4.1.2. Testovi i kvalitet koda

Za oba servisa implementirani su JUnit 5 testovi. Api-service sadrži ItemControllerTest koji koristi Mockito za izolaciju kontrolera od repozitorijuma, testira uspješno preuzimanje *item*-a, uspješno kreiranje *item*-a, te odbijanje kreiranja sa praznim ili nedostajućim imenom. Notify-service sadrži HealthControllerTest koji verifikuje da /health vraća HTTP 200, a /health-buggy vraća HTTP 500.

Pokretanje testova u CI/CD *pipeline*-u koristi *mvn verify* komandu koja izvršava kompilaciju, pokretanje testova i verifikaciju. Ukoliko bilo koji test ne prođe, *pipeline* se automatski prekida, čime se sprečava isporuka neispravne verzije koda.

4.2. Kontejnerizacija aplikacije

Kontejnerizacija predstavlja ključni tehnički temelj ovog rada, jer omogućava standardizaciju okruženja i reproducibilnost procesa isporuke. Bez nje, poređenje ručne i automatizovane implementacije ne bi imalo punu težinu, budući da bi razlike u sistemskom okruženju mogle uticati na rezultate mjerenja.

Za kontejnerizaciju korišćena su dva tipa Dockerfile-a, prilagođena različitim scenarijima:

Dockerfile.manuel koji se koristi u ručnom procesu implementacije i sadrži dvije faze. U prvoj fazi (*build stage*), koristi se Maven okruženje za kompilaciju aplikacije i generisanje izvršne JAR datoteke. U drugoj fazi (*runtime stage*), koristi se znatno manja Eclipse Temurin JRE slika koja samo kopira gotov JAR. Ovakva podjela eliminiše *build* alate iz finalne slike.

Dockerfile.ci koji se koristi u CI/CD scenariju je znatno jednostavniji. Sastoji se samo od *runtime* faze koja kopira već kompilirani JAR artefakt u JRE sliku. Ovo je moguće jer GitHub Actions *pipeline* već obavlja kompilaciju i testiranje prije izgradnje Docker slike, pa nema potrebe za fazom izgradnje unutar Dockerfile-a.

Docker Compose konfiguracija predstavlja centralnu tačku upravljanja sistemom. Za svaki eksperimentalni scenario kreiran je poseban compose fajl. Osnovna konfiguracija

definiše Traefik, PostgreSQL, notify-service, i api-blue servis. Za Blue/Green scenario, konfiguracija je proširena api-green servisom koji koristi Docker Compose profile mehanizam (profiles: ["green"]), čime ostaje ugašen dok se eksplicitno ne aktivira.

Posebna pažnja posvećena je načinu verzionisanja slika. U CI/CD scenariju, slike se taguju skraćenim Git *commit* SHA identifikatorom (prvih 7 karaktera), čime se obezbjeđuje precizno praćenje verzija i mogućnost povratka na bilo koju prethodnu verziju.

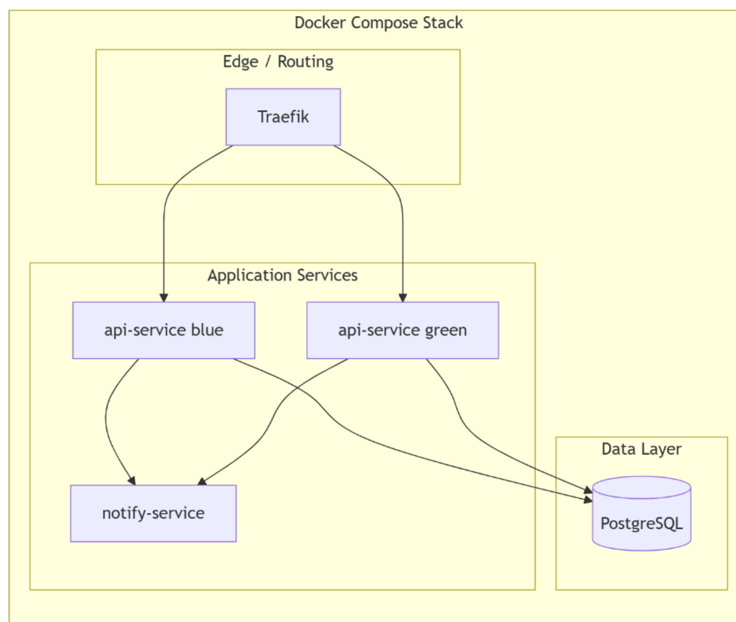
4.2.1. Traefik konfiguracija

Traefik konfiguracija podijeljena je na dvije komponente. Statička konfiguracija definiše ulaznu tačku na portu 80 i referencira dinamički konfiguracionu datoteku sa uključenom *watch* opcijom, koja omogućava automatsko učitavanje promjena bez ponovnog pokretanja Traefika.

Dinamička konfiguracija definiše dva backend servisa: api-blue i api-green, sa njihovim odgovarajućim adresama unutar Docker mreže, te router koji usmjerava sav dolazni saobraćaj ka jednom od njih. Prebacivanje saobraćaja između verzija svodi se na izmjenu jednog polja u ovoj konfiguracionoj datoteci, nakon čega je potrebno restartovati Traefik kako bi promjena bila primijenjena.

4.2.2. Docker Compose za Blue/Green scenario

Docker Compose konfiguracija za Blue/Green scenario definiše obje instance glavnog servisa: api-blue i api-green, sa identičnom strukturom, ali različitim *environment* varijablama. Ključna razlika je SERVICE_VERSION varijabla čija vrijednost odgovara nazivu instance, te profiles polje koje je postavljeno na green za novu verziju. Profil mehanizam osigurava da se api-green instanca ne pokreće automatski sa *docker compose up*, već samo kada se eksplicitno navede odgovarajući profil.



Slika 2. Logički prikaz kontejnerske organizacije

Na Slici 2 prikazan je logički raspored kontejnera unutar Docker Compose *stack*-a, organizovanih u tri sloja: Edge/Routing (Traefik), Application Services (api i notify servis) i Data Layer (PostgreSQL).

4.3. Konfiguracija CI/CD pipeline-a

Automatizacija procesa izgradnje i isporuke realizovana je korišćenjem GitHub Actions sistema. Za svaki eksperiment kreiran je poseban *workflow* fajl sa odgovarajućom logikom implementacije.

Osnovni CI/CD *pipeline workflow* (deploy.yml) se sastoji od tri sekvencijalna zadatka. Prvi zadatak (build-and-test) preuzima izvorni kod, postavlja JDK 21 sa Maven *dependency cache*-om, i pokreće *mvn verify* za oba servisa. Ova faza uključuje kompilaciju i pokretanje JUnit testova sa Mockito *framework*-om. Generisani JAR artefakti se uploaduju za naredni zadatak.

Drugi zadatak (docker-push) preuzima JAR artefakte, prijavljuje se na GitHub Container Registry, vrši izgradnju Docker slike korišćenjem Dockerfile.ci, i objavljuje ih sa dva taga: skraćeni Git *commit* SHA i *latest*. Korišćenje SHA taga umjesto samo *latest*-a omogućava precizno verzionisanje i povratak na bilo koju prethodnu verziju.

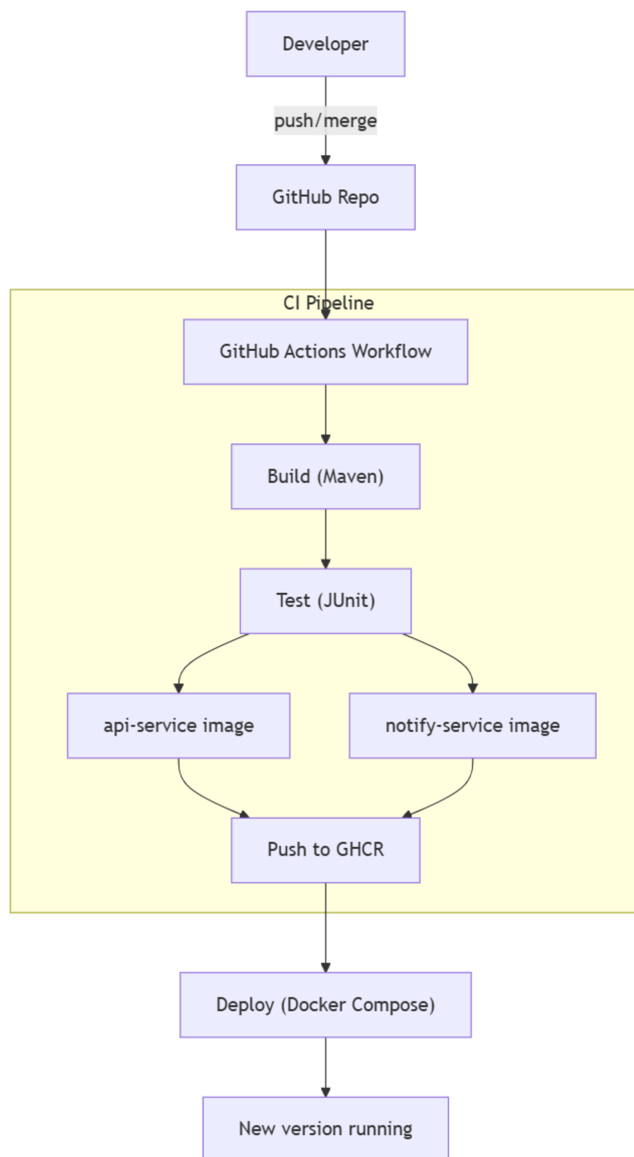
Treći zadatak (deploy) nakon pokretanja novih kontejnera izvršava automatsku provjeru dostupnosti servisa. Skripta u petlji šalje HTTP zahtjeve ka *health endpoint*-u i čeka

potvrdu dostupnosti, pri čemu nakon određenog broja neuspješnih pokušaja prijavljuje grešku i prekida *pipeline*.

