VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS STUDIJŲ PROGRAMA

Nuolatinio tiekimo grandinė Java mikroservisų architektūrinio stiliaus projektuose

Continuous Delivery Pipeline in Java Microservices Architectural Style Projects

Bakalaurinis darbas

Atliko: Lukas Milašauskas (parašas)

Darbo vadovas: Lekt. Gediminas Rimša (parašas)

Vilnius – 2021

TURINYS

Įvadas

Programinės įrangos (toliau PĮ) diegimas į testavimo ar produkcinę aplinkas, dažnai būna daug laiko reikalaujantis procesas. Informacinės sistemos, turinčios daug komponenčių, pavyzdžiui mikroservisų architektūrinio stiliaus, reikalauja daug žingsnių, patikrinimų norint užtikrinti sklandų diegimą. Didelio skaičiaus skirtingų žingnsių atlikimas nevisada vyksta sklandžiai ir atsiranda žmogiškų klaidų. Įsivaizduokime, kad kuriamos informacinės sistemos produkcinėje aplinkoje atsirado kritinė klaida, be kurios sistemos vartotojai negali vykdyti svarbių operacijų. Pataisymas padarytas ir ištestuotas, tačiau jo diegimas į aplinkas yra procesas, kurį pakartoti užtrunka ilgai ir jis susideda iš labai daug žingsnių.

Vienas iš šios problemos sprendimų yra nuolatinis tiekimas. Šio sprendimo esmė yra tiksliai apibrėžti, kokiu būdu nauji PĮ kodo pakeitimai galėtų atsidurti sistemos aplinkose, bei užtikrinti, kad kodas visada būtų paruoštas diegimams. PĮ kelią nuo programuotojų kompiuterių iki produkcinės aplinkos galima apibrėžti kaip naudojamų praktikų grandinę. Egzistuoja daug būdų, kaip galima būtų įgyvendinti šią grandinę, tačiau užtikrinti sklandžią proceso eigą ir suderinti galimas praktikas gali būti sudėtinga. Taip pat, svarbu įsitikinti ar naudojama grandinė tikrai yra kokybiška. Šiam tikslui egzistuoja brandos modelis, išgryninantis nuolatinio tiekimo grandinės kokybės matus.

Šiame darbe pabandysime suprojektuoti ir įgyvendinti nuolatinio tiekimo grandinės šabloną, kuris užtikrintų aukštą, techninėmis galimybėmis pasiekiamą, brandos lygį. Taip pat, vis labiau tobulėjant debesų kompiuterijos sprendimams, šablone stengsimes kiek įmanoma, vengti savų resursų naudojimo.

Nuolatinio tiekimo grandinėse naudojamos praktikos ir įrankiai dažnai priklauso nuo programavimo kalbos, kuria rašoma pati PĮ, todėl grandinė bus pritaikyta Java mikroservisų architektūrinio stiliaus projektams. Taip pat, šiame darbe nebus kalbama apie duomenų bazių administravimą nuolatinio tiekimo kontekste.

Darbo tikslas

Suprojektuoti ir realizuoti nuolatinio tiekimo grandinės šabloną Java mikroservisų architektūrinio stiliaus projektams, kuris užtikrintų aukštą, techninėmis galimybėmis įgyvendinamą, nuolatinio tiekimo brandos lygį.

Uždaviniai tikslui pasiekti

1. Atlikti literatūros apžvalgą, siekiant apsibrėžti grandinės dalis ir jų gerąsias praktikas.
2. Suprojektuoti aukšto nuolatinio tiekimo brandos lygio grandinės šabloną Java mikroservisams.
3. Įgyvendinti suprojektuotą grandinės šabloną.

