Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nejc Ačkun

Pregled in primerjava platform s CI/CD funkcionalnostmi

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Aljaž Zrnec
Somentor: asist. dr. Marko Poženel

Ljubljana, 2020

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Pregled in primerjava platform s CI/CD funkcionalnostmi

Tematika naloge:

Neprekinjena integracija in postavitev (ang. CI/CD - Continuous Integration/Continuous Deployment) sta v okviru procesa razvoja programske opreme zelo pomembni aktivnosti. V okviru diplomske naloge primerjajte tri platforme, ki CI/CD omogočajo, pri čemer se osredotočite na platformo GitHub in njeno izvedbo tega postopka. Implementirajte praktični primer izvajanja testov z uporabo GitHub ter na koncu izbrane platforme ovrednotite.

Zahvaljujem se mentorju viš. pred. dr. Aljažu Zrnecu in somentorju asist. dr. Marku Poženelu za vso pomoč pri izdelavi diplomske naloge. Zahvaljujem se tudi staršema za vso podporo med časom študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	application program interface	programski vmesnik
BTO	business technology optimization	optimizacija poslovnih tehnologij
CD	continuous delivery or deployment	neprekinjena dostava oz. postavitev
CDE	continuous delivery	neprekinjena dostava
CI	continuous integration	neprekinjena integracija
HTTP	hypertext transfer protocol	protokol za prenos hiperteksta
JS	JavaScript	JavaScript
SCM	Source Code Management	upravljanje izvorne kode
VCS	Version Control System	sistem za upravljanje različic
YAML	YAML Ain't Markup Language	YAML Ain't Markup Language

Povzetek

Naslov: Pregled in primerjava platform s CI/CD funkcionalnostmi

Avtor: Nejc Ačkun

Testiranje programske opreme danes igra zelo pomembno vlogo med razvojnim procesom. Ampak ročno testiranje je dokaj zamudno, zato postopek testiranja avtomatiziramo. Avtomatiziramo lahko tudi dostavo novih različic opreme k uporabnikom. To imenujemo neprekinjena integracija/postavitev (CI/CD). Na voljo je veliko platform, ki omogočajo funkcionalnosti CI/CD. Toda katero izbrati? V nalogi so predstavljene in med seboj primerjane tri platforme - GitHub, Bitbucket in GitLab. Osredotočil sem se na njihove CI/CD funkcionalnosti in izvedbo le teh, primerjal pa sem tudi monetizacijo platform in funkcije, ki jih ponujajo. Ugotovil sem, da je izvedba CI/CD funkcionalnosti zelo podobna na vseh izbranih platformah, ter da se vsaka platforma s svojimi funkcijami osredotoča na drug del trga. Za konec sem podal še priporočila za različne vrste ekip in podjetji.

Ključne besede: neprekinjena integracija, neprekinjena postavitev, platforme, GitHub, Bitbucket, GitLab.

Abstract

Title: Review and comparison of platforms with CI/CD functionalities

Author: Nejc Ačkun

Software testing plays a vital role in today's software development. But manual testing is often quite time intensive, so the testing procedure is automated. We can also automate the delivery process of new versions of our software to the end customer. This is called continuous integration/deployment. There is a wide variety of platforms that support CI/CD functionalities, but which one to choose? In this document I have reviewed and compared three different platforms - GitHub, Bitbucket and GitLab. I focused on the execution of their CI/CD functionalities, while also comparing their monetization and other functions they offer. My analysis shows that the execution of CI/CD functionality is very similar on all three compared platforms and that each platform focuses on a different part of the market. In the final chapter I also added some recommendations for different types of teams and companies.

Keywords: continuous integration, continuous deployment, platforms, GitHub, Bitbucket, GitLab.

Poglavje 1

Uvod

Programska oprema v današnjem življenju igra nepogrešljivo vlogo. Praktično na vsakem koraku nas spremlja neka vrsta programske opreme. Na mobilnem telefonu imamo aplikacije za komunikacijo, igre, sam operacijski sistem itd. Na računalniku uporabljamo urejevalnike teksta, brskalnik - tudi skoraj vsaka spletna stran, ki jo obiščemo, je v ozadju neke vrste aplikacija.

Že dolgo pa razvijanje programske opreme ni zaključen proces. Lansiranje nekega izdelka na trg še zdaleč ne pomeni konec procesa razvoja te programske opreme. Uporabniki danes pričakujejo redne popravke in nove funkcije. Ampak kako razvijalci zagotovijo, da bodo vse funkcije programske opreme delovale tudi po implementaciji popravkov? Nove funkcije imajo tudi velik potencial za povzročitev težav pri drugih, že obstoječih funkcijah. To težavo lahko rešimo s testiranjem programske opreme.

Namen testiranja programske opreme je dokazati, da ta deluje tako, kot je bilo načrtovano. Testiranje prav tako opozori razvijalce na napake, ki mogoče niso tako očitne. A testiranje predstavlja velik napor pri razvijanju programske opreme. Večji kot je obseg kode, daljše ter bolj zamudno postaja testiranje. Problem nastane tudi pri večjih razvojnih oddelkih, kjer več razvijalcev razvija med seboj neodvisne spremembe. Ta problem rešuje avtomatizirano testiranje, in ostale prakse, ki jih določa CI oz. neprekinjena integracija.

Lahko pa avtomatiziramo tudi samo postavitev novih različic naše programske opreme in s tem dostavo novosti do končnega uporabnika. Tako nove različice, na katerih se uspešno izvedejo vsi testi, takoj nadaljujejo pot proti uporabnikom. Temu postopku pravimo CD oz. neprekinjena dostava/postavitev.

Obstaja veliko platform, ki omogočajo uporabo funkcionalnosti CI/CD. Toda katero izbrati?

V diplomski nalogi bom na kratko opisal testiranje programske opreme, kaj sploh sta CI ter CD, kdaj se je pričela uporaba ter kaj ponujata ekipam, ki funkcionalnosti CI/CD uporabljajo. Nato bom predstavil tri platforme, ki omogočajo funkcionalnosti CI/CD: GitHub, Bitbucket ter GitLab. Na kratko jih bom opisal, nato pa bom na vsako izmed njih naložil enako aplikacijo in postavil cevovod CI/CD. Postopke bom primerjal, na koncu pa bom platforme kar se da objektivno ocenil in napisal priporočila za različne primere uporabe (recimo priporočilo za majhno razvojno ekipo, za večje podjetje itd.).

Poglavje 2

Testiranje prog. opreme in CI/CD

Ko govorimo o CI/CD skoraj da moramo omeniti tudi sam proces testiranja programske opreme, saj velik del CI/CD predstavlja avtomatizacija tega postopka. V tem poglavju bom najprej predstavil zgodovino testiranja programske opreme, nato bom opisal testiranje nasploh (katere vrste/nivoje testiranja poznamo, kaj so največje prednosti testiranja itd.), končal pa bom z opisom pojmov CI ter CD.

2.1 Zgodovina testiranja

V petdesetih letih prejšnjega stoletja je bilo testiranje znano enostavno kot postopek, s katerim so programerji iskali napake v svojih programih. Definiranih ni bilo nobenih posebnih postopkov ali metod za doseganje tega cilja. Koncepti razhroščevanja in testiranja še niso bili jasno definirani in niso bili jasno ločeni. Alan Turing pa je v tem obdobju napisal članek, ki velja kot prvi članek napisan na temo testiranja programov. Članek opisuje Turingov test [59] - preizkus zmožnosti stroja oz. programa, da se obnaša inteligentno oz. človeško. Če to malce posplošimo, gre za dokazovanje, da program zadostuje našim potrebam. S tem opisom danes (delno) definiramo funkcijsko

testiranje [49].

V šestdesetih in sedemdesetih letih so se prvič pojavili predlogi o izčrpnem testiranju programske opreme [48, 54] - pregledu vseh možnih vhodnih podatkov skozi vse možne poti v programski opremi. Ugotovljeno je bilo, da je zaradi ogromne količine različnih možnosti takšno testiranje teoretično nemogoče [34, 53], prav tako pa bi se pojavile težave pri ugotavljanju napak pri sami specifikaciji. Ko je razvoj programske opreme napredoval skozi šestdeseta in sedemdeseta leta, se je definicija testiranja programske opreme spremenila v definicijo postopka, ki demonstrira pravilnost programa oz. da sistem ali program dela to kar mora [48].

V zgodnjih sedemdesetih se je kratkoročno pojavila tehnika preverjanja programske opreme med specifikacijo, načrtovanjem in implementacijo sistema s t.i. "dokazom pravilnosti" [40, 45, 42]. Za enostavne teste je bil koncept primeren, saj je preveriti delovanje in dokazati teoretično delovanje dokaj enostavno. Ampak, ker večji deli programa niso bili testirani je velik del napak ostal neodkrit do postopka implementacije. Koncept je bil označen kot neučinkovit [40].

V poznih sedemdesetih je bilo rečeno, da je testiranje proces izvajanja programske kode z namenom iskanja napak, ne pa dokazovanja da programska oprema deluje [62, 52]. Ta nova definicija je poudarila, da je dober testni primer tisti, ki ima največjo možnost, da odkrije novo, doslej še neodkrito napako - uspešen test pa tisti, ki to stori. Ta definicija je bila popolno nasprotje tistim, ki so bile v veljavi prej [48] - dokazovanje da nekaj deluje pravilno/dokazovanje da nekaj ne deluje pravilno.

V osemdesetih letih je bila definicija testiranja razširjena še na področje preprečevanja napak [51]. Predlagano je bilo, da je potrebna metodologija testiranja - predvsem, da mora biti testiranje prisotno čez celotno dobo razvijanja in da mora biti testiranje kontroliran proces [39]. Poudarjena je bila pomembnost testiranja ne samo programa, ampak tudi potreb/ciljev, načrtovanja, programske kode, in testov samih. Pojem "testiranje" se je v tem obdobju največkrat navezoval na testiranje sistema, ko je bil delujoč

program dostavljen naročniku. Danes takšnemu testiranju pravimo sistemsko testiranje [30].

V sredini osemdesetih let so se pojavila prva orodja za avtomatsko testiranje [46], katerih cilj je bil izboljšava učinkovitosti ter kvalitete končnega sistema oz. aplikacije. Pričakovano je bilo, da lahko računalnik opravi veliko več testov, bolj zanesljivo, kot človek. V začetku so bila ta orodja dokaj primitivna in brez možnosti pisanja naprednih skript.

V začetku devetdesetih je bila priznana pomembnost načrtovanja testov že od samega začetka razvoja. Definicija testiranja je bila spremenjena v načrtovanje, pisanje, vzdrževanje in poganjanje testov ter testnih okolji [48]. V tem obdobju so se pojavila tudi bolj napredna orodja [32, 27, 6] za avtomatsko testiranje - programi za snemanje/replikacijo akcij z vgrajenimi skriptnimi jeziki ter obsežnimi možnostmi poročanja rezultatov. S pojavom in priljubljenostjo interneta v sredini devetdesetih pa se je programska oprema pogosto razvijala brez definiranega modela testiranja, ker je zelo otežilo testiranje.

V začetku enaindvajsetega stoletja je podjetje Mercury Interactive (danes v lasti podjetja Hewlett-Packard) predstavilo še širšo definicijo testiranja, ko so predstavili koncept BTO oz. business technology optimization [48].

Danes pa je testiranje programske opreme še bolj pomembno kot kadarkoli prej. Z razvojem novih tehnologij in digitalizacijo skoraj vseh predelov našega življenja, so tolerance za še tako majhne napake zelo nizke.

2.2 Testiranje programske opreme

Namen testiranja programske opreme je dokazovanje, da program oz. sistem deluje pravilno in da odkrijemo napake preden sistem pride v uporabo. Programsko opremo testiramo z umetnimi podatki, storiti pa želimo dve stvari:

- Naročniku ali razvijalcu dokazati, da programska oprema izpolnjuje zahteve. Če govorimo o programski opremi, ki je narejena po meri naročnika to ponavadi dosežemo z vsaj enim testom za vsako zahtevo, ki

je definirana v seznamu oz. dokumentu zahtev. Če gre pa za generično programsko opremo, pa dokaz dosežemo z vsaj enim testom za vsako zahtevo, ki bo dosegla končno različico sistema.

- Najti želimo vhodne podatke oz. kombinacijo teh, kjer je obnašanje programske opreme nepravilno in ne ustreza specifikaciji. S tem preprečimo sistemske zrušitve, nezaželeno interakcijo z drugimi sistemi ipd.

Prvi točki pravimo validacijsko testiranje, kjer pričakujemo pravilno obnašanje sistema pri uporabi testnih scenarijev, ki simulirajo pričakovano uporabo sistema. Drugi točki pravimo pa testiranje za napake, kjer so testni scenariji sestavljeni z namenom odkrivanja napak in ni potrebno, da so podobni realni uporabi. Testiranje programske opreme pa ne more dokazati, da je programska oprema brez napak, saj se vedno lahko najde scenarij, ki povzroči napako in na katerega med testiranjem nismo pomislili.