Recreate *workflow* proširuje osnovni *pipeline* logikom za precizno mjerenje perioda nedostupnosti i automatski oporavak. Period nedostupnosti se mjeri bilježenjem Unix *timestamp*-a u milisekundama u trenutku zaustavljanja stare i potvrde dostupnosti nove verzije. Ukoliko provjera /notifycheck *endpoint*-a vrati grešku (što je očekivano jer poziva /health-buggy), *workflow* automatski pokreće proceduru povratka na prethodnu verziju i mjeri njeno trajanje na isti način.

Blue/Green *workflow* realizuje proces implementacije kroz četiri faze. U prvoj fazi pokreće se api-green kontejner korišćenjem Docker Compose profila, dok api-blue nastavlja da prima saobraćaj. U drugoj fazi vrši se interna provjera zdravlja nove instance direktno unutar kontejnera. U trećoj fazi mijenja se Traefik konfiguracija i saobraćaj se prebacuje na api-green. U četvrtoj fazi provjerava se /notifycheck *endpoint*, i ukoliko vrati grešku, pokreće se oporavak vraćanjem Traefik konfiguracije na api-blue.

Na Slici 3 prikazan je kompletan tok CI/CD procesa od *push/merge* akcije developera, kroz GitHub Actions *workflow* sa Maven *buildom* i JUnit testovima, do kreiranja Docker slika za oba servisa, objave na GHCR i finalne implementacije putem Docker Compose-a.



Slika 3. Tok CI/CD procesa

4.4. Organizacija repozitorijuma i workflow fajlova

Izvorni kod projekta organizovan je u GitHub repozitorijumu sa sljedećom strukturom:

- **api-service/** – izvorni kod glavnog REST servisa sa Maven konfiguracijom, Dockerfile-ovima i testovima.
- **notify-service/** – izvorni kod pomoćnog servisa za testiranje stabilnosti.
- **infra/** – Docker Compose fajlovi i Traefik konfiguracija.
- **.github/workflows/** – GitHub Actions *workflow* fajlovi za svaki eksperiment.

Za svaki tip implementacije kreiran je poseban *workflow* fajl: deploy.yml (osnovni CI/CD za Eksperiment 1), recreate.yml (recreate strategija za Eksperiment 2), i bluegreen.yml (Blue/Green strategija za Eksperiment 2). Svi *workflow*-i koriste workflow_dispatch *trigger* koji omogućava ručno pokretanje iz GitHub interfejsa, čime se eksperimenti mogu ponavljati bez izmjena i slanja novog koda na repozitorijum.

Infrastrukturni direktorijum sadrži više Docker Compose varijanti: docker-compose.yml (glavni fajl sa Blue/Green podrškom), docker-compose.cicd.yml (za osnovni CI/CD), docker-compose.recreate.yml (za recreate scenario), i docker-compose.eks1-manuel.yml (za ručnu implementaciju sa lokalnim procesom izgradnje).

4.5. Implementacija eksperimenata

Eksperimenti su implementirani direktno na opisanom sistemu, uz minimalne izmjene konfiguracije između scenarija.

Implementacija Eksperimenta 1:

U ručnom scenariju, developer se konektuje na VPS putem SSH, pristupa direktorijumu projekta, pokreće testove i Docker Compose. Docker Compose koristi Dockerfile.manuel koji izvršava kompletni Maven build unutar kontejnera. Vrijeme se mjeri štopericom od početka SSH konekcije do potvrde dostupnosti.

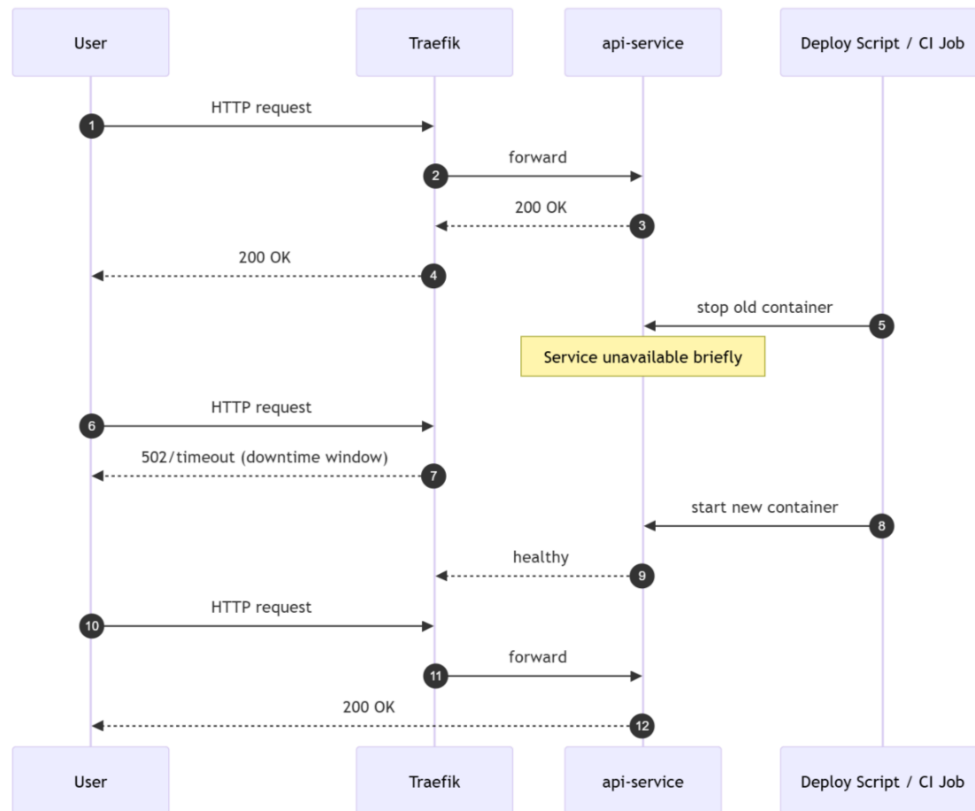
Važno je napomenuti da je tokom prvog ručnog pokušaja implementacija značajno duže trajala (317 sekundi) jer SSH ključ za Git *pull* komandu nije imao standardno id_ed25519 ime, pa je bilo potrebno kreirati SSH config fajl za povezivanje ključa. Ovakve nepredviđene situacije su tipični primjeri problema koje automatizacija eliminiše.

U CI/CD scenariju, *workflow* se pokreće ručno putem GitHub Actions interfejsa (workflow_dispatch *trigger*). *Pipeline* automatski izvršava sve korake: izgradnja, testove, kreiranje slika, objavljivanje na *registry*, i implementaciju na VPS. Vrijeme se očitava iz logova GitHub Actions.

Implementacija Eksperimenta 2:

Oba scenarija u drugom eksperimentu koriste workflow_dispatch *trigger* kako bi se eksperiment mogao ponoviti više puta bez potrebe za dodatnim izmjenama i slanjem koda na repozitorijum.

U recreate scenariju, *workflow* automatski mjeri period nedostupnosti korišćenjem Unix *timestamp*-a. Skripta zaustavlja api-blue, bilježi početak, pokreće novu verziju, čeka *health check* u *loop*-u (30 pokušaja svake 2 sekunde), i bilježi kraj. Za povratak na prethodnu verziju, proces se ponavlja sa starim tagom slike.



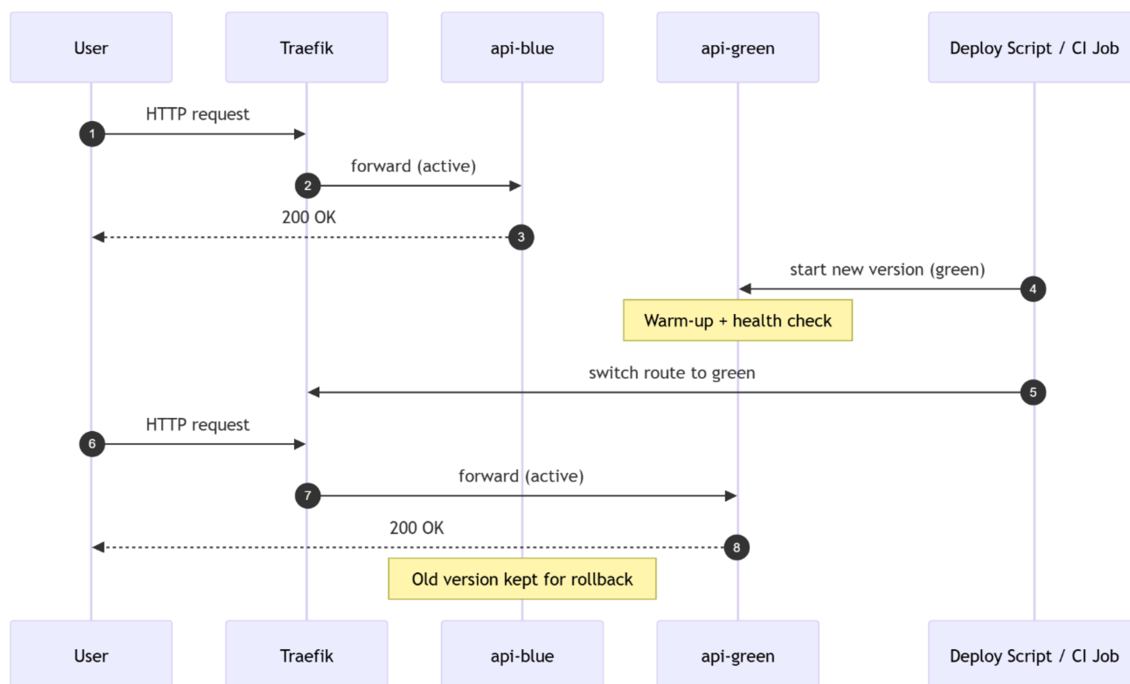
Slika 4. Recreate implementacija

Na Slici 4 prikazan je dijagram recreate implementacije koji ilustruje proces implementacije nove verzije aplikacije, pri čemu se postojeća instanca najprije gasi, a zatim pokreće nova. Tokom perioda između gašenja stare verzije i uspješnog završetka *health check*-a nove instance dolazi do kratkotrajne nedostupnosti servisa, nakon čega se saobraćaj ponovo normalno usmjerava i aplikacija nastavlja sa radom.