1. Kas yra nuolatinis tiekimas (angl. „Continuous Delivery").

1.1. Atsiradimas ir principai

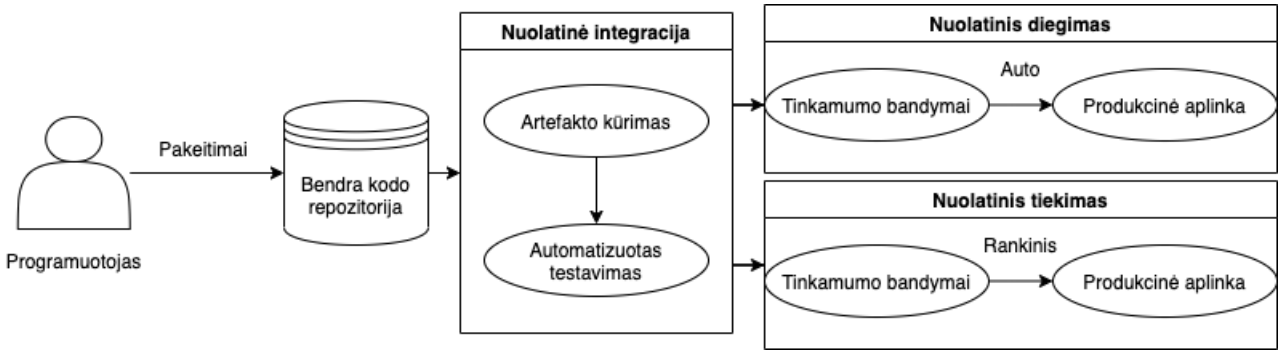
Apie nuolatinio tiekimo samprata ir praktikas pirmą kartą rašė Jez Humble ir David Farley savo knygoje „Continuous Delivery" [HF10] dar 2010 metais. Savo knygoje jie teigia, kad toks terminas buvo pasirinktas iš pirmo Agile programinės įrangos kūrimo manifesto principo: „Mūsų aukščiausias prioritetas yra patenkinti kliento poreikius, anksti ir nuolat pristatant vertę kuriančia programinę įrangą". Taigi, šis terminas atspindi procesų ir praktikų visumą, kurių pagalba būtų galima kuo dažniau ir efektyviau tiekti programinę įrangą. Programinės įrangos tiekimas suprantamas ne tik kaip programinės įrangos kūrimas, bet kaip jos kokybės ir stabilumo užtikrinimas, bei diegimas. Siekiant užtikrinti šiuos dalykus naudojama nemaža aibė praktikų, kurios turi būti suderinamos tarpusavyje ir palengvinti, o ne sunkinti PĮ diegimo procesą ir sumažinti jo trukmę. Remiantis Martin Fowler straipsniu „ContinuousDelivery" [Fow13] nuolatinio tiekimo pagrindiniai privalumai yra tokie:

1. ***Sumažinta diegimo rizika***. Dėl mažo kiekio diegiamų pakeitimų, atsiranda mažesnė terpė klaidoms ir mažesnio masto klaidos galėtų atsirasti, kurias taip pat lengviau ir pataisyti.
2. ***Įtikimas progresas***. Yra daug praktikų, ką laikyti pabaigta užduotimi. Pabaigta užduotis, kai ji yra įdiegta į tam tikrą sistemos aplinką yra žymiai patikimesnis rodiklis, negu programinės įrangos kūrėjo teigimas, kad užduotis baigta.
3. ***Vartotojų atsiliepimai***. Didžiausia rizika programinei įrangai sudaro jos panaudojamumas. Visada norima kurti programinę įranga, kuri yra naudojama. Mažesni ir dažnesni diegimai užtikrina, kad tikri vartotojai greičiau pamato sukurtą rezultatą ir gali įvertinti jo naudą.