Tipično bo komercialni sistemi testiran na treh nivojih [58]:

- **Razvojno testiranje** - testiranje sistema med procesom razvoja, z namenom odkrivanja hroščev in napak. V ta proces so ponavadi vključeni načrtovalci sistema in programerji.

- **Izdajno testiranje** - ločena testna ekipa testira končno različico sistema, preden je ta izdan uporabnikom. Glavni namen je preverjanje, če sistem odgovarja zahtevam naročnika.

- **Uporabniško testiranje** - kjer uporabniki testirajo sistem v lastnem okolju. V to kategorijo spadajo tudi testi sprejemljivosti, kjer se naročnik odloči, če je sistem pripravljen za uporabo ali pa potrebuje nadaljnji razvoj.

Testiranje je ponavadi mešanica ročnega in avtomatskega testiranja. Pri ročnem testiranju tester program zaganja z umetnimi podatki in primerja rezultate s pričakovanimi. Razlike in napake so nato posredovane razvijalcem.

Pri avtomatskem testiranju pa se testi izvedejo kot del programa ali skripte, ki se zažene vedno ko želimo sistem v razvoju testirati. Takšno testiranje je hitrejše od ročnega, sploh ko govorimo o regresijskem testiranju [35, 37, 36] - ponovnem izvajanju prejšnjih testov, da se prepričamo da spremembe v kodi ne povzročajo novih napak.

Na žalost popolna avtomatizacija testiranja ni mogoča. Avtomatski testi se načeloma ne uporabljajo za testiranje celotnih grafičnih vmesnikov, prav tako pa ne morejo pokazati, da program nima nezaželenih stranskih učinkov [58].

2.2.1 Razvojno testiranje

Razvojno testiranje vključuje vse testne dejavnosti, ki jih izvaja ekipa, ki program oz. sistem razvija. Ta tip testiranja razdelimo v tri stopnje:

- **Testiranje enot** - testiranje posameznih enot ali objektnih razredov. Pri testiranju enot se osredotočimo na testiranje funkcionalnosti objektov ali metod.
- **Testiranje komponent** - integriranje več posameznih enot v komponente. Pri testiranju komponent se osredotočamo na vmesnike komponent, ki omogočajo dostop do funkcij komponent.
- **Sistemsko testiranje** - nekaj komponent oz. vse komponente so integrirane v sistem in ta je testiran kot celota. Sistemsko testiranje se osredotoča na testiranje interakcije med komponentami.

Razvojno testiranje je primarno testiranje za napake, kjer je naš cilj odkrivanje napak v programski opremi ali sistemu in se ponavadi prepleta s procesom razhroščevanja.

2.2.2 Izdajno testiranje

Izdajno testiranje je proces testiranja določene različice sistema, ki je namenjena uporabi zunaj razvojne ekipe oz. v testnem okolju. Ponavadi gre

za različico namenjeno za naročnika, pri kompleksnih projektih pa gre lahko za različico namenjeno drugi razvojni ekipi, ki razvija sorodni sistem. Med izdajnim in sistemskim testiranjem obstajata dve pomembni razliki:

- Razvojna ekipa sistema naj ne bi bila odgovorna za izdajno testiranje.
- Izdajno testiranje je proces validacije, ki zagotavlja, da sistem odgovarja zahtevam in da je dovolj dober za uporabo pri naročniku. Sistemsko testiranje razvojne ekipe se osredotoča na odkrivanje napak v sistemu.

2.2.3 Uporabniško testiranje

Uporabniško ali naročniško testiranje je korak v procesu testiranja sistema, kjer uporabniki ali stranke zagotovijo vhodne podatke in nasvete za sistemsko testiranje. To lahko vključuje formalno testiranje sistema, ki ga je naročil zunanji dobavitelj. Lahko pa gre tudi za neformalen proces kjer uporabniki eksperimentirajo z novim izdelkom da preverijo, če jim ustreza in če odgovarja njihovim potrebam. Uporabniško testiranje je ključno, tudi po izvedbi celovitega sistemskega in izdajnega testiranja. Vplivi uporabniških delovnih okolij imajo lahko učinek na zanesljivost, zmogljivost, uporabnost in robustnost sistema.

Uporabniško testiranje delimo v tri skupine:

- **Alfa testiranje** - izbrana skupina uporabnikov programske opreme tesno sodeluje z razvojno ekipo pri testiranju zgodnjih izdaj programske opreme oz. sistema.
- **Beta testiranje** - izdaja sistema je postane na voljo širši skupini uporabnikov, kar jim omogoči eksperimentiranje in s tem prijavo napak, ki jih najdejo, razvijalcem sistema.
- **Testiranje sprejemljivosti** - stranke testirajo sistem in podajo odločitev ali je sistem pripravljen za postavitev v okolje naročnika.

2.3 Sodoben razvoj programske opreme

Danes večina programske opreme začne življenjski cikel kot prazen repozitorij na eni izmed platform za upravljanje z izvorno kodo (angl. Source Code Management - SCM) oz. upravljanje z različicami (angl. Version Control System - VCS).

Uporaba repozitorija nam omogoča enostavni nadzor nad spremembami, ki jih dodajamo v izvorno kodo z uporabo sistema uveljavitev (angl. commits) - spremembe uveljavimo na repozitorij, kar repozitorij prenese iz starega stanja v novo, posodobljeno stanje. Za vsako datoteko se hrani celotna zgodovina sprememb, kar nam olajša primerjanje različic in iskanje napak v izvorni kodi. V ekstremnih primerih, ko pride do kritičnih napak, nam repozitorij ponuja tudi možnost enostavne razveljavitve sprememb v projektu. Prednosti tega sistema se pokažejo predvsem na projektih, kjer na izvorni kodi dela več ljudi naenkrat.

Še ena pomembna funkcija, ki jo omogočajo platforme je ustvarjanje vejitev v repozitoriju. V repozitoriju lahko definiramo več različnih vej in tako ustvarimo več neodvisnih različic projekta, brez tveganja, da bi spremembe v eni veji povzročile težave v drugi veji. Ko z razvijanjem sprememb končamo lahko vejo enostavno združimo nazaj v glavno vejo. V ločenih vejah lahko hranimo tudi projekt v različnih stanjih - recimo v glavni veji projekt z vsemi spremembami, v drugi veji pa bolj stabilno različico sistema, primerno za testiranje oz. za dostavo do uporabnikov.

Implementacija nove funkcije ali pisanje popravka za projekt se začne z lokalno kopijo trenutnega stanja glavne veje repozitorija, v katerem je shranjen naš projekt. Medtem ko razvijamo naše spremembe na lokalni kopiji je zelo verjetno, da bodo drugi razvijalci uspešno končali razvoj svojih sprememb, ter jih dodali v glavno vejo repozitorija. Tako, dlje kot mi razvijamo naše spremembe, bolj se bo naša lokalna kopija razlikovala od tiste, ki je trenutno aktualna na repozitoriju. To lahko predstavlja velik problem - naše spremembe lahko pokvarijo spremembe drugih in obratno. Za odpravo tega tveganja skrbi neprekinjena integracija - CI, ki je opisna v poglavju 2.4.

S podobnim problemom pa se srečamo tudi ko je potrebno nove funkcije in posodobitve dostaviti do uporabnikov. Dlje časa kot je minilo od zadnje posodobitve, večja je možnost, da pride do napak in težje bo odpravljanje teh, saj bomo morali pregledati velik del kode. Ta problem odpravlja neprekinjena postavitev - CD, ki je opisana v poglavju 2.6.

2.4 CI oz. Continuous Integration

Problemom s prejšnjega podpoglavja se izognemo tako, da na več točkah med razvojem lokalno izvedemo avtomatiziran postopek graditve ter testiranja projekta. Če se projekt pravilno zgradi in se izvedejo vsi testi, je projekt v dobrem stanju. Ko končamo z razvojem naših sprememb, lahko začnemo razmišljati o združitvi naših sprememb v glavno vejo repozitorija. Zelo pomembno je, da pred združitvijo našo lokalno kopijo posodobimo z vsemi spremembami, ki so bile dodane odkar smo kopijo naredili, ter še enkrat izvedemo postopek graditve in testiranja. Tako se prepričamo, da ne prihaja do konfliktov med našimi spremembami in spremembami drugih. Če se pojavijo napake, te popravimo, in ponovno izvedemo graditev in testiranje. Ko se naša lokalna kopija uspešno zgradi in se uspešno izvedejo vsi testi, lahko naše spremembe dodamo v glavno vejo repozitorija. Strežnik nato še enkrat izvede graditev ter testiranje s svojo konfiguracijo, kar lahko prikaže napake, do katerih pride zaradi razlik v konfiguraciji med našim računalnikom in strežnikom ali pa zaradi napake pri posodabljanju repozitorija z našimi lokalnimi spremembami. Napake na vseh korakih razvoja so tako hitro odkrite in jih lahko čimprej odpravimo [38].

V zgornjem odstavku so na primeru iz realnega življenja predstavljeni problemi ter rešitve, ki jih uporaba praks neprekinjene integracije prinaša.

Neprekinjena integracija (angl. Continuous Integration) je torej praksa oz. skupek praks med razvojem projekta, kjer vsi člani razvojne ekipe pogosto - ponavadi večkrat dnevno - združijo svoje delo na eno mesto - danes je to ponavadi repozitorij projekta. Pred in po združitvi se izvede postopek

avtomatskega testiranja, kar poskrbi za minimizacijo konfilktov med delom različnih razvijalcev. Pojem je prvič omenil Gary Boock leta 1991 v svoji knjigi [29], a ni omenjal združevanja kode oz. integracije večkrat dnevno.

2.4.1 Prakse neprekinjene integracije

Gledano s čisto teoretičnega vidika, so prakse Continuous Integration - CI naslednje [38]:

- **Vzdrževanje repozitorija izvorne kode:** projekti za delovanje potrebujejo veliko izvorne kode in ostalih datotek. Sledenje le tem lahko zelo hitro postane naporno in uide izpod nadzora, sploh ko projekt razvija več ljudi. Skozi leta se je uveljavilo več sistemov oz. platform za upravljanje z izvorno kodo - najbolj poznana sistema sta verjetno Git in Mercurial, najbolj poznana platforma pa GitHub. Te platforme omogočajo organizacijo datotek projekta v repozitorije in tako poenostavijo sledenje. V repozitoriju projekta morajo biti na voljo vse datoteke, ki so potrebne za graditev projekta na novem računalniku z minimalno količino drugih potreb - izvorno kodo projekta naj bi z enim ali dvema ukazoma v ukazni vrstici novega računalnika postavili v delujoče stanje. Še ena funkcionalnost, ki jo platforme ponavadi omogočajo, je možnost kreiranja različnih vej v repozitoriju. Tako imamo lahko glavno vejo z aktualno izvorno kodo in recimo vejo za testne funkcije. Mnenje strokovnjakov s tega področja je, da je ta funkcionalnost ponavadi preveč uporabljena in vodi v preveliko število različnih vej, ki jih je potrebno vzdrževati, tako da uporabo minimiziramo [38]. V današnjem svetu razvoja programske opreme ponavadi vsak projekt začne življenje kot prazen projekt na eni izmed platform za upravljanje z izvorno kodo, tako da je ta praksa v uporabi skoraj povsod.

- **Avtomatizacija procesa graditve:** pretvorba izvorne kode v delujoč program ali sistem je ponavadi zapleten proces, ki vključuje več

med seboj nepovezanih korakov - prevajanje kode, polnjenje podatkovnih baz s testnimi podatki, dodajanje drugih datotek itd. Večino oz. vse te korake pa ponavadi lahko avtomatiziramo. S tem minimiziramo možnost za nastanek uporabniških napak pri postavitvi. Glavni cilj je, da lahko celoten sistem zgradimo in postavimo v neko končno, delujoče stanje z vpisom enega ukaza v ukazno vrstico. Obstaja več orodji, ki to omogočajo - izbira je ponavadi odvisna od tipa projekta, na katerem delamo. Še ena pomembna funkcionalnost teh orodji je analiza sprememb ob vsaki posodobitvi. Proces graditve lahko traja kar nekaj časa, zato če naredimo le nekaj manjših sprememb v kodi, ponavadi ni potrebno izvajanje celotnega postopka graditve. Ta orodja analizirajo spremembe in poskušajo minimizirati čas, ki je potreben za graditev posodobljene različice sistema.

- **Zgrajeni sistem naj se samotestira:** v primeru, da je graditev sistem uspešna, potem naj se samodejno zaženejo še avtomatski testi za ta sistem. Za to moramo seveda imeti napisane avtomatske teste, ki preverijo večji del programske kode, če se ta obnaša pravilno. Pogoj, da je sistem samotestiran je, da v primeru nedelujočih testov to povzroči neuspešno graditev samega sistema. Seveda se na te teste ne smemo zanašati, da nam pokažejo vse mogoče napake in hrošče v naši kodi.