U Blue/Green scenariju, api-green se pokreće paralelno korišćenjem Docker Compose profila. Interno se provjerava zdravlje: *docker compose exec api-green curl localhost:8080/actuator/health*. Nakon potvrde, Traefik konfiguracija se mijenja sed komandom i Traefik se restartuje.

Traefik ima *watch: true* opciju za file provider, ali sed -i komanda kreira novi *inode* (novi fajl), a Docker volume mount prati stari *inode*. Zbog toga je potreban *docker compose restart traefik* nakon svake promjene. U produkciji sa alatima poput Kubernetes-a, Consul-a ili Traefik API-ja, prebacivanje saobraćaja se vrši gotovo trenutno (ispod 1 sekunde). U ovom eksperimentu korišćena je improvizacija koja demonstrira princip Blue/Green implementacije, uz minimalni *overhead* od restarta Traefik-a koji traje oko 0.6 sekundi.

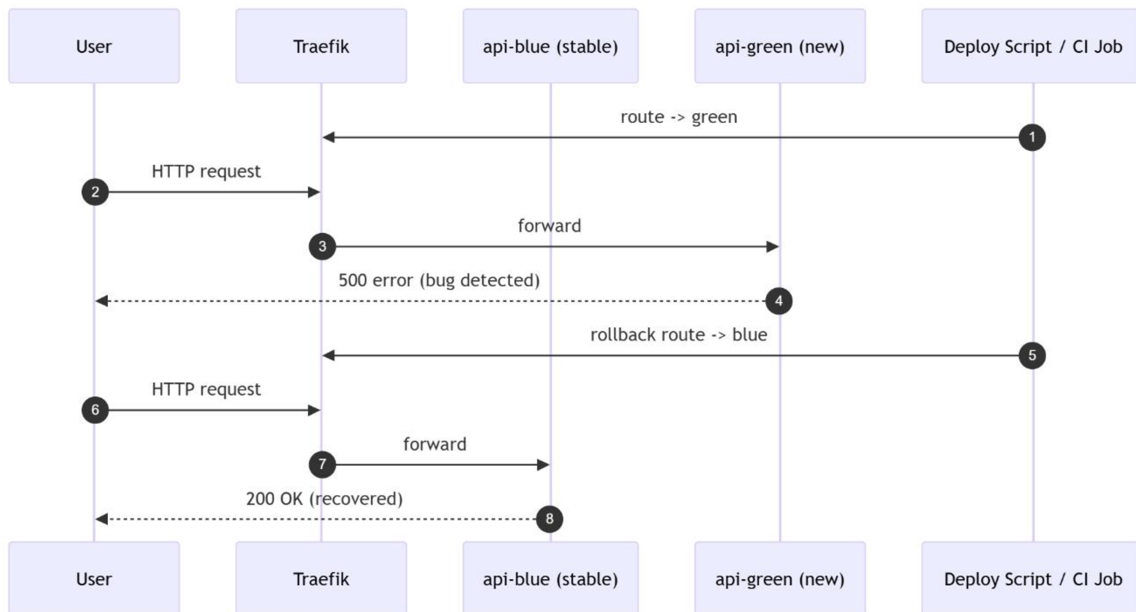
S obzirom da Blue/Green scenario nema eksplicitne *timestamp*-e za period nedostupnosti unutar *workflow*-a, mjerenje je rađeno eksterno sa lokalnog računara korišćenjem dva paralelna terminala. Oba terminala slala su HTTP zahtjeve ka serveru svakih 0,2 sekunde — prvi ka /actuator/health *endpointu*, a drugi ka /notifycheck *endpointu*. Uz svaki odgovor bilježeni su HTTP status kod, trenutno vrijeme i vrijednost X-Service-Version *headera*. Na osnovu ovih zapisa precizno je određen trenutak kada servis prestaje da odgovara i kada ponovo postaje dostupan, što je omogućilo izračunavanje perioda nedostupnosti i trajanja povratka na prethodnu verziju.



Slika 5. Blue/Green implementacija sa prebacivanjem saobraćaja

Na Slici 5 prikazan je tok Blue/Green procesa implementacije kroz četiri faze. U početnom stanju sav saobraćaj usmjeren je ka aktivnoj api-blue instanci putem Traefik *reverse*

proxy-a. U drugoj fazi pokreće se api-green kao nezavisni kontejner unutar iste Docker mreže, pri čemu api-blue nastavlja nesmetano da prima korisničke zahtjeve. Nakon što interna provjera potvrdi ispravnost nove instance, Traefik konfiguracija se mijenja i saobraćaj se preusmjerava na api-green. Ključna prednost ovog pristupa vidljiva je u tome što api-blue ostaje pokrenuta i u svakom trenutku je dostupna za trenutni oporavak ukoliko se detektuje problem.



Slika 6. Povratak na prethodnu verziju u Blue/Green modelu

Slika 6 prikazuje scenario oporavka koji se aktivira kada nova verzija (api-green) ne prođe provjeru /notifycheck *endpointa*. Kako api-blue instanca nije bila ugašena tokom implementacije, oporavak se svodi isključivo na izmjenu Traefik konfiguracije i njen restart, bez potrebe za ponovnim pokretanjem ili preuzimanjem kontejnera. Ovaj dijagram naglašava fundamentalnu prednost Blue/Green strategije u odnosu na Recreate pristup: minimalno vrijeme oporavka koje ne zavisi od brzine pokretanja aplikacije, već samo od brzine rekonfiguracije *proxy* sloja.

5. Eksperimentalni rezultati i analiza

U ovom poglavlju prikazani su rezultati sprovedenih eksperimenata i njihova interpretacija u kontekstu postavljenih hipoteza. Svi eksperimenti su ponovljeni pet puta, a u analizi su korišćene prosječne vrijednosti kako bi se umanjio uticaj slučajnih varijacija u sistemskom opterećenju i mrežnim uslovima.

5.1. Ručni vs CI/CD proces isporuke

Prvi eksperiment imao je za cilj da utvrdi razliku između ručnog i automatizovanog procesa isporuke. Vrijeme isporuke mjereno je od trenutka početka procesa (SSH konekcija za ručni, pokretanje *workflow*-a za CI/CD) do trenutka kada je nova verzija aplikacije postala dostupna putem HTTP zahtjeva.

Rezultati pet mjerenja za svaki scenario prikazani su u Tabeli 2.

Tabela 2 – Pojedinačna mjerenja vremena isporuke (u sekundama)

Pokušaj	Ručni (s)	CI/CD (s)
1.	317	78
2.	177	66
3.	164	63
4.	142	56
5.	125	66

Prosječno vrijeme ručne implementacije iznosi 185.0 sekundi (SD = 68.4s), dok prosječno vrijeme CI/CD implementacije iznosi 65.8 sekundi (SD = 7.1s). Ovo predstavlja smanjenje vremena isporuke od 64.4%.

Sumirani prikaz prosječnih vrijednosti, standardne devijacije i procentualnog smanjenja dat je u Tabeli 3.

Tabela 3 – Sumirani rezultati Eksperimenta 1

Metrika	Ručni	CI/CD
Prosječno vrijeme (s)	185.0	65.8
Standardna devijacija (s)	68.4	7.1
Najbrže mjerenje (s)	125	56
Najsporije mjerenje (s)	317	78

Posebno je uočljiva velika varijabilnost u ručnom scenariju. Prvo mjerenje (317 sekundi) značajno odudara od ostalih jer je tokom prvog pokušaja došlo do problema sa SSH ključem – ključ nije imao standardno ime (id_ed25519), pa je bilo potrebno kreirati SSH config fajl za povezivanje. Ovakve nepredviđene situacije su tipičan primjer problema koje automatizacija eliminiše. Čak i nakon što je problem riješen, ručna mjerenja pokazuju veću varijabilnost (SD = 68.4s) u poređenju sa CI/CD mjerenjima (SD = 7.1s), što ukazuje na veću konzistentnost automatizovanog procesa.