1.2. Nuolatinė integracija, nuolatinis tiekimas, nuolatinis diegimas

Kalbant apie nuolatinį tiekimą privalu paminėti ir kitus susijusius terminus. Nuolatinė integracija, nuolatinis tiekimas ir nuolatinis diegimas yra trys praktikos, kurios glaudžiai susijusios savo reikšmėmis ir atsakomybėmis. Remiantis straipsniu „Continuous Integration Theater" [FFdC+19] nuolatinė integracija (angl. *„Continuous Integration"*) apibūdinama kaip praktika, įgalinanti programuotojus apjungti savo atskiras PĮ kodo kopijas į pagrindinę kopiją bent kelis kartus per dieną. Šios praktikos tikslas yra automatizuoti naujų pakeitimų kokybės užtikrinimą ir pagreitinti diegimo ciklą, taip didinant komandos, kuri dirba prie projekto produktyvumą. Šio darbo rašymo metu egzistuoja daug įrankių, kurių pagalba galima automatizuoti PĮ kodo testavimą, analizę ir konfigūracijų valdymą su kiekvienu pakeitimu ar diegimu, taip pat, interaktyviai valdyti procesus. Nors pagal apibrėžimą nuolatinė integracija ir nuolatinis tiekimas turi skirtingas atsakomybes, jie yra susiję tuo, kad nuo integracijos metu kuriamų artefaktų ir pačios integracijos eigos priklauso ar bus galima įgyvendinti nuolatinį tiekimą, todėl nuolatinė integracija yra būtina nuolatiniam tiekimui. Nuolatinio diegimo terminas atsirado metais anksčiau negu nuolatinio tiekimo ir buvo apibrėžtas Timothy Fitz straipsnyje „Continuous Deployment" [Fit09]. Ši praktika akcentuoja tai, kad kiekvienas pakeitimas turi būti automatiškai ir nuolat diegiamas į informacinės sistemos aplinką. Skirtumų tarp nuolatinio tiekimo ir nuolatinio diegimo nėra labai daug,

tačiau pasak Jez Humble straipsnio „Continuous Delivery vs Continuous Deployment“ [Hum10] nuolatinio tiekimo praktika ir terminas buvo sukurti todėl, kad nuolatinis diegimas užtikrina automatizuotą ir dažną diegimą, kur diegiama kiekviena stabili PĮ kodo versija. Nuolatinio tiekimo atveju, diegimai yra palikti ne PĮ kūrėjų rankose, bet verslo ar tų, kuriems priklauso sistema, tačiau nuolatinio tiekimo pagalba, taip pat užtikrinama, kad PĮ kodo versijos yra stabilios ir jas galima diegti nesunkiai ir automatizuotai rankiniu būdu. Autorius pabrėžia, kad kiekviena stabili kodo versija realiame pasaulyje sunkiai galėtų būti diegiama į produkcinę aplinką, nes dažnu atveju kelios skirtingos kodo versijos arba mikroservisų architektūrinio stiliaus atveju keli mikroservisai gali būti glaudžiai susiję. Pavienių mikroservisų diegimas privestų prie darbo su sistema stabdančių veiksnių, tačiau viskas priklauso nuo situacijos. Apibendrinant, šios trys praktikos yra gan glaudžiai susijusios. Nuolatinės integracijos pagalba galima programuotojų pakeitimus validuoti, verifikuoti, bei nesunkiai prijungti prie pagrindinės projekto kodo bazės. Nuolatinis diegimas ir nuolatinis tiekimas su nuolatinės integracijos pagalba padeda PĮ kūrėjams efektyviau atlikti diegimus ir juos valdyti. Šią sąveika tarp praktikų vaizduoja diagrama (1 pav.):



1 pav. Nulatinių procesų sąveika.

Nuolatinio tiekimo įgyvendinimą šiame darbe apibrėšime kaip gradinę praktikų, kurių pagalba funkcionalumų įgyvendinimas iš programuotojų kompiuterių nukeliautų iki produkcinės aplinkos.

1.3. Nuolatinio tiekimo grandinės dalys

Siekiant sudaryti nuolatinio tiekimo gradinės šabloną, reikia apsibrėžti gradinės dalis.

Knygoje „Continuous Delivery“ [HF10] yra keletas reikalavimų, be kurių neįmanoma įgyvendinti nuolatinio tiekimo:

1. Programinio kodo versijavimas.
2. Automatizuotas artefaktų kūrimas.
3. Bendras komandos susitarimas ir procesų išgryninimas.