- **Vsi člani ekipe naložijo svoje spremembe v glavno vejo repozitorija vsaj enkrat dnevno:** z rednim nalaganjem posodobitev na repozitorij lahko zelo hitro ugotovimo, če prihaja do konfliktov med spremembami različnih članov ekipe, kar nam omogoča tudi hitro reševanje le teh. Če se spremembe nalagajo vsakih nekaj ur, in v tem obdobju pride do težav, moramo pregledati relativno majhen del kode, da najdemo napako. Edini pogoj, da lahko razvijalec naloži svoje posodobitve na repozitorij je, da se njegova različica uspešno zgradi in testira. Nato mora lokalno kopijo posodobiti s spremembami, ki so bile dodane v glavni repozitorij od izdelave njegove lokalne kopije in postopek gra-

ditve ter testiranja še enkrat ponoviti. Ta praksa pomaga razvijalcem predvsem z razdelitvijo dela na manjše, bolj obvladljive segmente, kar s seboj prinese tudi lažje sledenje napredkom na projektu.

- **Vsaka posodobitev naj sproži graditev glavne veje na integracijski napravi:** tudi z rednim, lokalnim testiranjem posodobljenih različic sistema še vedno lahko pride do tega, da se različica s težavami pojavi na glavni veji repozitorija. Razlogov za to je lahko več. Gre lahko seveda za napako s strani enega od razvijalcev, ki naloži kodo, ki se ne zgradi ali testira uspešno. Lahko pa se pojavijo tudi razlike v okoljih med računalniki razvijalcev, kar lahko napake prikrije. Zato moramo redno preverjati, če posodobljena različica sistema povzroča težave tudi na integracijski napravi - naprava, ki je konfigurirana podobno kot produkcijsko okolje. V primeru, da spremembe razvijalca povzročijo napake, mora biti ta posodobitev kode označena kot neveljavna. Spremembe so ponavadi razveljavljene, razvijalec pa je zadolžen za odpravo napak. Za takšno testiranje lahko skrbimo na dva glavna načina. Prvi način je ročna graditev projekta na integracijski napravi. Razvijalec sam na integracijski napravi zgradi najnovejšo različico sistema, ki vključuje tudi njegove spremembe, in spremlja napredek. Če je z novo različico vse v redu, so njegove posodobitve veljavne. Drugi način pa je uporaba strežnika za neprekinjeno integracijo. Ta spremlja stanje repozitorija kode in ob vsaki spremembi samodejno naloži in zgradi kodo na integracijski napravi, ter obvesti člana, ki je sprožil postopek, o končnem stanju graditve in testiranja.

- **Graditev sistema naj ostaja hitra:** z dodajanjem novih funkcij in povečevanjem obsega programske kode se lahko postopek graditve sistema občutno podaljša. Ena izmed glavnih prednosti neprekinjene integracije je hitro pridobivanje povratnih informacij. V primeru, da je postopek graditve dolg, potem to prednost izgubimo. Dober cilj je, da naj bi graditev sistema trajala okoli 10 minut. Vsaka minuta pa je

tukaj pomembna, saj vsaka posodobitev glavne veje repozitorija sproži tudi postopek graditve in testiranja. Razvijalec mora počakati na rezultate tega postopka, kar pomeni da lahko tukaj prihranimo veliko časa, če je ta postopek hiter. Veliko izboljšavo lahko prinese uvedba t.i. postavitvenega cevovoda. Postavitveni cevovod zaporedno sproži več postopkov graditve sistema. Korake v cevovodu konfiguriramo sami - prva graditev ponavadi izpusti večino dolgotrajnih korakov in preveri le, če je trenutna koda dovolj dobra oz. stabilna da lahko na njej delajo drugi razvijalci. Naslednji korak oz. koraki pa izvedejo celoten postopek graditve in testiranja ter pokažejo napake, ki jih prejšnji koraki niso zaznali. Cevovod lahko v kasnejših korakih tudi paraleliziramo.

- **Testiranje v klonu produkcijskega okolja:** glavni razlog za testiranje sistema je, da preprečimo, da bi se napake pojavile v produkciji oz. da bi napake dosegle končnega uporabnika. Zelo pomemben del tega je okolje, v katerem bo končni sistem deloval. Če postopek testiranja izvajamo v drugačnem okolju, prihaja do tveganja, da rezultati ne bodo enaki kot v končnem, produkcijskem okolju. Zato je pomembno, da naše testno okolje replicira produkcijsko okolje kar se da natančno. Poudarek je tukaj na podatkovni bazi (če je ta prisotna), operacijskem sistemu, strojni opremi ipd. Pojavijo se seveda omejitve - lahko gre za finančne omejitve pri kloniranju ali pa za preveliko število različnih produkcijskih okolij (predvsem pri razvoju namiznih aplikacij), da bi preverili delovanje na vseh ipd. Na tem področju nam lahko pomaga virtualizacija, ki v večini primerov olajša minimizacijo razlik.

- **Pridobivanje najnovejše različice sistema naj bo enostavno:** naročniki oz. uporabniki sistema s težavo pravilno napovejo in izrazijo zahteve sistema v postopku načrtovanja. Skozi razvoj sistema se ponavadi pojavijo dodatne zahteve in pripombe glede implementacije določenih funkciji. Te informacije želimo pridobiti kakor hitro je to mogoče, da lahko pripombe upoštevamo ter da lahko postopek

razvoja prilagodimo novim zahtevam. To nam prihrani čas in minimizira količino kode, ki jo moramo prilagajati. Pomembno je, da naši naročniki oz. skupina preizkuševalcev vedno ve, kje lahko pridobi najnovejšo stabilno različico sistema, ki je zgrajena in je bila uspešno testirana - z drugimi besedami, primerna za končno uporabo. Ponudimo lahko tudi več različic sistema - predvsem pri demonstracijah je bolj pomembno, da oseba pozna funkcije sistema in zna z njim upravljati, kot pa da sistem vključuje najnovejše funkcije in podpira vse nefunkcionalne zahteve.

- **Rezultat zadnjega poskusa graditve vidijo vsi člani ekipe:** komunikacija je ena izmed najpomembnejših stvari pri neprekinjeni integraciji. Zagotoviti moramo, da lahko vsak preveri stanje trenutne različice sistema oz. aplikacije. Ena izmed najbolj pomembnih informacij je trenutno stanje glavne veje - če pride do napak pri graditvi oz. testiranju najnovejše različice mora informacija čim hitreje doseči razvijalce, da začnejo z razvijanjem popravkov. Večina sistemov oz. platform za CI ponuja vmesnike za enostaven vpogled v stanje sistema - če trenutno poteka nov postopek graditve in testiranja, stanje zadnje graditve in testiranja itd. Ponavadi ponujajo tudi samodejno obveščanje razvijalcev v primeru neuspešne graditve ali testov - obveščanje lahko poteka neposredno preko grafičnega vmesnika platforme, e-pošte ali pa preko integracij s platformami, kot so npr. Slack.

- **Avtomatizacija postavitve:** pri neprekinjeni integraciji med postopkom razvoja sistem premikamo in postavljamo v več različnih okoljih, sploh če postopek testiranja razdelimo v več korakov - osnovni testi se izvedejo v enem okolju, ostali testi pa v enem ali več drugih okoljih. Ker med temi okolji redno, lahko tudi večkrat dnevno, premikamo naš sistem v različnih stanjih je smiselno postopek postavitve v teh okoljih avtomatizirati. Lahko pa gremo še korak dlje in avtomatiziramo tudi prehod iz razvojnega okolja v produkcijsko okolje oz. dostavo novih

različic sistema do uporabnikov. Z avtomatizacijo tega postopka pa začnemo govoriti o neprekinjeni postavitvi (angl. Continuous Deployment - CD), ki bo opisana v podpoglavju 2.6 CD oz. Continuous Deployment.

2.4.2 Prednosti neprekinjene integracije

Neprekinjena integracija omogoča podjetjem doseganje krajših in bolj pogostih obdobji lansiranja, boljšo kvaliteto programske opreme in boljšo produktivnost njihovih razvojnih ekip [56, 61]. Napake so hitro odkrite in zato je potrebno pregledati občutno manjše dele kode, da napako najdemo in popravimo [38]. Prakse neprekinjene integracije so splošno sprejete med najbolj popularnimi odprtokodnimi projekti, število projektov, ki prakse uporablja pa se povečuje [41].

2.5 CD/CDE oz. Continuous Delivery

Uporabniki danes pričakujejo redne posodobitve aplikacij, ki jih uporabljajo v vsakodnevnem življenju. Lansiranje nove različice aplikacije je za razvojno ekipo težko pričakovan dogodek, ki s seboj nosi kar nekaj tveganja in potrebuje veliko priprav - kljub testiranju se še vedno lahko pojavijo napake, o novi različici je potrebno obvestiti uporabnike itd. Podobno kot pri težavi med razvojem - dlje kot razvijamo spremembe na lokalni kopiji repozitorija, manj bo ta podobna aktualni različici - ki jo poskuša rešiti neprekinjena integracija, tudi pri novih različicah velja, da več časa kot je minilo od lansiranja zadnje različice, težje bo odpravljanje potencialnih težav. Pogosto lansiranje in avtomatizacija tega postopka rešujeta težavo velikih sprememb med različicami, kjer lahko iskanje napak traja dolgo časa in odpravljata strah, ki se pojavi ob lansiranju nove različice.

Continuous delivery oz. neprekinjena dostava, ki je včasih namesto s CD označena tudi s kratico CDE [56], je praksa med razvojem sistema, kjer razvoj poteka v kratkih, kontroliranih korakih. Vse spremembe izvorne kode

avtomatsko sprožijo postopek graditve in testiranja, ter so nato tudi avtomatsko postavljene v testnem produkcijskem okolju. Če neprekinjeno dostavo pravilno implementiramo potem zagotovimo, da je sistem vedno v stanju, ki je uspešno prestal celotni postopek testiranja in je pripravljen za lansiranje v produkcijsko okolje [1]. Pomembno je poudariti, da se postopek avtomatskega lansiranja nove različice v končno produkcijsko okolje ne sproži samodejno - to moramo storiti ročno, ko smo na to pripravljeni oz. na željo naročnika. Glavni cilj neprekinjene dostave je sistem, ki je vedno pripravljen na ta korak. Gre za razširitev praks, ki jih uvaja neprekinjena integracija, tako da je uporaba neprekinjene integracije pogoj za uporabo neprekinjene dostave [56].

2.5.1 Prednosti neprekinjene dostave

Ena izmed glavnih prednosti, ki jih ponuja uporaba neprekinjene dostave je izboljšava komunikacije in sodelovanja z uporabniki [44], ostale prednosti pa so nižja tveganja ob lansiranju novih različic, hitrejše dostavljanje posodobitev, višja kvaliteta sistema in večje zadovoljstvo ter produktivnost razvojnih ekip [44, 31, 43, 1].

2.6 CD oz. Continuous Deployment

Lahko pa prakso neprekinjene dostave popeljemo še korak dlje.

Continuous deployment oz. neprekinjena postavitve je praksa, ki avtomatsko in neprekinjeno postavlja sistem v produkcijskem okolju. Neprekinjena postavitev ne vključuje nobenga koraka, ki ne bi bil avtomatiziran - takoj ko razvijalec posodobi vejo repozitorija, ki je namenjena produkciji, te spremembe vstopijo v t.i. postavitveni cevovod in dosežejo produkcijsko okolje. Edini razlog za prekinitev postopka postavitve je nedelujoč test oz. kakšna druga napaka, ki povzroči neuspešno graditev sistema. Pomembno se je zavedati, da je uporaba neprekinjene postavitve pogojena z uporabo neprekinjene dostave, vendar obratno ne velja - neprekinjena postavitev je

nadaljevanje neprekinjene dostave [56]. Čeprav je uporaba neprekinjene dostave primerna za vse vrste projektov, to ne velja vedno tudi za prakso neprekinjene postavitve. Možnost uporabe neprekinjene postavitve je zelo odvisna od tipa sistema ter podjetja za njim [56].

2.6.1 Prednosti neprekinjene postavitve

Ena izmed glavnih prednosti uporabe neprekinjene postavitve je, da se lahko odzivamo na povratne informacije uporabnikov v skoraj realnem času. Če uporabniki opazijo nepravilno delovanje sistema in o tem obvestijo razvojno ekipo, lahko takoj začnemo z razvijanjem popravkov. Ko težave odpravimo in nova koda uspešno preide skozi vse postopke testiranja popravki takoj dosežejo uporabnike. Podoben postopek velja za implementacijo novih funkcij [4]. Uporaba praks neprekinjene integracije nam prav tako omogoča drastična večanja razvojnih ekip in obsega kode brez vpliva na produktivnost ali kvaliteto kode [55]. Ostale prednosti vključujejo boljšo kvaliteto sistema, boljšo produktivnost in možnost izboljšave zadovoljstva uporabnikov z bolj pogostimi posodobitvami [47].