CI/CD implementacija donosi nekoliko ključnih prednosti koje su uočene tokom eksperimenta:

- Automatsko pokretanje testova prije implementacije – u ručnom procesu programer može zaboraviti da pokrene testove i implementira neispravan kod.
- Git *commit* SHA kao tag slike umjesto generičkog *latest* taga – omogućava precizno verzionisanje i pripremu za mehanizme oporavka.
- *Maven dependency caching* u GitHub Actions smanjuje vrijeme faze izgradnje jer se zavisnosti ne preuzimaju svaki put.
- *Health check* sa *retry* logikom (10 pokušaja svake 2 sekunde) automatski verifikuje dostupnost servisa, dok u ručnom procesu developer ručno provjerava *endpointe*.

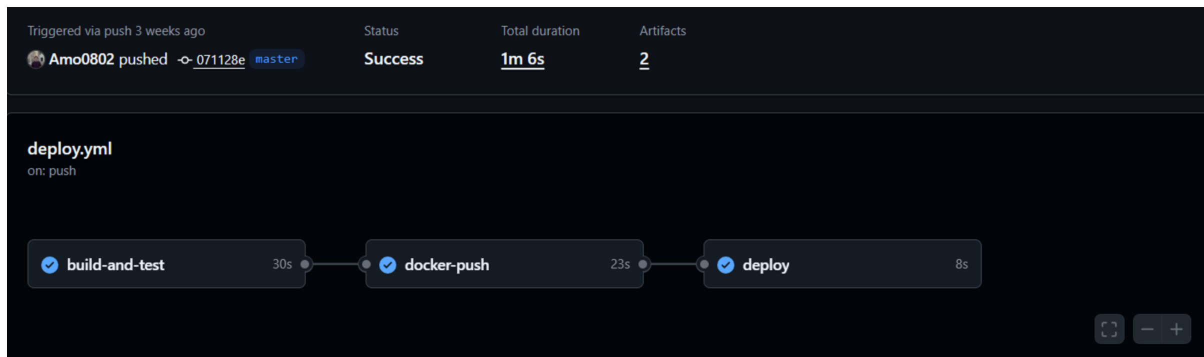
Ovi rezultati direktno potvrđuju glavnu hipotezu rada – automatizacija značajno smanjuje vrijeme isporuke softvera i povećava konzistentnost procesa.

Ručni proces implementacije uključivao je sljedeće korake: SSH konekciju na server, navigaciju do direktorijuma projekta, pokretanje testova komandom *./mvnw test*, navigaciju do infra direktorijuma, te pokretanje *docker compose up* komande sa *multi-stage* Dockerfile-om koji izvršava Maven *build* unutar kontejnera. Svaki od ovih koraka zahtijeva ručni unos komandi i praćenje rezultata.

Značajna varijabilnost ručnih mjerenja objašnjava se ne samo tehničkim faktorima, već i ljudskim. Developer mora pamtiti redoslijed komandi, navigirati između direktorijuma, i ručno interpretirati izlaz svake komande. Zamor i rutina tokom ponavljanja procesa mogu dovesti do preskakanja koraka (npr. zaboravljanje pokretanja testova) ili pogrešnog unosa.

CI/CD *pipeline* eliminisao je sve ručne korake i zamijenio ih automatizovanim procesom. Pokretanje *workflow*-a zahtijeva samo jedan klik na GitHub interfejsu. *Pipeline* sam preuzima kod, pokreće testove, vrši izgradnju slike, objavljuje na *registry*, i implementira na

server. Breakdown vremena po zadacima (prosječne vrijednosti iz GitHub Actions logova): build-and-test faza oko 30 sekundi (zahvaljujući Maven *dependency cache*-u), docker-push faza oko 20 sekundi, deploy faza oko 15 sekundi.



Slika 7. Uspješno izvršen CI/CD pipeline u GitHub Actions interfejsu

Na Slici 7 prikazan je primjer uspješno izvršenog CI/CD pipeline-a u GitHub Actions interfejsu. Vidljive su tri sekvencijalne faze (build-and-test, docker-push i deploy) sa ukupnim trajanjem od 1 minut i 6 sekundi, što odgovara drugom mjerenju u Tabeli 2.

Posebno je važna razlika u Dockerfile-ovima. Ručni scenario koristi *multi-stage* Dockerfile.manuel koji izvršava kompletni Maven *build* unutar Docker kontejnera, što zahtijeva preuzimanje svih zavisnosti i kompilaciju. CI/CD scenario koristi jednostavni Dockerfile.ci koji samo kopira gotov JAR artefakt u *runtime* sliku, jer je izgradnja već završena u prethodnom zadatku. Ova razlika značajno doprinosi smanjenju vremena faze implementacije.

5.2. Recreate vs Blue/Green

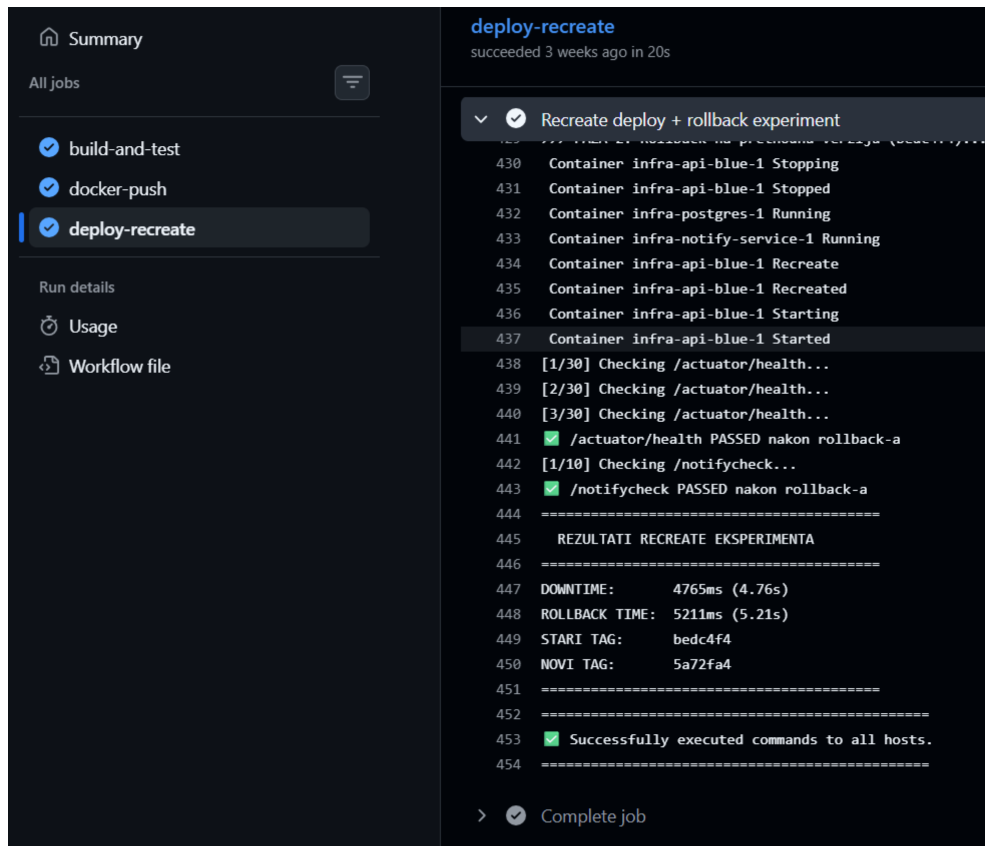
Drugi eksperiment fokusiran je na poređenje recreate i Blue/Green strategije implementacije nove verzije aplikacije. Za svaki scenario mjerena su dva parametra: period nedostupnosti sistema i vrijeme oporavka nakon detekcije greške.

Rezultati pet mjerenja prikazani su u Tabelama 4 i 5.

Tabela 4 – Mjerenja za Recreate strategiju (u sekundama)

Pokušaj	Downtime (s)	Rollback (s)
1.	4.74	7.34
2.	4.89	7.46
3.	4.76	5.21
4.	4.76	5.35
5.	4.73	5.36

U recreate modelu, prosječni period nedostupnosti iznosi 4.78 sekundi, što predstavlja period tokom kojeg sistem nije bio dostupan korisnicima. Ovaj period nedostupnosti nastaje jer se stara verzija kompletno zaustavlja prije pokretanja nove, a nova verzija mora proći kroz Spring Boot inicijalizaciju i uspostavljanje konekcije sa bazom podataka.



Slika 8. Log recreate eksperimenta u Github Actions

Na Slici 8 prikazan je log izvršavanja recreate eksperimenta iz GitHub Actions interfejsa. Vidljivi su automatski izmjereni rezultati: period nedostupnosti od 4765ms (4.76s) i vrijeme povratka na prethodnu verziju od 5211ms (5.21s), uz identifikaciju starog i novog taga slike putem Git commit SHA identifikatora.

Tabela 5 – Mjerenja za Blue/Green strategiju (u sekundama)

Pokušaj	Downtime (s)	Rollback (s)
1.	2.53	1.67
2.	2.77	1.40
3.	2.76	1.64
4.	2.50	1.51
5.	2.73	1.57

U Blue/Green modelu, prosječni period nedostupnosti iznosi 2.66 sekundi. Važno je napomenuti da ovaj period nedostupnosti u najvećoj mjeri proizilazi iz tehničke improvizacije korišćene u eksperimentu – Traefik restart koji je potreban jer sed -i komanda kreira novi *inode*, a Docker *volume mount* prati stari. U produkcijskim okruženjima sa alatima kao što su Kubernetes, Consul ili Traefik API, prebacivanje saobraćaja se vrši gotovo trenutno, pa bi realan period nedostupnosti bio blizak nuli. Uprkos ovom ograničenju, Blue/Green strategija pokazuje 44% manji period nedostupnosti u odnosu na recreate.