Remiantis šiais punktais į grandinės šabloną būtina įtraukti kodo versijavimą, bei automatizuotą artefaktų kūrimą. Siekiant įgyvendinti nuolatinį tiekimą, būtinoji sąlyga yra nuolatinė integracija, todėl būtina į grandinės dalis įtraukti dar vieną svarbią nuolatinės integracijos praktiką - automatizuotą testavimą.

Nuolat tiekiant PĮ mikroservisų architektūrinio stiliaus sistemose būna funkcionalumų, kurių veikimas priklauso ne tik nuo vieno mikroserviso. Taip pat, sistemos sąsajomis gali naudotis ne tik vidinės sistemos dalys, bet ir kitos išorinės sistemos. PĮ diegimas ir suderinamumas reikalauja versijavimo, tik šiuo atveju ne kodo, bet pačios sistemos ar atskirų saityno paslaugų. Dėl šios priežasties į nuolatinio tiekimo grandinės šabloną įtrauksime sistemos versijavimo praktikas.

Informacinėse sistemose dažnai atsiranda poreikis turėti ne tik vieną produkcinę aplinką, bet ir testuotojams, programuotojams skirtą vieną ar kelias aplinkas. Skirtingos aplinkos dažnai reikalauja skirtingų sistemos nustatymų, o tam reikalingas konfigūracijų valdymas. Šia praktiką, taip pat, įsitrauksime į savo grandinės šabloną.

Kaip jau minėta prie apibrėžimo, PĮ diegimas ir jų valdymas yra svarbi nuolatinio tiekimo dalis, todėl ją būtina įtraukti į mūsų kuriamą gradinės šabloną.

Šiame darbe nenagrinėsime sistemų duomenų valdymo priemonių ir tokių praktikų kaip duomenų bazių migracijų.

Taigi, sudaromas nuolatinio tiekimo grandinės šablonas susidės iš šių dalių:

1. Programinio kodo versijavimas.
2. Automatizuotas artefaktų kūrimas.
3. Automatizuotas testavimas.
4. Sistemos versijavimas ir konfigūracijų valdymas.
5. PĮ diegimas.

2. Programinės įrangos kodo versijavimo šakų naudojimo strategijos naudojantis įrankiu Git

Remiantis knyga „Continuous Delivery“ [HF10] vienas iš pirminių kriterijų be kurių neįmanomas nuolatinis tiekimas yra versijų kontrolė. Kodo versijų sistemos įgalina programinės įrangos kūrėjus dirbti lygiagrečiai prie tos pačios kodo bazės ir kurti naujas projekto versijas, kurias būtų galima apjungti. 2018 metais Abraham Marín-Pérez ir Daniel Bryant išleistoje knygoje „Continuous Delivery in Java“ [MB18] autoriai teigia, kad 2005 metais atsiradęs paskirstyto (angl. *„distributed“*) kodo versijavimo įrankis Git knygos rašymo metu buvo pasirenkamas kaip dažniausiai naudojamas įrankis ir industrijos standartas, todėl šiame darbe kalbėsime būtent apie jį. Šis įrankis leidžia PĮ kūrėjams turėti atskirą vietinį serverį ar naudotis debesų kompiuterijos sprendimais, kur būtų saugomos projektų versijos, o visi komandos programuotojai galėtų parsisiųsti kodo šakas ar kurti naujas ir jas modifikuoti. Atlikus užduotis vėl sukelti į serverį, kur versijos apžvelgiamos komandos narių. Pakeitimų apžvalgos, kurias įgalina tokios kodo saugojimo sistemos kaip GitHub ar Gitlab, metu kiti programuotojai gali rašyti komentarus ir patarti kaip šiuos pakeitimus pagerinti ar paruošti sekantiems nuolatinio tiekimo žingsniams. Po apžvalgos, kai pakeitimai būna patvirtinti, jie prijungiami prie kitų kodo versijų, šakų. Git įrankyje pakeitimai užrašomi istorijoje užfiksavimo (angl. *„commit“*) pavidalu. Kiekviena kodo šaka savyje turi visus pakeitimų užfiksavimus nuo projekto pradžios. Apjungiant atskiras šakas tie užfiksavimai chronologine seka uždedami ant viršaus ir taip šakos apjungiamos, kol galiausiai turėtų susiliet į pagrindinę projekto šaką. Siekiant užtikrinti sklandų ir produktyvų darbą atsirado daug skirtingų darbo eigų, kaip šakos galėtų būti valdomos ir kuriamos kol pakeitimų įrašai nukeliauja į pagrindinę kodo repozitorijos (angl. *„repository“*) šaką. Toliau pateikiamos ir aprašomos dažnai naudojamos kodo versijavimo darbo eigos naudojantis įrankiu Git.