Poglavje 3

Platforme

V zadnjih letih se je pojavilo veliko različnih platform, ki omogočajo funkcionalnosti CI/CD. Ugotovil sem, da lahko platforme načeloma razdelimo v tri glavne kategorije:

- **Platforme za upravljanje z izvorno kodo:** glavni namen teh platform je ponujanje storitve upravljanja z izvorno kodo - hranjenje izvorne kode v repozitorijih, omogočanje vejitev itd. Dodatne funkcije na teh platformah pa lahko omogočajo tudi uporabo CI/CD funkcionalnosti. Primer takšne platforme je GitHub, ki od leta 2019 ponuja funkcionalnosti CI/CD z GitHub Actions.
- **Namenske platforme za projektno delo:** te platforme so zelo podobne platformam za upravljanje z izvorno kodo - ponavadi to tudi v osnovi so. Razlika je, da so te platforme bolj osredotočene na ostale funkcije, ki so potrebne za razvoj in vzdrževanje sistema ter za upravljanjem z večjimi razvojnimi ekipami. Med te funkcije spada seveda tudi CI/CD. Primera takšnih platform sta Bitbucket in GitLab.
- **Namenske platforme za CI/CD:** te platforme same ne ponujajo sistema za upravljanje z izvorno kodo in skrbijo izključno samo za neprekinjeno integracijo - CI ter nekatere tudi za neprekinjeno postavitev - CD. Zanašajo se na zunanjo platformo za hranjenje izvorne kode, ob

posodobitvah pa od repozitorija dobijo signal, se na njega povežejo in pričnejo z izvajanjem funkcionalnosti CI/CD. Nekatere tudi ne ponujajo lastnih strežnikov - tako moramo za postavitev in strežnik poskrbeti sami. Primer takšne platforme je TravisCI in DroneCI - slednji ne ponuja lastnih strežnikov.

Za primerjavo platform, ki omogočajo funkcionalnosti CI/CD, sem si v okviru diplomskega dela izbral 3 platforme, na katere sem naložil aplikacijo in nato postavil cevovod CI/CD. V primerjavi sem se osredotočil na platformo **GitHub** in na njeno izvedbo postopka CI/CD. Postopek sem primerjal še na platformi **Bitbucket**, ki je bolj osredotočena na projektno vodenje. Postopek pa sem primerjal še na platformi **GitLab**. V nadaljevanju so vse platforme natančneje predstavljene.

3.1 GitHub

GitHub [8] je platforma za razvoj programske opreme, ki razvijalcem omogoča hranjenje programske kode v repozitorijih z uporabo sistema Git [28, 57]. Prva koda se je na platformi pojavila oktobra 2007, avgusta 2019 pa je imela platforma že več kot 50 milijonov uporabnikov ter več kot 100 milijonov različnih repozitorijev. Platformo uporablja kar nekaj znanih podjetji oz. projketov npr.: Google, Spotify, Facebook, Nasa ter Node.js. Leta 2018 je platforma postala last podjetja Microsoft [16]. V zadnjih nekaj letih se platforma vse bolj spreminja v platformo za upravljanje s projekti, vendar platforma GitHub se še vedno osredotoča predvsem na odprtokodne projekte in v svetu razvoja programske opreme še vedno ostaja prva izbira za hranjenje novih, predvsem odprtokodnih, projektov. GitHub ponuja skoraj vse svoje funkcije brez doplačila za samostojne razvijalce in ekipe, ki kodo hranijo v javnih repozitorijih. Število javnih/privatnih repozitorijev je neomejeno in neomejeno je tudi število ekipnih članov - od aprila 2020 tudi za privatne repozitorije. Slika 3.1 prikazuje logotip platforme GitHub.

S funkcijo GitHub Actions, ki CI/CD podpira od novembra 2019 [9],

Slika 3.1: Logotip platforme GitHub, © 2020 GitHub, Inc. [10]

platforma omogoča izdelavo in izvajanje delovnih tokov in tako omogoča vse funkcionalnosti CI/CD. Funkcija je na voljo tako javnim kot zasebnim repozitorijem, s to razliko, da imajo javni repozitoriji neomejeno število minut izvajanja. K izvajanju se šteje izvajanje akcij, ki jih definiramo v repozitoriju - ko se akcije izvajajo porabljajo zakupljene minute izvajanja. Če izvajamo akcije na operacijskem sistemu Linux se minute porabljajo v razmerju 1:1, za operacijski sistem Windows se vsaka minuta izvajanja šteje kot dve minuti, za operacijski sistem macOS pa se eno minuta izvajanja šteje kot 10 minut izvajanja.

3.1.1 Monetizacija platforme

GitHub uporabnikom ponuja 4 različne stopnje plačljivih funkcij v razponu od 0$ do 21$ na uporabnika na mesec s paketom Enterprise. Če razvijamo projekt v javnem repozitoriju imamo na voljo neomejeno število minut izvajanja akcij v sklopu GitHub Actions, neomejeno prostora za shranjevanje programskih paketov in večino ostalih funkcionalnosti, ki jih platforma ponuja. Če želimo imeti privatni repozitorij pa se število funkcij zmanjša - na voljo imamo 2000 minut izvajanja akcij na mesec in 500MB za shranjevanje programskih paketov. Pri podpori pa nam je ne glede na tip repozitorija na voljo le podpora skupnosti. Če želimo dokupiti dodatne minute lahko to storimo po tarifi 0.008$ na minuto, prostor za shranjevanje pa lahko povečamo po tarifi 5$ za 50GB prostora. Dražji paketi povečajo količino minut za izvajanje akcij, ponujajo več prostora za shranjevanje paketov, omogočajo uporabo vseh funkcij v privatnih repozitorijih ter nudijo boljšo, hitrejšo, podporo. Pa-

ket Enterprise pa omogoča tudi integracijo s storitvama GitHub Enterprise Cloud in Server in še nekatere funkcije specializirane za podjetja, GitHub One pa doda še prednostno 24/7 podporo, neprekinjeno učenje in še nekaj drugih, zelo specializiranih funkcij.

3.2 Bitbucket

Platforma Bitbucket [5] na prvi pogled ponuja podobne oz. enake funkcije kot GitHub. Glavna funkcija platforme je enaka kot pri platformi GitHub - ponujanje repozitorijev za hranjenje in upravljanje z izvorno kodo. Vendar BitBucket svoje storitve že od samega začetka trži profesionalnim razvijalcem in večjim razvojnim ekipam. Platforma je pričela delovati leta 2008, septembra leta 2010 pa je platformo kupilo podjetje Atlassian [2]. Trenutno platformo uporablja več kot 1 milijon ekip in več kot 10 milijonov uporabnikov. Med najbolj znane uporabnike platforme sodijo Ford, Pandora in WeWork. Platforma je načeloma plačljiva, vendar majhnim projektom, na katerih sodeluje 5 ali manj ljudi, omogoča uporabo brez doplačila. Slika 3.2 prikazuje logotip platforme Bitbucket.

Slika 3.2: Logotip platforme BitBucket, © 2020 Atlassian [3]

Glavni namen platforme je upravljanje z večjimi projekti, tako da platforma vključuje kar nekaj funkcij in orodji posebno namenjenih temu - recimo enostavno integracijo z zelo popularnima orodjema za projektno delo Jira in Trello. Ta orodja sta del zelo širokega ekosistema, ki ga ponuja podjetje Atlassian. Ta ekosistem je še ena pomembna lastnost platforme Bitbucket saj omogoča integracijo z večino teh orodji oz. storitev. Za omogočanje CI/CD funkcionalnosti skrbi funkcija Bitbucket Pipelines.

Za vključitev platforme v primerjavo sem se odločil predvsem zaradi njene popularnosti, sploh v profesionalnih krogih razvoja programske opreme.

Platforma je pogosto omenjena v člankih in blogih podjetij, tudi sam pa sem se s platformo že večkrat srečal pri počitniškem delu.

3.2.1 Monetizacija platforme

Bitbucket ponuja 3 različne pakete funkcij v razponu od 0$ do 6$ na uporabnika na mesec. Platforma omogoča uporabo brez doplačila, vendar ta paket je samo za majhne ekipe - ustvarimo lahko neomejeno javnih/zasebnih repozitorijev, ampak na vsakem lahko skupaj dela do 5 članov. Za funkcijo Bitbucket Pipelines imamo na voljo le 50 minut izvajalnega časa, v funkciji Git Large File Storage pa lahko hranimo do 1GB podatkov. Brez doplačila imamo še vedno dostop do integracij s storitvama Jira in Trello in dostop do enake podpore, kot uporabniki plačljivih paketov. Dodatni čas in prostor lahko ekipe zakupijo dinamično, če ga potrebujejo - cena je 10$ za 1000 minut izvajalnega časa in 10$ za 100GB prostora za Git Large File Storage. Plačljivi paketi omogočajo dodajanje neomejenega števila članov na vse repozitorije, povišajo količino izvajalnega časa na 2500 minut za 3$ in 3500 minut za 6$ ter povečajo prostor za Git Large File Storage na 5GB za 3$ in 10GB za 6$. Najdražji paket Premium pa ponuja ekipam še nekaj dodatnih funkcij, ki izboljšajo nadzor nad projekti. Bitbucket pa ponuja tudi možnost postavitve platforme na lastnih strežnikih in podatkovnih središčih. Cena je odvisna od števila uporabnikov. Licenca za lasten strežnik je enkratni nakup, za podatkovna središča pa je licenca letna.

3.3 GitLab

Platforma GitLab [11] je zelo podobna platformi Bitbucket - osrednja funkcija je sistem za upravljanje z repozitoriji izvorne kode, platforma pa se osredotoča na podporo celotnega življenjskega cikla DevOps [33], torej podpira vse korake od procesa načrtovanja projekta, do lansiranja na trg in vzdrževanja ter zagotavljanja varnosti. Za razliko od platforme Bitbucket, GitLab zastonjskim uporabnikom omogoča izdelavo projektov, na katerih lahko sodeluje

neomejeno število ljudi. Platforma ima trenutno več kot 40 miljonov uporabnikov in beleži zelo hitro rast v popularnosti [12]. GitLab uporabljajo podjetja, kot so Goldman Sachs, Bayer, Sony, Lockheed Martin, Wish in druga.

Eden izmed glavnih razlogov zakaj sem platformo vključil v primerjavo je, da veliko uporabnikov oz. ekip svoje projekte seli na GitLab prav iz platforme Bitbucket. Večina jih kot glavne razloge za selitev navaja predvsem integrirano funkcionalnost neprekinjene integracije ter njeno izvedbo, navajajo pa tudi možnost ustvarjanja skupin in podskupin uporabnikov ter boljšo podporo in funkcije za lastne, po meri konfigurirane "Runner"-je [19]. Oddelek za spletni razvoj v podjetju, kjer sem v sklopu fakultete opravljal prakso, je nekaj časa nazaj tudi preselil vse svoje projekte iz platforme Bitbucket na GitLab. Platforma je tudi glavna konkurenca platformi Bitbucket. Slika 3.3 prikazuje logotip platforme GitLab.

Slika 3.3: Logotip platforme BitBucket, © 2020 GitLab [14]

Tako kot Bitbucket lahko tudi GitLab sami gostimo na svojih strežnikih. Ena izmed večjih posebnosti platforme je, da je velik del (Community Edition) programske kode celotne platforme, ki jo lahko namestimo na lastnih stežnikih, odprtokoden, podjetje pa posluje po načelu odprtega jedra [13], kar pomeni da delijo več informacij z javnostjo kot druga podjetja.

3.3.1 Monetizacija platforme

Platforma GitLab ponuja 4 različne plane za ekipe s cenami od 0$ do 99$ na uporabnika na mesec. Zastonjskim uporabnikom platforma ponuja neomejeno število repozitorijev z neomejenim številom članov. Platforma omogoča

podporo vseh stopenj cikla DevOps, možnost uporabe uporabnikom lastnih "Runner"-jev - naprav za poganjanje cevovodov CI/CD in 2000 minut izvajanja CI/CD na napravah, za katere skrbi GitLab itd. V dražjih planih pridobimo dostop do več funkcij, ki nam omogočajo boljši nadzor nad projektom, boljšo, hitrejšo podporo in v najdražjih paketih celo izvajanje CI/CD za repozitorije na drugih platformah. GitLab ponuja tudi možnost postavitve platforme na lastnem strežniku, če to želimo - večina kode, ki jo platforma uporablja je javno dostopna. Najdražji plan ponuja še napredne varnostne nastavitve za aplikacije, vpoglede za vodstveno raven, 50000 minut izvajanja cevovodov CI/CD itd. Ta paket je namenjen izključno največjim podjetjem, ki želijo svoj celoten razvojni del prestaviti na platformo.

Poglavje 4

Primerjava platform

4.1 Primerjava funkcij

Če primerjamo glavne funkcije, ki jih platforme ponujajo, hitro opazimo, da večjih razlik praktično ni. Eden izmed glavnih razlogov za to je, da so izbrane platforme ena drugi največja konkurenca - predvsem platformi Bitbucket in GitLab. Zaradi tega se platforme trudijo ostati konkurenčne ena drugi z repliciranjem funkcij, ki jih konkurenca ponuja.