Razlika u vremenu povratka na prethodnu stabilnu verziju je još izraženija. Recreate oporavak prosječno traje 6.14 sekundi jer zahtijeva potpuno zaustavljanje nove verzije, pokretanje stare verzije i čekanje na *health check*. Blue/Green oporavak prosječno traje samo 1.56 sekundi jer zahtijeva samo promjenu Traefik konfiguracije i restart – stara verzija (api-blue) je cijelo vrijeme bila aktivna i spremna za rad.

Na Slici 9 prikazan je primjer eksternog mjerenja Blue/Green implementacije pomoću dva paralelna terminala. Lijevi terminal prati /actuator/health endpoint, a desni /notifycheck endpoint. Vidljiv je prelaz sa X-Service-Version: blue na green, kratkotrajan period sa HTTP 404 i 503 greškama tokom prebacivanja saobraćaja, te povratak na blue nakon detekcije greške.

```
X-Service-Version: blue
18:29:40.382 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:40.793 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:41.202 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:41.613 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:42.121 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:42.639 | 18:29:42.941 | 18:29:43.348 | 18:29:43.657 | 18:29:44.166 | 18
:29:44.475 | 18:29:44.755 | 18:29:45.043 | HTTP/1.1 404 Not Found
18:29:45.404 | HTTP/1.1 200 OK
X-Service-Version: green
18:29:45.774 | HTTP/1.1 200 OK
X-Service-Version: green
18:29:46.117 | 18:29:46.423 | 18:29:46.691 | 18:29:47.036 | 18:29:47.445 | 18
:29:47.753 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:48.165 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:48.575 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:48.982 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:49.496 | HTTP/1.1 200 OK
X-Service-Version: blue
```

```
X-Service-Version: blue
18:29:39.981 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:40.382 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:40.794 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:41.202 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:41.613 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:42.122 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:42.639 | 18:29:42.941 | 18:29:43.349 | 18:29:43.655 | 18:29:44.166 |
18:29:44.476 | 18:29:44.754 | 18:29:45.041 | HTTP/1.1 404 Not Found
18:29:45.404 | HTTP/1.1 503 Service Unavailable
X-Service-Version: green
18:29:45.912 | 18:29:46.270 | 18:29:46.626 | 18:29:46.923 | 18:29:47.243 |
18:29:47.547 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:47.962 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:48.371 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:48.781 | HTTP/1.1 200 OK
X-Service-Version: blue
18:29:49.189 | HTTP/1.1 200 OK
X-Service-Version: blue
```

Slika 9. Eksterno mjerenje Blue/Green implementacije putem dva paralelna terminala

Posebno je važna i konzistentnost mjerenja. Standardna devijacija perioda nedostupnosti za recreate je 0.06s, a za Blue/Green 0.12s, što ukazuje na visoku ponovljivost rezultata u oba scenarija. Kod oporavka, recreate pokazuje veću varijabilnost (SD = 1.03s) što se objašnjava varijacijom u vremenu pokretanja Spring Boot aplikacije.

Uporedni prikaz prosječnih vrijednosti i standardne devijacije za oba scenarija dat je u Tabeli 6.

Tabela 6 – Sumirani rezultati Eksperimenta 2

Metrika	Recreate	Blue/Green
Prosječni downtime (s)	4.78	2.66
SD downtime (s)	0.06	0.12
Prosječni rollback (s)	6.14	1.56
SD rollback (s)	1.03	0.10

Ovi rezultati potvrđuju da izbor strategije implementacije ima značajan uticaj na pouzdanost isporuke i vrijeme oporavka sistema. Blue/Green pristup pokazao se znatno stabilnijim u scenariju implementacije neispravne verzije, sa 75% bržim povratkom na stabilnu verziju.

U recreate scenariju, period nedostupnosti nastaje jer se stara verzija kompletno zaustavlja komandom *docker compose stop api-blue*, čime sav saobraćaj počinje da vraća greške. Nova verzija mora proći kroz potpunu inicijalizaciju: pokretanje JVM-a, Spring Boot

auto-konfiguraciju, uspostavljanje konekcijskog *pool*-a sa PostgreSQL bazom, i registraciju *actuator endpointa*. Tek nakon toga sistem ponovo odgovara na zahtjeve.

Povratak na stabilnu verziju u recreate modelu dodatno produžava nedostupnost jer zahtijeva ponovno zaustavljanje (sada neispravne) verzije, postavljanje prethodnog taga slike, pokretanje kontejnera, i čekanje na kompletnu Spring Boot inicijalizaciju. Zato recreate oporavak (prosječno 6.14s) traje duže od inicijalnog perioda nedostupnosti (prosječno 4.78s).

Blue/Green strategija fundamentalno mijenja pristup implementaciji. Nova verzija (api-green) se pokreće kao nezavisan kontejner dok api-blue nastavlja da prima produkcijski saobraćaj. Interna provjera zdravlja (*docker compose exec api-green curl localhost:8080/actuator/health*) osigurava da je nova verzija potpuno inicijalizovana prije bilo kakvog prebacivanja.

Period nedostupnosti u Blue/Green scenariju nastaje isključivo tokom Traefik restarta, koji traje približno 0.6 sekundi. Ostatak mjerenog perioda nedostupnosti (do prosječnih 2.66s) odnosi se na propagaciju promjene i reuspostavljanje HTTP konekcija. U produkcijskom okruženju sa Kubernetes Ingress-om ili Traefik API-jem, ovo bi bilo eliminisano.

Oporavak je posebno efikasan jer api-blue nikada nije bio ugašen. Jedino što je potrebno je vratiti Traefik konfiguraciju na prvobitno stanje i restartovati Traefik. Api-blue odmah nastavlja da prima saobraćaj jer je cijelo vrijeme bio aktivan, sa uspostavljenim konekcijama ka bazi podataka i potpuno inicijalizovanim Spring Boot kontekstom. Zato Blue/Green oporavak traje samo 1.56 sekundi.

5.3. Uporedna analiza i diskusija

Analizom rezultata oba eksperimenta može se uočiti konzistentan trend: automatizacija i pravilno odabrane strategije implementacije značajno unapređuju efikasnost i pouzdanost isporuke softvera.

Najveće smanjenje vremena postignuto je eliminacijom ručnih koraka u prvom eksperimentu. CI/CD *pipeline* je 64.4% brži od ručnog procesa, pri čemu je konzistentnost rezultata znatno veća. Ručni proces je podložan nepredviđenim problemima, dok CI/CD proces eliminiše ovu kategoriju grešaka.

Drugi eksperiment je pokazao da Blue/Green strategija donosi značajne prednosti u pogledu smanjenja perioda nedostupnosti i posebno u pogledu brzine oporavka. Činjenica da

Blue/Green oporavak traje prosječno 1.56 sekundi naspram 6.14 sekundi za recreate, predstavlja značajnu razliku u kontekstu produkcijskih sistema gdje svaka sekunda nedostupnosti ima direktan finansijski uticaj.

Važno je naglasiti kvalitativne prednosti koje nisu direktno mjerljive vremenom ali su uočene tokom eksperimenata:

- Automatizacija eliminiše ljudske greške – developer ne može zaboraviti pokrenuti testove ili implementirati pogrešnu verziju.
- Git *commit* SHA tagovi omogućavaju precizno praćenje koja verzija koda je implementirana i jednostavan oporavak na bilo koju prethodnu verziju.
- *Health check*-ovi sa *retry* logikom automatski detektuju probleme, dok u ručnom procesu developer mora sam pratiti status servisa.
- Blue/Green strategija pruža sigurnosnu mrežu – stara verzija ostaje aktivna tokom implementacije i spremna za trenutni oporavak.

Uprkos ograničenjima eksperimentalnog okruženja (dva servisa, Traefik restart umjesto instant prebacivanja), dobijeni podaci jasno ukazuju na prednosti CI/CD i kontejnerizacije. U složenijim produkcionim sistemima sa većim brojem servisa, ove prednosti bi bile još izraženije jer se ručna kompleksnost povećava eksponencijalno sa brojem komponenti, dok CI/CD *pipeline* skalira linearno.

Potrebno je napomenuti da dio smanjenja vremena isporuke u CI/CD scenariju proizilazi iz činjenice da se faza izgradnje (*Maven build*) izvršava na GitHub-ovoj infrastrukturi, a ne lokalno na VPS-u. Dok ručni proces koristi multi-stage Dockerfile koji kompajlira kod na istom serveru koji obrađuje produkcijske zahtjeve, CI/CD pipeline razdvaja fazu izgradnje od faze implementacije. Ovo razdvajanje je samo po sebi prednost automatizacije. U ručnom procesu developer koristi resurse produkcijskog servera za izgradnju, dok CI/CD pristup taj teret premješta na eksterne *runner*-e.

Zbog ograničenog broja ponavljanja, statistička obrada ograničena je na deskriptivnu statistiku (aritmetička sredina i standardna devijacija). Za formalniju statističku validaciju potreban bi bio veći uzorak i primjena inferencijalnih metoda poput t-testa, što prevazilazi obim ovog rada ali predstavlja smjernicu za buduća istraživanja.

Eksperimenti nisu uključivali testiranje pod opterećenjem (*load testing*), čime bi se moglo ispitati ponašanje strategija implementacije pod realnim produkcijskim uslovima sa

konkurentnim korisničkim zahtjevima. PostgreSQL baza podataka korišćena je kao komponenta sistema radi realističnije arhitekture, ali sam proces migracije šeme baze nije bio dio eksperimentalnog scenarija. Fokus je bio isključivo na aplikativnom sloju isporuke.