Platforma GitHub ponuja praktično vse svoje funkcije uporabnikom brez dodatnega plačila. Edina omejitev je, da so funkcije omogočene le v javnih oz. odprtokodnih repozitorijih. GitHub s tem pospešuje rast odprtokodnih projektov - podpora odprtokodnih projektov oz. koncepta odprtokodnosti je eno izmed glavnih načel platforme. Če se odločimo za plačljiv paket Team, nam je na voljo uradna podpora in omogočena je uporaba vseh funkciji, tudi v privatnih repozitorijih. Uradna podpora omogoča neposredno komunikacijo s podjetjem preko e-pošte, kar je hitrejše in ponuja bolj zanesljive rešitve kot forumi in ostali skupnostni kanali, na katere smo omejeni brez tega paketa. Paket Enterprise pa omogoči še dostop do funkcij, namenjenih podjetjem. Vendar, ker platforma načeloma ni namenjena projektnemu vodenju, ponuja malo specializiranih funkcij oz. orodji za delo na projektih in upravljanje z njimi. Za funkcije projektnega vodenja in upravljanja s projekti se platforma

zanaša na ekosistem uradnih in neuradnih integracij z zunanjimi orodji kot so Jira in Trello. V tabeli 4.1 so bolj pregledno predstavljeni paketi platforme GitHub in njihove glavne prednosti.

GitHub		
Paket	Cena (mesečno)	Glavne prednosti
Free	0$	Neomejeno javnih/zasebnih repozitorijev, neomejeno članov v repozitorijih, večina funkcij na voljo javnih repozitorijih, uporabniški "Runner"-ji
Team	4$/uporabnika	Večina funkcij v privatnih repozitorijih, uradna podpora
Enterprise	21$/uporabnika	Integracija z GitHub Server in Cloud, napredne varnostne funkcije, napredna podpora, 50000 minut izvajanja CI/CD
One	Posamezno	Podpora s prednostjo

Tabela 4.1: Tabela prikazuje pakete, ki jih ponuja platforma GitHub

Opazimo lahko, da je platforma Bitbucket najbolj omejujoča pri funkcijah, ki jih ponuja uporabnikom brez doplačila. Usmerjena je predvsem v ponujanje bolj specializiranih funkcij večjim ekipam in podjetjem, tako da je plan brez plačila namenjen predvsem manjšim, samostojnim ekipam, ki lahko platformo, tudi z omejitvami, uporabljajo brez večjih težav. Platforma za potrebe projektnega dela ponuja tudi enostavno integracijo s priljubljenima orodjema Trello in Jira. Orodji Trello in Jira sta del večjega ekosistema storitev, ki jih ponuja podjetje Atlassian. V okviru ekosistema si podjetja lahko poljubno izberejo paket storitev, ki jim najbolj ustreza. S paketom Standard za 3$ na uporabnika na mesec platforma postane po funkcijah zelo konkurenčna platformi GitHub z vključenim paketom Team, ki je za 1$ dražji. Vendar, če pri platformi GitHub želimo enak nabor naprednih funkcij, kot jih omogoča Bitbucket s paketom Premium za 6$ na uporabnika na mesec, se moramo odločiti za občutno dražji paket Enterprise. V tem primeru postane

Bitbucket občutno cenejša in načeloma boljša opcija za večja podjetja. V tabeli 4.2 so bolj pregledno predstavljeni paketi platforme Bitbucket in njihove glavne prednosti.

Bitbucket		
Paket	Cena (mesečno)	Glavne prednosti
Free	0$	Neomejeno javnih/zasebnih repozitorijev (do 5 članov), uradna podpora
Standard	3$/uporabnika	Neomejeno članov v repozitorijih, 2500 minut izvajanja CI/CD
Premium	6$/uporabnika	Napredne varnostne funkcije

Tabela 4.2: Tabela prikazuje pakete, ki jih ponuja platforma Bitbucket

Platforma GitLab pa poskuša biti dobra alternativa tako platformi GitHub kot tudi Bitbucket. Zastonjskim uporabnikom ponuja primerljive funkcije kot platforma GitHub - ena izmed prednosti pa je neomejeno število članov na javnih in privatnih repozitorijih. S paketom Bronze uspešno konkurira GitHub-ovem paketu Team. Večina naprednih funkciji za vodenje in nadzor nad projekti je vključenih v dražji paket Silver, ki je dobra konkurenca platformi Bitbucket. Paket Silver omogoča tudi povezavo z repozitoriji na obeh izmed platform GitHub in Bitbucket ter izvajanje CI/CD nad njimi preko platforme GitLab. Paket Silver ponuja tudi integracijo z orodjem Jira. Paket Gold pa je osredotočen na večja podjetja, ki želijo celoten proces razvijanja in projektnega vodenja opravljati preko enotne platforme. GitLab se na vseh nivojih osredotoča na ponujanje lastnih orodji za vse potrebe - na nižjih nivojih za razvoj, na višjih pa za projektno vodenje, podjetne funkcije in upravljanje z ekipami. Tako se GitLab izogne odvisnosti od zunanjih platform kot je npr. TravisCI in orodji kot so npr. Trello, Jira in Slack. V tabeli 4.3 so bolj pregledno predstavljeni paketi platforme GitLab in njihove glavne prednosti.

GitLab		
Paket	Cena (mesečno)	Glavne prednosti
Free	0$	Neomejeno javnih/zasebnih repozitorijev, podpora vseh stopenj cikla DevOps, uporabniški "Runner"-ji
Bronze	4$/uporabnika	Uradna podpora, osnovne skupinske funkcije
Silver	19$/uporabnika	Napredne skupinske funkcije, napredne varnostne funkcije, napredna podpora, CI/CD za zunanje repozitorije
Gold	99$/uporabnika	Podpora s prednostjo, profesionalne varnostne funkcije in analize

Tabela 4.3: Tabela prikazuje pakete, ki jih ponuja platforma GitLab

4.2 Postopek postavitve cevovoda CI/CD

4.2.1 Testna aplikacija

Za postopek postavitve cevovoda CI/CD sem potreboval enostavno aplikacijo, ki bi vključevala komponente, za katere bi lahko spisal enotske teste. Odločil sem se za enostaven strežnik v okolju Node.js [21]. Vsebuje dve datoteki JavaScript [7], v katerih se nahajata razreda Vector4f in Matrix4f, ki omogočata ustvarjanje in nekaj enostavnih operacij nad vektorji in matrikami. Datoteki skupaj vključujeta 15 različnih funkciji, za katere sem lahko izdelal teste. Sprva sem načrtoval, da bi aplikacija funkcije obeh razredov ponujala kar preko obrazcev na spletni strani, nato pa sem se zaradi možnosti izdelave dodatnih testov odločil, da bom te funkcionalnosti ponujal preko vmesnika (angl. API - Application Programming Interface) in da bodo strani vključevale le navodila za uporabo.

Na vseh treh platformah sem začel s praznim projektom, v katerem sem inicializiral aplikacijo Node.js in namestil naslednje pakete:

- **express.js** [50] - minimalno in fleksibilno ogrodje za gradnjo spletnih aplikacij in vmesnikov,
- **morgan** [26] - vmesno programje/sistem za beleženje zahtev HTTP, ki pridejo na naš strežnik,
- **nodemon** [23] - pripomoček, ki spremlja stanje izvorne kode našega projekta in ob spremembah samodejno ponovno zažene strežnik
- **Mocha** [24] - fleksibilno testno ogrodje/knjižnica za jezik JavaScript,
- **Chai** [15] - knjižnica za preverjanje testnih trditev, ki jo lahko združimo z vsemi JavaScript testnimi ogrodji,
- **SuperTest** [18] - modul za testiranje pravilnosti zahtev HTTP.
- **nyc** [17] - orodje za enostavno preverjanje pokritosti kode s testi.

V projekt sem dodal datoteko JS z razredom Vector4f in za vse funkcije v tem razredu napisal enotske teste. Nato sem dodal vstopno stran ter stran z navodili za uporabo klicev vmesnika povezanih z vektorji. Za lažje oblikovanje strani sem uporabil rešitev BootstrapCSS [20] in Bootstrap predlogo Album [22]. Na tej točki sem na platformah dodal CI del cevovoda.

Nato sem na strežnik dodal poti ter spisal funkcije za del vmesnika, ki se navezuje na vektorje. Za novo funkcionalnost vmesnika sem dodal še teste, spisal pa sem tudi teste za streženje začetne strani in strani z navodili. Dodal sem še teste za obravnavanje napak na strežniku. Na tej točki sem na platformah dodal še CD del cevovoda.

Isti postopek sem nato ponovil še za datoteko z razredom Matrix4f - datoteko sem dodal, za njo spisal teste, dodal stran za pomoč, dodal funkcionalnost vmesnika in dodal še teste za vmesnik.

Končna struktura testov je bila takšna:

- Matrix class tests - testi za razred Matrix4f (12 testov),
- Vector class tests - testi za razred Vector4f (19 testov),

- Server error handling tests - testi za strežniško obravnavanje napak (1 test),
- Server function tests - testi za streženje strani (3 testi),
- Matrix API function tests - testi funkcij vmesnika za matrike (13 testov),
- Vector API function tests - testi funkcij vmesnika za vektorje (27 testov).

Skupno se torej izvede 75 testov. Orodje nyc sporoči, da je s testi pokritih 95.18% vseh poti v aplikaciji. Na sliki 4.1 je prikazano celotno tekstovno poročilo orodja nyc.

File	% Stmts	% Branch	% Funcs	% Lines
All files	99.61	95.18	100	99.59
GitHub	96.43	83.33	100	96.43
app.js	94.74	75	100	94.74
middlewares.js	100	100	100	100
GitHub/js	100	100	100	100
Matrix4f.js	100	100	100	100
Vector4f.js	100	100	100	100
GitHub/routes	100	94.47	100	100
index.js	100	100	100	100
matrixApi.js	100	100	100	100
vectorApi.js	100	87.93	100	100

Slika 4.1: Tekstovno poročilo orodja nyc

Za postavitev aplikacije sem se odločil za platformo Heroku, ker ponuja zastonjsko gostovanje za manjše, nekomercialne, osebne projekte in projekte, ki so ustvarjeni kot "proof of concept". Aplikacije zastonjskih uporabnikov se po 30 minutah neaktivnosti samodejno ugasnejo. Vsaka aplikacija se kreira s praznim repozitorijem Git, kar je eden izmed načinov za nalaganje kode na platformo. Projekt lahko posodabljamo še z uradnim vmesnikom Heroku v ukazni vrstici, s platformo GitHub pa obstaja tudi neposredna integracija, kjer lahko z nekaj kliki povežemo platformo Heroku na naš repozitorij in Heroku sam posodablja kodo - nastavimo lahko tudi da počaka izvajanje CI.

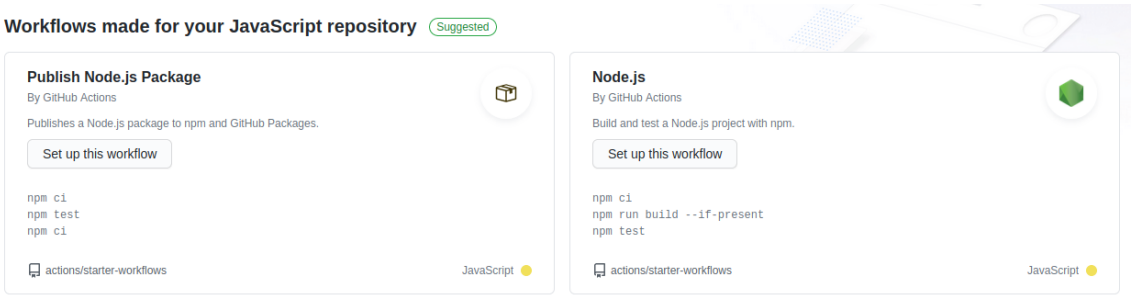
Za ta način se nisem odločil, saj ta integracija obstaja samo za platformo GitHub in ne tudi za drugi dve platformi v moji primerjavi.

4.2.2 GitHub

Platforma GitHub omogoča funkcionalnosti CI/CD z uporabo funkcije GitHub Actions. GitHub Actions omogoča uporabnikom dodajanje delovnih tokov v repozitorije v obliki datotek YAML - YAML [25] je standard oz. jezik za serializacijo podatkov, ki se pogosto uporablja za pisanje konfiguracijskih datotek. Datoteke so shranjene v korenski mapi `.github`, podmapa `workflows`. Sistem je dokaj robusten in omogoča izvajanje akcij za skoraj vse programske jezike, izvajanje na različnih verzijah različnih operacijskih sistemov, uporabo Docker vsebnikov [60], izdelavo obvestil in nalog na platformah kot so Jira, Slack, Microsoft Teams itd. Platforma nam omogoča tudi postavitev lastnih, osebnih naprav za izvajanje akcij. Trenutno je na voljo okrog 40 uradnih predlog za izdelavo novih delovnih tokov ter več kot 4700 uradnih in neuradnih akcij - vnaprej pripravljene akcije oz. funkcije, ki jih lahko uporabimo v našem delovnem toku, da si olajšamo nekatere pogoste postopke. GitHub Actions je na voljo za vse repozitorije in, v primeru da je repozitorij odprtokoden, omogoča neomejeno število minut izvajanja delovnih tokov. Dokumentacija za GitHub Actions je dobra, obsežna in ponuja obsežne opise funkcij ter sintakse za pisanje, s praktičnimi primeri v obliki izsekov kode.

Postavitev CI/CD funkcionalnosti na platformi je preprosta, enostavna in nedvoumna. Ko imamo v repozitoriju pripravljene prve teste, na glavni strani repozitorija izberemo zavihek Actions. V primeru, da nimamo izdelanih še nobenih akcij nas zavihek pošlje na stran z uradnimi in neuradnimi predlogami za izdelavo novih akcij. GitHub analizira projekt in ponudi nekaj najbolj verjetnih možnosti. V primeru, da predlogi niso v redu, ponudi iskalnik. Če želimo akcijo spisati popolnoma sami, lahko izberemo možnost za prazno datoteko. V mojem primeru je platforma avtomatsko predlagala predlogo za avtomatizacijo testiranja aplikacije Node.js. Slika 4.2 prikazuje

predlagani predlogi, kjer vidimo predlagano predlogo za testiranje v okolju Node.js.



Slika 4.2: Predlagane predloge

S klikom na predlogo se odpre spletni urejevalnik datoteke YAML, v kateri bo shranjena naša akcija. Spremenil sem ime akcije, dodal sem polje `paths-ignore` in dodal ignoriranje sprememb v datoteki `README.me`. Na koncu sem odstranil še korak, ki poskuša zagnati ukaz `npm build --if-present`, saj sem vedel da ta ukaz ni potreben - ukaz zažene postopek graditve, če je ta potreben za delovanje projekta oz. definiran v konfiguracijski datoteki. Za moj projekt graditev ni potrebna, zato sem ukaz odstranil. Slika 4.3 prikazuje kodo datoteke YAML, ki bo izvedla avtomatsko testiranje.

Datoteko sem uveljavil v repozitorij in ob naslednjih spremembah se je akcija uspešno sprožila in proces avtomatskega testiranja se je začel izvajati.

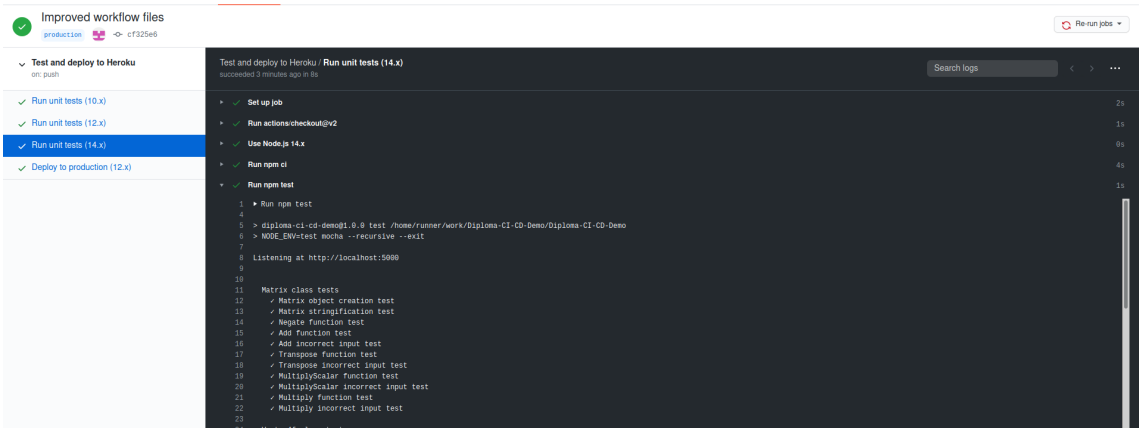
Uporabniški vmesnik za pregled nad izvajanjem akcij je enostaven in funkcionalen. Na levi strani je prikazan seznam korakov in njihovo stanje, na desni pa je prikazan izhod, ki se izpiše med izvajanjem korakov. Slika 4.4 prikazuje grafični uporabniški vmesnik za pregled izvajanja delovnega toka. Edina stvar, ki jo pogrešam, je pogled v obliki drevesa oz. način prikaza, kjer lahko enostavno vidimo tudi odvisnosti med koraki (recimo če je korak postavitve je odvisen od korakov testiranja). Zanimivo je, da je grafični urejevalnik delovnih tokov obstajal v prvih različicah GitHub Actions. Z izdajo GitHub Actions V2 pa je bil urejevalnik odstranjen - nov urejevalnik naj bi bil dodan v eni izmed naslednjih različic.

```yaml
name: Unit tests

on:
  push:
    branches: [ master ]
    paths-ignore:
    - 'README.md'
  pull_request:
    branches: [ master ]
    paths-ignore:
    - 'README.md'

jobs:
  test:
    name: Run unit tests
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [10.x, 12.x, 14.x]
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - run: npm ci
    - run: npm test
```

Slika 4.3: Korak testiranja



Slika 4.4: Uporabniški vmesnik za pregled izvajanja akcij na platformi GitHub

Testi se izvedejo v treh okoljih Node.js. Sprva sem mislil nastavitev spremeniti, potem pa sem se odločil, da je testiranje v več različnih okoljih Node.js pametno, čeprav za moj enostaven primer mogoče nepotrebno.

Ko sem pričel z ustvarjanjem akcije za postavitev na platformo Heroku sem naletel na nekaj težav. Začel sem z dodajanjem ključa Heroku API

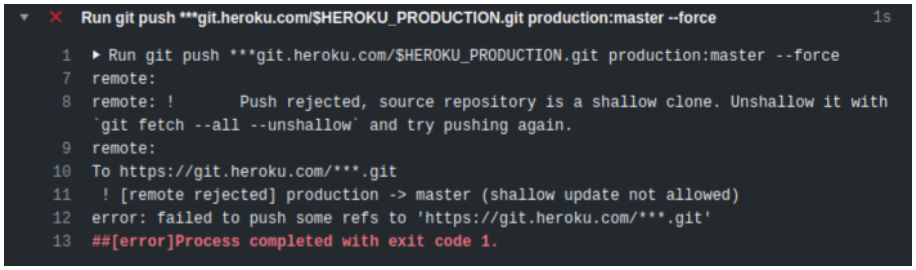
in imenom moje Heroku aplikacije med zaščitene spremenljivke v GitHub repozitorij. Nova datoteka YAML je bila dokaj podobna tisti za avtomatsko testiranje, tako da sem za osnovo vzel kar datoteko za CI. Zamenjal sem ime in spremenil ime veje, ki akcijo sproži. Na koncu sem dodal še korak, ki sproži ukaz `shell` z dvema okoljskima spremenljivkama. Pri tem koraku sem odstranil tudi izvajanje v več različnih verzijah okolja Node.js, saj želim, da se ta korak izvede le enkrat. Vrstica `needs: test` določi, da mora korak postavitve počakati z izvajanjem, dokler se prejšnji korak ne konča uspešno. Slika 4.5 prikazuje akcijo avtomatske postavitve.

```
27  deploy:
28    name: Deploy to production
29    needs: test
30    runs-on: ubuntu-latest
31    strategy:
32      matrix:
33        node-version:
34        - 12.x
35    steps:
36    - uses: actions/checkout@v2
37      with:
38        fetch-depth: 0
39    - name: Use Node.js ${{ matrix.node-version }}
40      uses: actions/setup-node@v1
41      with:
42        node-version: ${{ matrix.node-version }}
43
44    - shell: bash
45      env:
46        HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
47        HEROKU_PRODUCTION: ${{ secrets.HEROKU_APP_NAME }}
48      run: git push https://heroku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_PRODUCTION.git production:master
```

Slika 4.5: Korak postavitve

Zaradi konsistentnosti v primerjavi platform sem se odločil, da bom aplikacijo na Heroku posodabljal kar preko repozitorija Git aplikacije, tako da korak shell zažene enostaven `git push` ukaz. Datoteko sem uveljavil na repozitorij in v njem ustvaril novo vejo `production`. Na `master` vejo sem naložil nekaj posodobitev, nato pa sem spremembe združil še v vejo `production`. Akcija se je uspešno zagnala, testi so se uspešno izvedli, ampak korak za postavitev aplikacije na Heroku pa je javil napako. Slika 4.6 prikazuje sporočilo napake.

Heroku je spremembe zavrnil s sporočilom, da je repozitorij, s katerega se koda nalaga, t. i. plitek klon. Sam se s plitkimi kloni pri repozitorijih v



Slika 4.6: Napaka pri postavitvi

preteklosti še nikoli nisem srečal - mislil sem da sem nehote naredil kakšno napako ob posodabljanju repozitorija. Poskusil sem veliko možnih rešitev: pobrisal sem vejo `production` in jo poskusil ustvariti še enkrat, poskusil sem dodati opcijo `--force` na `git push` ukaz v datoteki YAML itd. Reševanje mi je vzelo kar nekaj časa, predvsem zato, ker na spletu ni veliko dokumentacije glede uporabe GitHub Actions za avtomatsko postavitev na Heroku. Večina ljudi uporablja neposredno integracijo med platformama ali pa uporabi predlogo, ki uporablja neuradno akcijo - tega nisem hotel, saj to prikrije vse kar se dogaja v ozadju. Nisem pa pomislil na eno malenkost - ko se izvede korak, ta izvede uradno akcijo `actions/checkout@v2`, ki klonira kopijo repozitorija. V primeru, da tej operaciji ne podamo argumenta `fetch-depth: 0` bo klon, ki ga akcija naredi, plitek, kar je povzročalo mojo težavo. Ko sem posodobil datoteko YAML in v repozitorij dodal spremembe je bila akcija izvedena uspešno in spremembe so se pojavile na aplikaciji.

4.2.3 Bitbucket

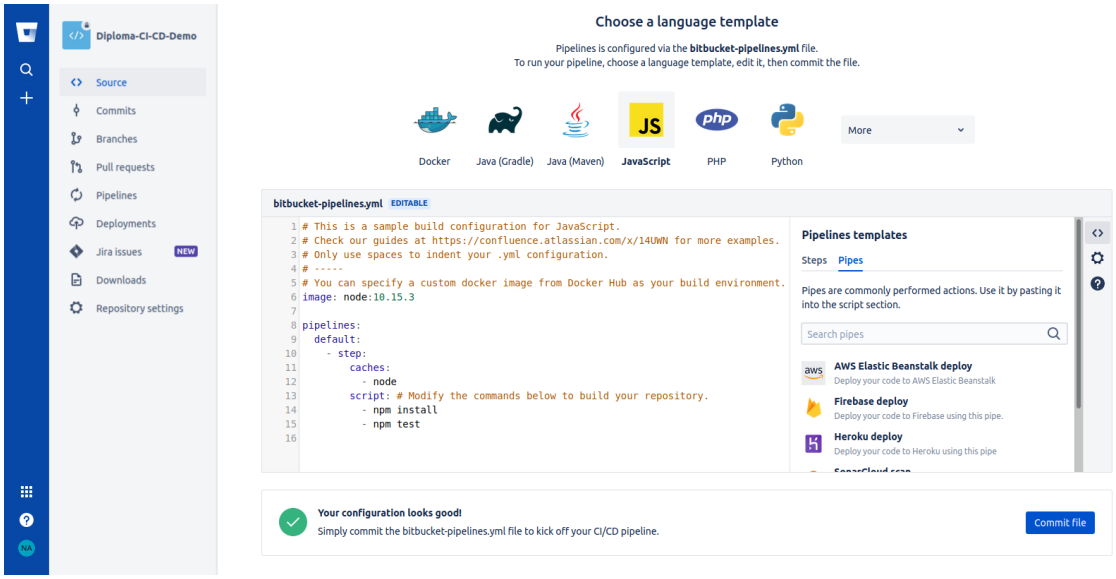
Na platformi Bitbucket lahko funkcionalnosti CI/CD omogočimo s funkcijo Bitbucket Pipelines. Definicija cevovodov oz. delovnih tokov se shrani v obliki datoteke YAML `bitbucket-pipelines.yml` v korenski mapi repozitorija. Ena izmed glavnih oz. najbolj opaznih razlik je, da za razliko od platforme GitHub, kjer lahko akcije definiramo v več različnih datotekah, smo pri platformi Bitbucket omejeni na samo eno datoteko, ki vsebuje celotno definicijo delovnih tokov in cevovodov za repozitorij. Večjih prednosti ali slabosti

zaradi tega ni, predvidevam pa, da je lahko preglednost pri kompleksnih tokovih in obsežnih aplikacijah slabša. Še ena večja razlika med platformama je način izvajanja delovnih tokov - GitHub Actions akcije izvaja na virtualnem stroju z možnostjo izvajanja v Docker vsebniku, Bitbucket pa cevovode izvaja izključno v Docker vsebnikih. Dokumentacija za Bitbucket Pipelines je dobra in obsežna, obstajajo tudi uradni vodiči in uradni repozitoriji, v katerih je prikazana uporaba za nekatere najbolj pogoste funkcije.