Iako su eksperimenti sprovedeni na sistemu sa dva servisa, rezultati pružaju korisne uvide za produkcijska okruženja. Ručna implementacija jednog servisa traje u prosjeku 185.0 sekundi. Za sistem sa 10 servisa, ukupno ručno vrijeme bi moglo biti 10-15 puta veće (uzimajući u obzir dodatnu kompleksnost koordinacije), dok bi CI/CD *pipeline* mogao paralelno izgraditi i implementirati sve servise uz minimalno dodatno vrijeme.

Slično tome, Blue/Green prednosti rastu u složenijim sistemima. U sistemu sa više servisa koji zavise jedni od drugih, recreate strategija zahtijeva pažljiv redoslijed zaustavljanja i pokretanja (da bi se očuvale zavisnosti), dok Blue/Green omogućava atomično prebacivanje kompletnog sistema na novu verziju.

Korišćenje Git *commit* SHA taga za verzionisanje slika, koje je implementirano u ovom radu, pruža osnovu za napredne prakse poput canary implementacije (postepeno preusmjeravanje saobraćaja) i automatskog oporavka baziranog na metrikama.

6. Zaključak

Cilj ovog rada bio je ispitati uticaj automatizacije isporuke softvera primjenom CI/CD *pipeline*-a i kontejnerizacije na efikasnost i pouzdanost procesa implementacije. Istraživanje je sprovedeno kroz dva kontrolisana eksperimenta nad višeservisnom aplikacijom implementiranom u Java Spring Boot okruženju i kontejnerizovanom pomoću Docker tehnologije.

Rezultati prvog eksperimenta pokazali su da automatizovani CI/CD proces značajno smanjuje ukupno vrijeme isporuke u poređenju sa ručnim pristupom. Prosječno vrijeme ručne implementacije iznosilo je 185.0 sekundi, dok je CI/CD implementacija prosječno trajala 65.8 sekundi, što predstavlja smanjenje od 64.4%. Pored kvantitativnog smanjenja vremena, CI/CD proces je pokazao znatno veću konzistentnost – standardna devijacija od 7.1 sekundi naspram 68.4 sekundi u ručnom scenariju. Eliminacijom ručnih koraka, automatizovanim fazama za izgradnju i testiranje, te standardizovanom kontejnerskom implementacijom, postignuta je veća dosljednost procesa i eliminisana kategorija grešaka uzrokovanih ljudskim faktorom.

Drugi eksperiment ukazao je na važnost izbora strategije implementacije. Recreate model je pokazao prosječni period nedostupnosti od 4.78 sekundi i prosječno vrijeme oporavka od 6.14 sekundi. Blue/Green pristup je smanjio period nedostupnosti na 2.66 sekundi i oporavak na svega 1.56 sekundi. Posebno je značajna razlika u vremenu oporavka – Blue/Green strategija omogućava 75% brži oporavak sistema jer stara verzija ostaje aktivna i spremna za trenutno preusmjeravanje saobraćaja.

Važno je napomenuti da bi u produkcijskim okruženjima sa alatima poput Kubernetes-a ili Consul-a, gdje se prebacivanje saobraćaja vrši gotovo trenutno, prednosti Blue/Green strategije bile još izraženije. U ovom eksperimentu korišćena improvizacija sa Traefik restartom uvodi minimalni *overhead*, ali i sa njim su rezultati jasno demonstrirali superiornost Blue/Green pristupa nad recreate strategijom.

Na osnovu dobijenih rezultata može se zaključiti da su postavljene hipoteze potvrđene:

- Automatizacija procesa implementacije značajno smanjuje vrijeme isporuke i povećava konzistentnost procesa, potvrđujući glavnu hipotezu rada.
- CI/CD *pipeline* eliminiše greške uzrokovane ljudskim faktorom i razlikama između okruženja – kontejnerizacija obezbjeđuje da se ista verzija aplikacije izvršava u identičnim uslovima.

- Vrijeme oporavka sistema se značajno smanjuje primjenom Blue/Green strategije, koja omogućava oporavak bez ponovnog pokretanja aplikacije.

Potrebno je naglasiti određena ograničenja istraživanja. Eksperimenti su sprovedeni na VPS okruženju sa dva servisa, što predstavlja sistem ograničene kompleksnosti. U realnim produkcionim sistemima sa većim brojem servisa, distribuiranim okruženjem i povećanim opterećenjem, apsolutne vremenske razlike bi mogle biti drugačije. Međutim, relativne prednosti automatizacije i naprednih strategija implementacije bi se vjerovatno pojačale, jer ručna kompleksnost raste eksponencijalno sa brojem komponenti.

Dalja istraživanja mogla bi obuhvatiti primjenu orkestracionih platformi poput Kubernetes-a (gdje bi Blue/Green prebacivanje bilo gotovo trenutno), analizu uticaja horizontalnog skaliranja na proces isporuke, integraciju naprednih sistema za nadzor, te ispitivanje canary strategije implementacije koja postepeno preusmjerava saobraćaj na novu verziju.

Uprkos navedenim ograničenjima, rad demonstrira da čak i u relativno jednostavnom višeservisnom sistemu automatizacija isporuke donosi mjerljive i značajne koristi. Integracija CI/CD *pipeline*-a i kontejnerizacije predstavlja ne samo tehničko unapređenje procesa, već i strateški korak ka stabilnijem i efikasnijem razvoju softverskih sistema.

Literatura

1. Duvall, P.M., Matyas, S. & Glover, A. (2007). Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional. ISBN: 978-0-321-33638-5.
2. Forsgren, N. & Kersten, M. (2018). DevOps Metrics. Communications of the ACM, 61(4), pp. 44–48. DOI: 10.1145/3159169.
3. Forsgren, N., Humble, J. & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press. ISBN: 978-1-942788-33-1.
4. Humble, J. & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional. ISBN: 978-0-321-60191-9.
5. Kim, G., Humble, J., Debois, P. & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press. ISBN: 978-1-942788-00-3.
6. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 2014(239). ISSN: 1075-3583.
7. Meszaros, G. (2007). xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional. ISBN: 978-0-131-49505-0.
8. Mouat, A. (2016). Using Docker: Developing and Deploying Software with Containers. O’Reilly Media. ISBN: 978-1-491-91576-9.
9. Bass, L., Weber, I. & Zhu, L. (2015). DevOps: A Software Architect's Perspective. Addison-Wesley Professional. ISBN: 978-0-134-04984-7.
10. Poulton, N. (2019). Docker Deep Dive. Independently published. ISBN: 978-1-5272-1823-4.
11. Shahin, M., Babar, M.A. & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, 5, pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.
12. Turnbull, J. (2014). The Docker Book: Containerization is the New Virtualization. James Turnbull. ISBN: 978-0-988-82023-2.

13. DevOps Research and Assessment (DORA). Accelerate State of DevOps Report 2024. Google Cloud, 2024. https://dora.dev/research/2024/dora-report/
14. Fowler, M. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html
15. Fowler, M. Blue-Green Deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html
16. Docker Documentation. https://docs.docker.com
17. Traefik Documentation. https://doc.traefik.io/traefik
18. GitHub Actions Documentation. https://docs.github.com/en/actions

Dodatak diplomskom radu (Annex)

1. CI/CD – Kontinuirana integracija i kontinuirana isporuka – praksa automatizovanog testiranja i isporuke softvera koja omogućava brže i pouzdanije ažuriranje aplikacija.

2. Kontejnerizacija – Tehnika pakovanja aplikacije zajedno sa svim njenim zavisnostima u izolovane cjeline (kontejnere) radi konzistentnog pokretanja u različitim okruženjima.

3. Docker – Platforma otvorenog koda za kreiranje, pokretanje i upravljanje kontejnerima, koja omogućava standardizovano pakovanje aplikacija.

4. DevOps – Skup praksi i kulturnih principa koji objedinjuju razvoj softvera (Dev) i IT operacije (Ops) u cilju bržeg i pouzdanijeg isporučivanja softvera.

5. Pipeline – Automatizovani niz koraka koji se izvršavaju sekvencijalno ili paralelno kako bi se softver izgradio, testirao i isporučio.

6. Deployment – Proces instaliranja i pokretanja nove verzije softvera na ciljnom okruženju, bilo da je to server, cloud ili lokalna mašina.

7. Git – Distribuirani sistem za kontrolu verzija koji omogućava praćenje promjena u izvornom kodu i kolaboraciju među programerima.

8. GitHub Actions – Platforma za automatizaciju radnih tokova integrirana u GitHub, koja omogućava kreiranje CI/CD *pipeline*-a direktno iz repozitorijuma.

9. Rollback – Proces vraćanja sistema na prethodnu stabilnu verziju softvera u slučaju da nova verzija sadrži greške ili probleme.

10. Blue/Green Deployment – Strategija isporuke u kojoj se koriste dva identična okruženja – jedno aktivno (blue) i jedno sa novom verzijom (green), uz prebacivanje saobraćaja nakon verifikacije.

11. Recreate Deployment – Strategija isporuke u kojoj se stara verzija potpuno zaustavi prije pokretanja nove, što uzrokuje određeni period nedostupnosti servisa.

12. Downtime – Period tokom kojeg je sistem ili servis nedostupan korisnicima, obično usljed održavanja ili isporuke nove verzije.