Postavitev je tudi tukaj zelo hitra in enostavna. Začel sem s postavitvijo neprekinjene integracije. V repozitorij sem dodal prve datoteke in ko je bil ta pripravljen za prvo testiranje sem na glavni strani repozitorija kliknil gumb `Pipelines`. Bitbucket je zaznal, da repozitorij še nima ustvarjene datoteke `bitbucket-pipelines.yml` in me preusmeril na stran, ki na kratko predstavi funkcijo Bitbucket Pipelines in vpraša uporabnika za kateri jezik želijo spisati delovni tok. Ko uporabnik izbere jezik se pokaže spletni urejevalnik YAML in v njem se izpiše predloga za izbran jezik. V mojem primeru ta predloga ni potrebovala skoraj nobenih sprememb - spremenil sem samo ime in Docker vsebnik z različico Node.js, na kateri se bosta izvajala delovna tokova testiranja in postavitve. Slika 4.7 prikazuje spletni urejevalnik datotek YAML, ki ga ponuja platforma Bitbucket.

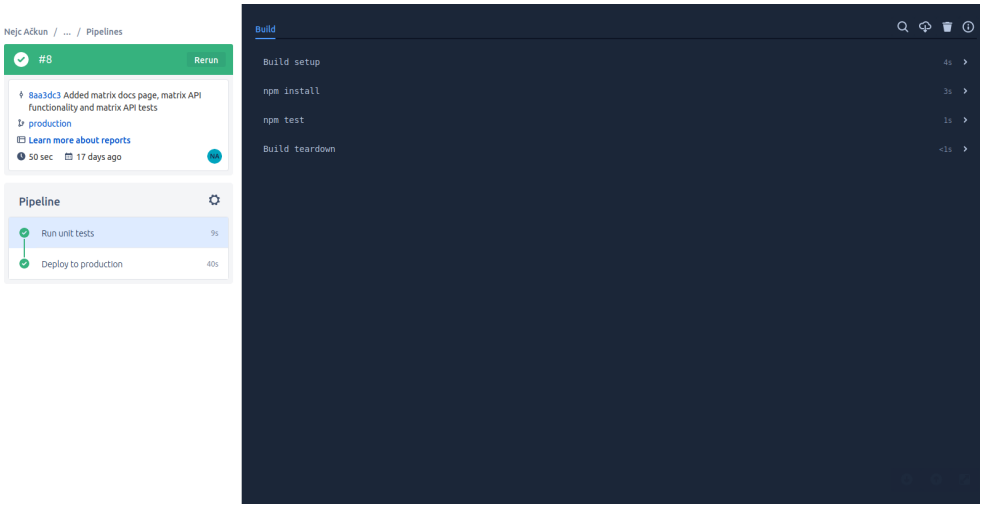
Na desni strani urejevalnika nam Bitbucket ponuja uporabo funkcije `Pipes` oz. cevi - vnaprej pripravljene funkcije oz. zaporedja ukazov za pogosto uporabljene akcije, ki jih lahko izvedemo med izvajanjem korakov v cevovodu in si tako olajšamo delo. Če kliknemo na katerega izmed predlogov, se prikaže okno z vsemi potrebnimi informacijami za uporabo - kaj je funkcija cevi, katere argumente sprejme itd. Naletel pa sem na manjši problem, in sicer spletni urejevalnik se je med uporabo večkrat sesul, kar je pomenilo, da sem izgubil vse spremembe. Delovni tok sem tako dokončno napisal lokalno. Ob uveljavitev datoteke na repozitorij se je delovni tok že avtomatsko zagnal.

Uporabniški vmesnik za pregled nad izvajanjem je zelo podoben tistemu na platformi GitHub, ampak dodanih je nekaj ključnih informacij. Na desni je prikazan izhod, ki se izpiše med izvajanjem testov, na levi pa je prikazanih



Slika 4.7: Spletni urejevalnik datotek YAML na platformi Bitbucket

nekaj podatkov o uveljavitvi, ki je izvajanje sprožila in osnovni podatki o samem izvajanju (čas in datum izvajanja). Na levi so prikazani tudi koraki v cevovodu - funkcija, ki ni bila prisotna na platformi GitHub. Slika 4.8 prikazuje grafični vmesnik za pregled nad izvajanjem cevovodov.



Slika 4.8: Uporabniški vmesnik za pregled nad izvajanjem cevovoda na platformi Bitbucket

V repozitorij sem nato dodal vse potrebne datoteke za postavitev aplikacije na platformo Heroku ter obe potrebni zaščiteni spremenljivki. Ko sem začel posodabljati datoteko `bitbucket-pipelines.yml` sem v dokumentaciji opazil, da Bitbucket Pipelines omogoča dodajanje neke vrste kazalcev na definicije različnih delovnih tokov. Slika 4.9 prikazuje uporabo kazalcev v končni datoteki YAML.

```yaml
test: &test
  step:
    name: Run unit tests
    caches:
      - node
    script:
      - npm install
      - npm test

deploy: &deploy
  step:
    name: Deploy to production
    script:
      - git push https://heroku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_APP_NAME.git production:master

pipelines:
  default:
  - <<: *test
  branches:
    master:
    - <<: *test
    production:
    - <<: *test
    - <<: *deploy

  pull-requests:
    master:
    - <<: *test
```

Slika 4.9: Prikaz uporabe kazalcev

To omogoča enostavno definiranje cevovodov za različne veje repozitorija. Definiral sem dva delovna tokova - `test`, ki sproži testiranje in `deploy`, ki sproži ukaz Git za nalaganje kode na Heroku. Tudi tokrat sem se odločil za neposredno nalaganje sprememb na repozitorij Git Heroku aplikacije, da bo primerjava kar se da objektivna - na voljo je bila tudi cev za nalaganje na Heroku, ampak se za uporabo nisem odločil. Ko definiramo izvajanje cevovodov se lahko sklicujemo kar na kazalce, ki smo jih definirali. Tako nam ni potrebno pisati enake kode večkrat če recimo delovni tok uporabimo večkrat za različne veje repozitorija, kot velja v mojem primeru.

Spremembe v datoteki sem uveljavil na repozitorij in ustvari novo vejo `production`. V njo sem združil spremembe iz veje `master`, kar je sprožilo

izvajanje cevovoda, ki se je uspešno izvedel in spremembe so bile takoj vidne tudi na aplikaciji Heroku.

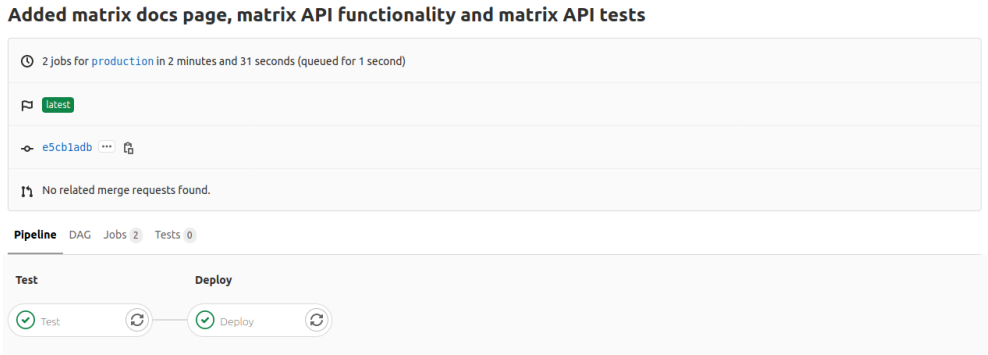
4.2.4 GitLab

CI/CD na platformi GitLab omogočimo enako kot na platformi Bitbucket - v korensko mapo repozitorija dodamo datoteko YAML z imenom `.gitlab-ci.yml`, ki hrani vse naše delovne tokove in cevovode. GitLab, tako kot Bitbucket, delovne tokove izvaja izključno v Docker vsebnikih, daje pa nam tudi možnost postavitve lastnih, osebnih "Runner"-jev - računalnikov oz. strežnikov, na katerih se naši tokovi izvajajo. Če se ne odločimo za to možnost nam GitLab ponuja svoje "Shared Runner"-je, ki delujejo na oblačni storitvi Google Cloud Platform in uporabljajo izolirane virtualne stroje za vsak projekt, kar zagotavlja varnost. Nad dokumentacijo za pisanje datoteke YAML sem bil malce razočaran, saj ta ni tako dobra kot pri drugih dveh platformah - primerov je malo in večinoma so napisani za samo en primer uporabe, opisi funkcij, ki jih lahko uporabimo v datoteki YAML so zelo osnovni in celotna dokumentacija je težje razumljiva kot pri drugih dveh platformah. Način definiranja tokov v datoteki YAML je tukaj spet drugačen, torej datoteke niso prenosljive, in osebno mi je ta način občutno manj pregleden kot način definiranja na platformi Bitbucket. Po robustnosti in fleksibilnosti pa je sistem na isti ravni kot druge dve platformi.

Postavitev funkcionalnosti CI/CD na tej platformi je enostavna, a traja dlje časa kot pri drugih dveh platformah, predvsem zaradi slabše dokumentacije. Za postavitev CI funkcij sem v repozitorij najprej dodal vse potrebne datoteke za izvajanje prvih testov, nato pa sem v levem meniju kliknil gumb `CI/CD`. GitLab ne ponuja nobene vrste spletnega urejevalnika datotek YAML - uporabnika pošlje na stran v dokumentaciji, ki opiše postopek postavitve cevovoda CI/CD. Platforma ponuja preverjevalnik za datoteke YAML in neuraden grafični vmesnik za konceptualno načrtovanje cevovodov - uradna različica, ki bo omogočala grafično ustvarjanje dejanskih cevovodov je trenutno v razvoju. Datoteko `.gitlab-ci.yml` sem ustvaril ročno v korenu

repozitorija in dodal osnovni delovni tok za testiranje. Ta se je le minimalno razlikoval od tistega na platformi Bitbucket, tako da mi ni delal pretiranih težav. Izvajanje cevovoda se je sprožilo takoj po uveljavitvi sprememb.

Uporabniški vmesnik za pregled nad izvajanjem se grafično razlikuje od ostalih dveh platform, ampak ponuja enake funkcionalnosti kot platforma Bitbucket. V zgornjem delu so prikazani podatki o samem cevovodu, stanju cevovoda in podatki o uveljavitvi, ki je izvajanje sprožila. V spodnjem delu pa so koraki v cevovodu prikazani kot objekti, ki so med seboj povezani če obstaja odvisnost. Ta vmesnik je enostaven za razumevanje in ponudi hiter pregled nad zgradbo cevovoda. Če izberemo katerega izmed objektov se odpre prikaz izhoda, ki se izpiše med izvajanjem. Na sliki 4.10 je prikazan grafični vmesnik za pregled nad izvajanjem cevovodov.



Slika 4.10: Uporabniški vmesnik za pregled nad izvajanjem cevovoda na platformi GitLab

Dodal sem še datoteke potrebne za postavitev CD in se lotil posodabljanja datoteke YAML. Postavitev CD pa mi je vzela kar nekaj časa - predvsem definiranje sprožilcev za različne veje. Če sem dokumentacijo prav razumel, GitLab-ova izvedba vedno sproži vse korake v cevovodu, za omejevanje tega pa se zanaša na pravila v slogu *samo če/only* in *razen če/except*, ki imajo v ozadju skrito kar kompleksno plast medsebojne odvisnosti. Še večji problem nastane ker enaka pravila uporabljamo tudi za definiranje datotek, katerih spremembe naj ne sprožijo delovnega toka in še nekaj drugih podobnih funk-

cij. Obstaja tudi pravilo *ko/when*, ki pa se obnaša kot neke vrste if stavek. Dokumentacijo sem bral kar nekaj časa preden sem uspel napisati končno skripto. Slika 4.11 prikazuje končno datoteko YAML.

```yaml
image: node:latest

cache:
  paths:
    - node_modules/

stages:
    - test
    - deploy

Test:
  stage: test
  script:
    - npm install
    - npm test

Deploy:
  only:
    - production

  stage: deploy
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=$HEROKU_APP_NAME --api-key=$HEROKU_API_KEY
```

Slika 4.11: Končna datoteka YAML na platformi GitLab

Tokrat za nalaganje na Heroku nisem uporabil ukaza `git push`, ampak Ruby ukaz `dpl`, ki aplikacijo samodejno naloži na platformo Heroku. Zaradi objektivnosti sem poskusil zadevo izvesti tako kot v drugih dveh primerih z uporabo Git, ampak nisem našel nobene dokumentacije na to temo - vsa dokumentacija za Heroku je kodo posodabljala z ukazom `dpl`.

4.2.5 Povzetek

Če povzamem trenutno poglavje - najboljšo dokumentacijo ponujata platformi GitHub in Bitbucket. Dokumentacija obeh platform je obsežna, razumljiva in ponuja praktične primere za skoraj vse funkcije. GitLab-ova dokumentacija pa je težje razumljiva, praktični primeri pa so pomanjkljivi.

Najboljšo izvedbo funkcionalnosti CI/CD ima po mojih izkušnjah GitHub, saj je postavitev zelo enostavna, zgradba in uporabljeni ukazi v dato-

tekah YAML so najbolj logični, možnost uporabe več različnih datotek pa pripomore pri berljivosti in razumljivosti delovnih tokov. Bitbucket se s svojo izvedbo zelo približa platformi GitHub - postavitev je enako lahka in zgradba datotek YAML je primerljivo logična in jasna. Bitbucket omogoča uporabo samo ene datoteke za shranjevanje delovnih tokov, ampak z uporabo kazalcev ta datoteka vseeno ostane enostavna za branje in razumevanje. GitLab s svojo zgradbo datotek YAML in slabšo dokumentacijo ne ponuja enako dobre izkušnje kot drugi dve platformi.

Grafični vmesnik pa je najbolje izveden na platformah Bitbucket in GitLab. Zelo dober - in mogoče grafično najbolj dodelan - je vmesnik platforme Bitbucket, ki je po mojem mnenju najbolj pregleden in uporabniku prijazen. Vmesnik platforme GitLab ponuja jasen in enostaven pregled nad stanjem posameznih delovnih tokov v cevovodu. Vmesnik platforme GitHub pa je zelo osnoven, ampak funkcionalen - razlog za to je verjetno to, da je funkcija GitHub Actions na voljo kratek čas - prva različica je bila izdana novembra 2019.

Poglavje 5

Ugotovitve in priporočila

Ugotovil sem, da vse preučevane platforme uporabljajo zelo podobne izvedbe CI/CD funkcionalnosti. Na vseh treh platformah so delovni tokovi in cevovodi definirani z datotekami YAML, ki so shranjene v korenu repozitorija ali pa v korenski mapi v primeru platforme GitHub. Način pisanja in ključne besede, ki se uporabljajo v datotekah se med platformami razlikujejo. Same akcije se izvajajo neposredno v virtualnih strojih ali pa v Docker vsebnikih. GitHub in GitLab omogočata testiranje v treh različnih operacijskih sistemih - Linux, MacOS in Windows, uporabnikom pa omogočata tudi postavitev lastnih agentov za poganjanje delovnih tokov. GitHub, gledano s čisto teoretičnega vidika, ne podpira cevovodov, vendar enake funkcionalnosti dosega z možnostjo uporabe mnogih datotek za delovne tokove, za razliko od platform Bitbucket in GitLab, kjer smo omejeni na eno samo datoteko. Akcije so se najhitreje izvedle na platformi Bitbucket, kjer je testiranje trajalo povprečno 12 sekund, postavitev pa 51 sekund. Naslednja najhitrejša je bila platforma GitHub, kjer je testiranje trajalo povprečno 32 sekund, postavitev pa 64 sekund. Zadnja, najpočasnejša, pa je bila platforma GitLab, kjer je testiranje trajalo povprečno 54 sekund, postavitev pa 152 sekund. GitLab in Bitbucket sta akcije izvajala v Docker vsebnikih, GitHub pa neposredno v virtualnem stroju.

Iz primerjav v prejšnjem poglavju in praktičnega dela na platformah sem

izpeljal naslednja priporočila za različne tipe projektov in ekip.

5.1 Samostojni razvijalci in manjše ekipe

Za samostojne razvijalce in manjše ekipe, ki razvijajo odprtokodno programsko opremo ali odprtokodne sisteme na kateremkoli nivoju je platforma GitHub verjetno najboljša izbira. Platforma ne omejuje velikosti ekipe, brez doplačila pa nam je omogočen dostop do praktično vseh funkcij, ki jih platforma ponuja. Je platforma z največjo skupnostjo in velja kot "go-to" platforma za odprtokodne projekte in upravljanje z izvorno kodo. Večja podjetja ponavadi ob razgovorih za zaposlitve vprašajo kandidate po GitHub računu, ki služi kot portfelj kandidatovih del in prikaz njegovih izkušenj. Če pa za delo potrebujemo privatne repozitorije, pa je dobra izbira tudi GitLab, ki te funkcionalnosti ne omejuje pri zastonjskih uporabnikih in doda še osnovne funkcije za projektno vodenje brez potrebe po zunanjih orodjih.

5.2 Samostojne ekipe in manjša podjetja

Za samostojne ekipe ali podjetja, ki pri svojem delu želijo boljši pregled na porabo časa, lažje razporejanje dela članom ekipe itd. je najboljša platforma GitLab, ki združuje vse potrebne funkcije za razvoj na enem mestu v paketu Starter/Bronze. Bitbucket ponuja dober pregled in upravljanje z izvorno kodo, vendar pa se za projektne funkcije zanaša na integraciji z orodjema Trello in Jira, ki sta oba na voljo zastonj, a z omejenimi funkcijami. Če ima naša ekipa manj kot 10 članov, je Bitbucket Standard and Premium tudi zelo konkurenčna izbira proti GitLab-u, če pa je ekipa večja pa je potrebna naročnina še na orodje Jira, da lahko izkoristimo vse funkcije obeh. GitHub je s paketom Team še vedno zelo dobra izbira z vidika upravljanja z izvorno kodo, za projektne funkcije pa se platforma zanaša na svoj ekosistem integracij z zunanjimi orodji, med drugim tudi z orodjema Jira in Trello.

5.3 Rastoče organizacije in večja podjetja

Za rastoče organizacije in podjetja z več ločenimi razvojnimi ekipami pa predlogi niso več tako enostavni. GitLab v paketu Silver/Premium ponuja ogromno funkciji za podporo med-ekipnega dela in ostalih operacij, ki so prisotne v ekipah s takšnim obsegom. Bitbucket pa s svojo izvedbo ponuja ekipam modularno dodajanje orodji, ki odgovarjajo njihovim zahtevam. Ena izmed glavnih prednosti platforme Bitbucket je ekosistem lastnih orodji, ki jih ponuja podjetje Atlassian. Orodja so uveljavljena, poznana in zaupanja vredna ter kvalitetna. GitHub prav tako ponuja paket Enterprise, ki je namenjen med-ekipnemu sodelovanju in ponuja funkcije, ki se osredotočajo na varnost projektov, hitrejši razvoj in boljšo podporo.

Poglavje 6

Zaključek

V diplomski nalogi sem predstavil in primerjal tri platforme, ki so v svetu razvoja programske opreme zelo uporabljene. Platforme, ki sem jih primerjal, so tri "go-to" možnosti pri modernem razvoju programske opreme in veljajo za uveljavljene, stabilne in varne izbire ko se odločamo za platformo. Obstaja tudi veliko drugih platform - nekatere so še v postopku razvoja, ki ponujajo podobne oz. na nekaterih področjih tudi boljše funkcije, kot te, ki sem jih primerjal. Pri izbiri platforme za funkcionalnosti CI/CD smo lahko torej zelo striktni z zahtevami, saj bomo po vsej verjetnost našli platformo, ki našim zahtevam odgovarja.

Literatura

[1] Amazon Web Services (AWS): What is Continuous Delivery? Dosegljivo: https://aws.amazon.com/devops/continuous-delivery/. [Dostopano: 9. 8. 2020].

[2] Atlassian Acquires Bitbucket.org, Distributed Version Control System Hosting. Dosegljivo: https://www.atlassian.com/blog/archives/atlassian_acquires_bitbucket_org_distributed_version_control_system_hosting. [Dostopano: 25. 8. 2020].

[3] Atlassian Press Kit. Dosegljivo: https://www.atlassian.com/company/news/press-kit. [Dostopano: 25. 8. 2020].

[4] Atlassian: What is Continuous Deployment? Dosegljivo: https://www.atlassian.com/continuous-delivery/continuous-deployment. [Dostopano: 9. 8. 2020].

[5] Bitbucket. Dosegljivo: https://bitbucket.org/. [Dostopano: 25. 8. 2020].

[6] Brošura za orodje Ballista. Dosegljivo: https://users.ece.cmu.edu/~koopman/ballista/ballista_brochure.pdf. [Dostopano: 27. 8. 2020].

[7] ECMAScript® 2020 Language Specification. Dosegljivo: https://www.ecma-international.org/ecma-262/. [Dostopano: 4. 9. 2020].

[8] GitHub. Dosegljivo: https://github.com/. [Dostopano: 25. 8. 2020].

[9] GitHub Actions now supports CI/CD, free for public repositories. Dosegljivo: https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/. [Dostopano: 9. 8. 2020].

[10] GitHub Logos and Usage. Dosegljivo: https://github.com/logos. [Dostopano: 25. 8. 2020].

[11] GitLab. Dosegljivo: https://about.gitlab.com/. [Dostopano: 25. 8. 2020].

[12] GitLab: Is it any good? Dosegljivo: https://about.gitlab.com/is-it-any-good/. [Dostopano: 25. 8. 2020].

[13] GitLab: Our stewardship of GitLab. Dosegljivo: https://about.gitlab.com/company/stewardship/. [Dostopano: 25. 8. 2020].

[14] GitLab Press Kit. Dosegljivo: https://about.gitlab.com/press/press-kit/. [Dostopano: 25. 8. 2020].

[15] Knjižnica Chai. Dosegljivo: https://www.chaijs.com/. [Dostopano: 27. 8. 2020].

[16] Microsoft to acquire GitHub for 7.5 billion dollars. Dosegljivo: https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/. [Dostopano: 9. 8. 2020].

[17] Modul nyc. Dosegljivo: https://istanbul.js.org/. [Dostopano: 1. 9. 2020].

[18] Modul SuperTest. Dosegljivo: https://github.com/visionmedia/supertest. [Dostopano: 27. 8. 2020].

[19] movingtogitlab Twitter kanal. Dosegljivo: https://twitter.com/movingtogitlab. [Dostopano: 25. 8. 2020].

[20] Orodje Bootstrap. Dosegljivo: https://getbootstrap.com/. [Dostopano: 25. 8. 2020].

[21] Platforma Node.js. Dosegljivo: `https://nodejs.org/en/`. [Dostopano: 25. 8. 2020].

[22] Predloga Bootstrap Album. Dosegljivo: `https://getbootstrap.com/docs/4.5/examples/album/`. [Dostopano: 25. 8. 2020].

[23] Pripomoček nodemon. Dosegljivo: `https://nodemon.io/`. [Dostopano: 27. 8. 2020].

[24] Testno ogrodje Mocha. Dosegljivo: `https://mochajs.org/`. [Dostopano: 27. 8. 2020].

[25] Uradna stran jezika YAML. Dosegljivo: `https://yaml.org/`. [Dostopano: 25. 8. 2020].

[26] Vmesno programje morgan. Dosegljivo: `https://github.com/expressjs/morgan`. [Dostopano: 27. 8. 2020].

[27] Wikipedia: BoundsChecker. Dosegljivo: `https://en.wikipedia.org/wiki/BoundsChecker`. [Dostopano: 27. 8. 2020].

[28] John D Blischak, Emily R Davenport, and Greg Wilson. A quick introduction to version control with Git and GitHub. *PLoS computational biology*, 12(1):e1004668, 2016.

[29] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings Publishing Company, 1991.

[30] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. In *International Conference on the Unified Modeling Language*, pages 194–208. Springer, 2001.

[31] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[32] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra tool set (software testing). In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, volume 2, pages 275–284 vol.2, 1989.

[33] Wouter de Kort. What Is DevOps? In *DevOps on the Microsoft Stack*, pages 3–8. Springer, 2016.

[34] E. Dijkstra. *Structured Programming*, page 41–48. Yourdon Press, USA, 1979.

[35] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.* Addison-Wesley, 2009.

[36] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 235–245, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. In *International Conference on Product Focused Software Process Improvement*, pages 3–16. Springer, 2010.

[38] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[39] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[40] Sidney L Hantler and James C King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.

[41] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[42] JC Huang. An approach to program testing. *ACM Computing Surveys (CSUR)*, 7(3):113–128, 1975.

[43] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[44] Juha Itkonen, Raoul Udd, Casper Lassenius, and Timo Lehtonen. Perceived benefits of adopting continuous delivery practices. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery.

[45] James C King. A new approach to program testing. *ACM Sigplan Notices*, 10(6):228–233, 1975.

[46] Gordon Kotik and Lawrence Markosian. Automating software analysis and testing using a program transformation system. *ACM SIGSOFT Software Engineering Notes*, 14(8):75–84, 1989.

[47] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[48] William E Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. CRC press, 2017.

[49] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[50] A. Mardan. *Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development.* Expert's voice in Web development. Apress, 2014.

[51] Harlan D Mills, Michael Dyer, and Richard C Linger. Cleanroom software engineering. 1987.

[52] Glenford J Myers. *Software Reliability.* John Wiley & Sons, Inc., 1976.

[53] Michael R Paige. An analytical approach to software testing. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC'78.*, pages 527–532. IEEE, 1978.

[54] Chittor V Ramamoorthy and Sill-bun F Ho. Testing large software with automated software evaluation systems. In *Proceedings of the international conference on Reliable software*, pages 382–394, 1975.

[55] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[56] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[57] Ravishankar Somasundaram. *Git: Version control for everyone.* Packt Publishing Ltd, 2013.

[58] I. Sommerville. *Software Engineering, Global Edition.* Pearson Education Limited, 2016.

[59] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[60] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014.

[61] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816, 2015.

[62] Elaine J Weyuker and Thomas J Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, 1980.