13. Mikroservisi – Arhitektonski obrazac u kome se aplikacija rastavlja na male, nezavisne servise koji komuniciraju putem mreže.

14. Monolitna arhitektura – Pristup razvoju softvera gdje su svi dijelovi aplikacije objedinjeni u jednu cjelinu, za razliku od mikroservisne arhitekture.

15. Docker Compose – Alat za definisanje i pokretanje višekontejnerskih Docker aplikacija pomoću jednog konfiguracionog fajla.

16. Docker Image – Šablon samo za čitanje koji sadrži instrukcije za kreiranje Docker kontejnera, uključujući kod, biblioteke i zavisnosti.

17. Dockerfile – Tekstualni fajl koji sadrži sve komande potrebne za izgradnju Docker slike, korak po korak.

18. Kontejner – Laka, izolovana instanca koja se pokreće na osnovu Docker slike i sadrži sve potrebno za rad aplikacije.

19. Virtualizacija – Tehnologija koja omogućava pokretanje više virtualnih mašina na jednom fizičkom serveru, sa potpunom izolacijom operativnih sistema.

20. Cloud Computing – Model isporuke IT resursa putem interneta, uključujući servere, skladište, baze podataka i softver na zahtjev.

21. Kontinuirana integracija – Praksa čestog spajanja promjena koda u zajedničko skladište, praćena automatizovanim testiranjem radi ranog otkrivanja grešaka.

22. Kontinuirana isporuka – Praksa automatizovanog pripremanja koda za isporuku u produkcijsko okruženje nakon uspješnog prolaska kroz sve faze testiranja.

23. Kontinuirani deployment – Proširenje kontinuirane isporuke gdje se svaka promjena koja prođe testove automatski postavlja u produkciju bez ljudske intervencije.

24. Build – Proces kompajliranja izvornog koda i generisanja izvršnog artefakta koji se može pokrenuti ili instalirati.

25. Artefakt – Rezultat procesa izgradnje softvera, poput JAR fajla, Docker slike ili izvršnog programa, spreman za isporuku.

26. Repozitorijum – Centralizovano mjesto za čuvanje izvornog koda, konfiguracija i dokumentacije projekta, najčešće upravljano pomoću Git-a.

27. Branch – Odvojena linija razvoja u Git-u koja omogućava rad na novim funkcionalnostima bez uticaja na glavni kod.

28. Merge – Proces spajanja promjena iz jedne grane u drugu unutar sistema za kontrolu verzija.

29. Pull Request – Zahtjev za pregled i spajanje koda iz jedne grane u drugu, koji omogućava timsku reviziju prije integracije.

30. Commit – Snimak promjena u izvornom kodu koji se čuva u istoriji repozitorijuma sa opisnom porukom.

31. Workflow – Definisani slijed koraka u automatizovanom procesu, kao što je CI/CD *pipeline* u GitHub Actions.

32. YAML – Format za serijalizaciju podataka koji se koristi za konfiguraciju CI/CD *pipeline*-a i Docker Compose fajlova.

33. Traefik – Reverzni proksi i load balancer dizajniran za rad sa kontejnerima, koji automatski otkriva servise i usmjerava saobraćaj.

34. Load Balancer – Komponenta koja raspodjeljuje dolazni mrežni saobraćaj između više servera radi poboljšanja performansi i dostupnosti.

35. Reverzni proksi – Server koji stoji između klijenata i backend servera, prosljeđujući zahtjeve i pružajući dodatnu sigurnost i kontrolu.

36. Spring Boot – Java framework koji pojednostavljuje kreiranje samostalnih, produkcijski spremnih aplikacija sa minimalnom konfiguracijom.

37. Java – Objektno orijentisani programski jezik širokog spektra primjene, poznat po principu platformske nezavisnosti.

38. API – Aplikativni programski interfejs – skup pravila i protokola koji omogućavaju komunikaciju između različitih softverskih sistema.

39. REST API – Arhitektonski stil za dizajn mrežnih aplikacija koji koristi HTTP metode za pristup i manipulaciju resursima.

40. HTTP – Protokol za prenos hiperteksta koji predstavlja osnovu komunikacije na World Wide Web-u.

41. Testiranje softvera – Proces evaluacije softvera radi otkrivanja razlika između očekivanog i stvarnog ponašanja sistema.

42. Unit test – Test koji provjerava funkcionalnost pojedinačne komponente ili metode u izolaciji od ostatka sistema.

43. Integracioni test – Test koji provjerava ispravnost interakcije između više komponenti ili servisa sistema.

44. Automatizovano testiranje – Korištenje alata i skripti za automatsko izvršavanje testova bez ručne intervencije.

45. Kvalitet koda – Mjera koliko dobro izvorni kod zadovoljava standarde čitljivosti, održivosti, performansi i funkcionalnosti.

46. Metrika – Kvantitativna mjera koja se koristi za procjenu performansi, kvaliteta ili efikasnosti procesa ili sistema.

47. Prosječno vrijeme isporuke – Metrika koja mjeri koliko vremena u prosjeku prođe od iniciranja do završetka procesa isporuke softvera.

48. Stopa uspješnosti – Procenat uspješno izvršenih isporuka u odnosu na ukupan broj pokušaja.

49. Konzistentnost – Osobina sistema da proizvodi iste rezultate pod istim uslovima, što je ključno za pouzdanu isporuku softvera.

50. Skalabilnost – Sposobnost sistema da efikasno raste i prilagođava se povećanom obimu posla ili broju korisnika.

51. Infrastruktura kao kod – Praksa upravljanja i konfigurisanja infrastrukture pomoću mašinski čitljivih konfiguracijskih fajlova umjesto ručnih procesa.

52. Orkestracija – Automatizovano upravljanje, koordinacija i raspoređivanje kontejnera u distribuiranom sistemu.

53. Kubernetes – Platforma otvorenog koda za automatizovanu orkestraciju kontejnera, njihovo skaliranje i upravljanje.

54. Docker Hub – Javni registar za skladištenje i distribuciju Docker slika, koji omogućava dijeljenje kontejnerizovanih aplikacija.

55. Registry – Skladište za čuvanje i distribuciju Docker slika, koje može biti javno ili privatno.

56. Environment Variable – Varijabla okruženja – konfigurabilna vrijednost koja se koristi za podešavanje ponašanja aplikacije bez izmjene koda.

57. Port Mapping – Mapiranje portova između kontejnera i host mašine, čime se omogućava pristup servisima unutar kontejnera.

58. Volume – Mehanizam za trajno čuvanje podataka generisanih i korišćenih od strane Docker kontejnera.

59. Mreža kontejnera – Virtualna mreža koja omogućava komunikaciju između Docker kontejnera unutar istog ili različitih host sistema.

60. Health Check – Mehanizam za automatsku provjeru da li je servis ili kontejner funkcionalan i spreman za primanje zahtjeva.

61. Logging – Proces bilježenja događaja, grešaka i aktivnosti sistema radi praćenja rada i dijagnostike problema.

62. Monitoring – Kontinuirano praćenje performansi, dostupnosti i zdravlja sistema pomoću specijalizovanih alata.

63. Agile – Skup principa i praksi za razvoj softvera koji naglašavaju iterativni razvoj, saradnju i prilagodljivost promjenama.

64. Scrum – Agilni okvir za upravljanje projektima koji koristi kratke iteracije (sprintove) za inkrementalni razvoj proizvoda.

65. Verzionisanje – Praksa dodjeljivanja jedinstvenih oznaka različitim stanjima softvera radi praćenja promjena i upravljanja izdanjima.

66. Semantic Versioning – Sistem verzionisanja softvera u formatu MAJOR.MINOR.PATCH koji jasno komunicira prirodu promjena.

67. Konfiguracija – Skup parametara i podešavanja koji definišu ponašanje softvera ili sistema u određenom okruženju.

68. Multi-stage Build – Tehnika u Docker-u koja koristi više faza izgradnje radi kreiranja manjih i optimizovanijih finalnih slika.

69. Cache – Privremeno skladište podataka koje ubrzava ponovljene operacije izbjegavanjem ponovnog računanja ili preuzimanja.

70. Latencija – Vremensko kašnjenje između slanja zahtjeva i primanja odgovora u komunikaciji između sistema.

71. Propusnost – Količina podataka ili zahtjeva koje sistem može obraditi u jedinici vremena.

72. Fault Tolerance – Sposobnost sistema da nastavi sa radom čak i kada dođe do kvara jedne ili više komponenti.

73. Redundancija – Dupliranje kritičnih komponenti sistema kako bi se osigurala kontinuirana dostupnost u slučaju kvara.

74. Sigurnost – Zaštita sistema, podataka i komunikacija od neovlašćenog pristupa, modifikacije ili uništavanja.

75. Enkripcija – Proces pretvaranja podataka u nečitljiv oblik pomoću algoritama, radi zaštite od neovlašćenog pristupa.

76. Automatizacija – Primjena tehnologija i alata za izvršavanje zadataka bez ručne ljudske intervencije.

77. Efikasnost – Mjera koliko se efektivno resursi koriste za postizanje željenog rezultata uz minimalan utrošak.

78. Performanse – Sposobnost sistema da ispuni zahtjeve u pogledu brzine, odziva i obrade podataka.

79. Optimizacija – Proces poboljšanja performansi, efikasnosti ili kvaliteta sistema ili procesa.

80. Debugging – Proces identificiranja, analiziranja i ispravljanja grešaka u softveru.

81. Log fajl – Fajl koji sadrži zapise o događajima i aktivnostima sistema, koristan za dijagnostiku i analizu.

82. Shell skripta – Program napisan za komandni interpreter operativnog sistema koji automatizuje izvršavanje niza komandi.

83. Linux – Operativni sistem otvorenog koda koji se široko koristi kao osnova za servere, kontejnere i cloud infrastrukturu.

84. Server – Računar ili softverski program koji pruža usluge ili resurse drugim programima ili uređajima putem mreže.

85. Produkcijsko okruženje – Realno okruženje u kome softver radi i služi krajnjim korisnicima.

86. Staging okruženje – Okruženje koje simulira produkciju i koristi se za finalno testiranje prije isporuke softvera korisnicima.

87. Razvojno okruženje – Lokalno okruženje programera u kome se piše, testira i debuguje kod prije integracije.

88. Webhook – Mehanizam koji omogućava automatsko pokretanje akcije u jednom sistemu kada se događaj desi u drugom.

89. Trigger – Unaprijed definisani uslov ili događaj koji pokreće automatizovani proces, poput CI/CD *pipeline*-a.

90. Paralelizacija – Istovremeno izvršavanje više zadataka radi ubrzanja ukupnog procesa obrade.

91. Inovacija – Uvođenje novih ideja, metoda ili tehnologija koje donose poboljšanje u postojećim procesima ili stvaraju nove vrijednosti.

92. Kreativnost – Sposobnost stvaranja originalnih rješenja i pristupa problemima na nov i neočekivan način.

93. Transformacija – Proces temeljne promjene u načinu funkcionisanja sistema, organizacije ili procesa, često potaknut novim tehnologijama.

94. Saradnja – Zajednički rad pojedinaca ili timova na ostvarivanju zajedničkog cilja, ključna za uspješan razvoj softvera.

95. Agilnost – Sposobnost brzog prilagođavanja promjenama zahtjeva i uslova uz održavanje kvaliteta i efikasnosti.

96. Pouzdanost – Osobina sistema da dosledno ispunjava svoju funkciju bez kvarova tokom očekivanog perioda rada.

97. Feedback – Povratna informacija o rezultatima rada koja služi za unapređenje procesa i proizvoda.

98. Best Practice – Dokazano najefikasniji postupak ili metoda za postizanje željenog rezultata u određenoj oblasti.

99. Open Source – Model razvoja softvera u kome je izvorni kod javno dostupan za korišćenje, modifikaciju i distribuciju.

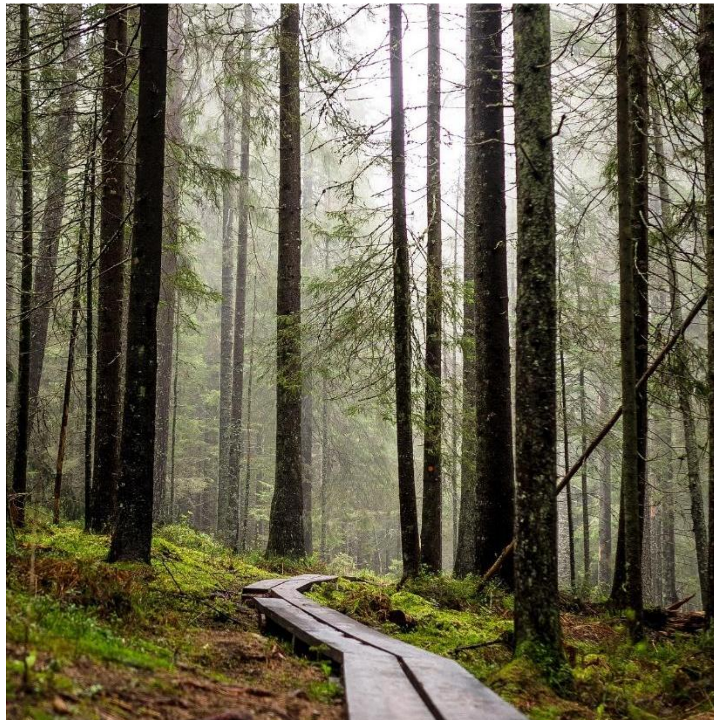
100. Digitalna transformacija – Promjena poslovnih procesa i strategija kroz primjenu digitalnih tehnologija radi povećanja efikasnosti i konkurentnosti.

Izvor: Pixabay

Slika 1. Slika zupčanika koji se međusobno uklapaju simbolizuje automatizaciju i povezanost procesa. Svaki zupčanik predstavlja korak u CI/CD *pipeline*-u, a njihovo savršeno uklapanje pokazuje kako automatizovani koraci rade zajedno za efikasnu isporuku softvera.

Izvor: Pixabay

Slika 2. Kontejnerski brod na otvorenom moru simbolizuje kontejnerizaciju u softveru. Kao što brodski kontejneri standardizuju transport robe, tako Docker kontejneri standardizuju pakovanje i isporuku aplikacija.

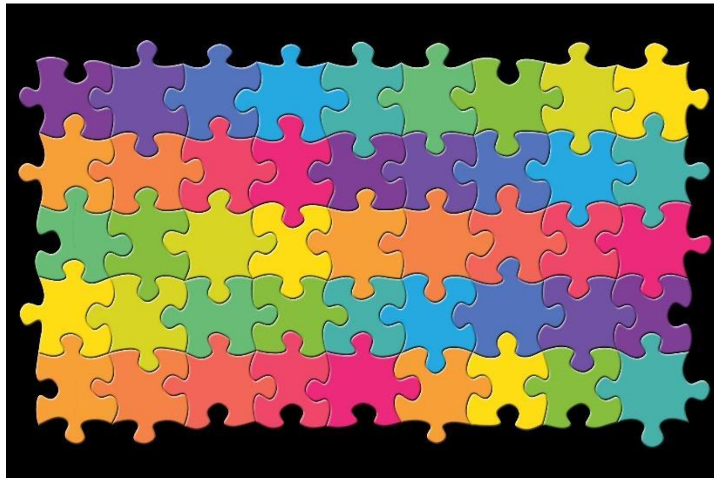


Izvor: Pixabay

Slika 3. Putanja kroz šumu nam šalje poruku da put do cilja nije uvijek pravolinijski. U razvoju softvera, *pipeline* nas vodi korak po korak od koda do produkcije, baš kao što staza vodi planinarima do vrha.

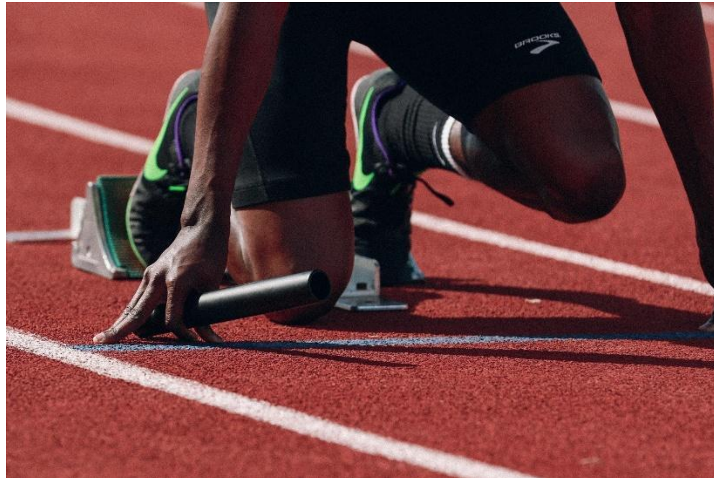
Izvor: Pixabay

Slika 4. Ova slika mosta koji povezuje dvije strane simbolizuje CI/CD kao most između razvoja i operacija. DevOps kultura gradi most saradnje koji omogućava brži i pouzdaniji protok softvera od ideje do korisnika.



Izvor: Pixabay

Slika 5. Slika slagalice čiji se djelovi uklapaju predstavlja integraciju različitih komponenti sistema. Svaki mikroservis je dio veće slike, a tek kada se svi pravilno povežu, nastaje funkcionalna cjelina.



Izvor: Pixabay

Slika 6. Slika trkača na startu simbolizuje brzinu isporuke softvera. Automatizacija CI/CD procesa omogućava timovima da budu brži od konkurencije, smanjujući vrijeme od ideje do realizacije.

Izvor: Pixabay

Slika 7. Fotografija tima koji radi zajedno podsjeća nas da ni najnaprednija tehnologija ne može zamijeniti ljudsku saradnju. DevOps je prije svega kulturna promjena koja zahtijeva timski rad, komunikaciju i povjerenje.

Izvor: Pixabay

Slika 8. Slika šahovske table simbolizuje stratešku prirodu strategija implementacije. Kao i u šahu, svaki potez mora biti promišljen.

Izvor: Pixabay

Slika 9. Ledeni brijeg sa vidljivim i skrivenim dijelom simbolizuje složenost softverskih sistema. Ono što korisnik vidi je samo vrh, ispod površine krije se čitava infrastruktura kontejnera, *pipeline*-a i automatizovanih procesa.

Izvor: Pixabay

Slika 10. Biljka koja raste kroz beton pokazuje da je napredak moguć i u najtežim uslovima.

Izvor: Pixabay

Slika 11. Slika izlaska sunca nad planinom simbolizuje budućnost DevOps-a i automatizacije. Kako se tehnologije poput CI/CD-a i kontejnerizacije dalje razvijaju, otvaraju se nove mogućnosti za brži, pouzdaniji i efikasniji razvoj softvera.