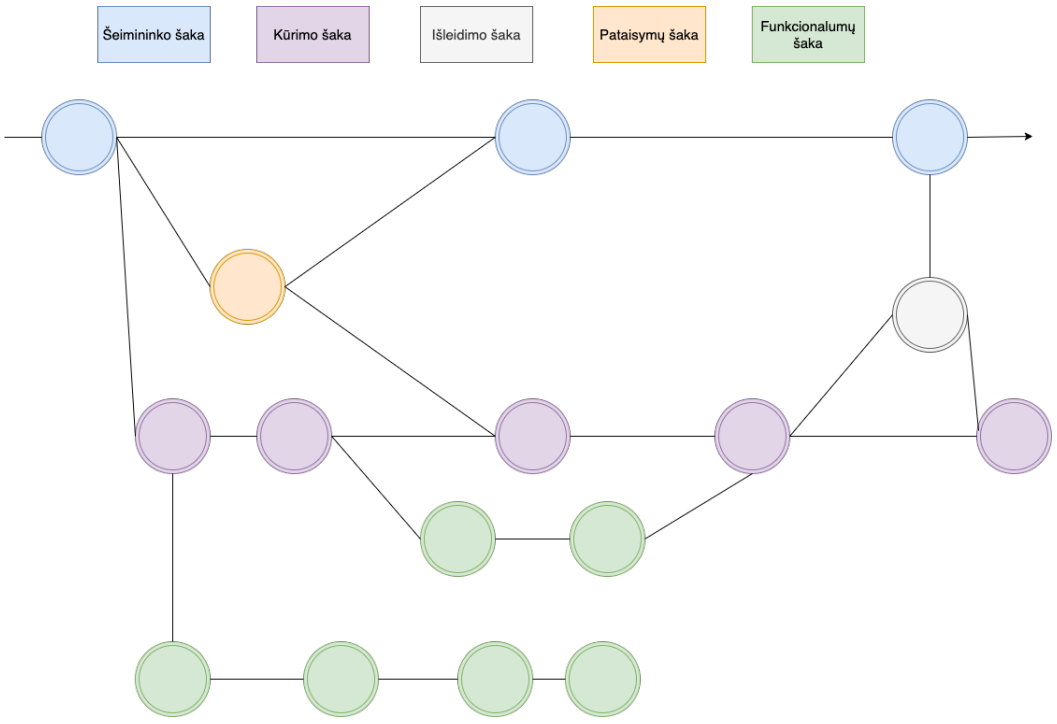
2.1. Pakeitimų užfiksavimo žinutės

Naudojantis Git įrankiu, kiekvienas pakeitimų arba apjungimų užfiksavimas savyje laiko įrašo žinutę (šiame įrankyje ji yra privaloma). Tačiau atsiranda nemažai neaiškumų kokios struktūros ji turėtų būti. Moksliniame straipsnyje „CoreGen: Contextualized Code Representation Learning for Commit Message Generation“ [Misc9] rašoma, kad pakeitimų užfiksavimo žinutės yra programuotojų kuriama pakeitimų dokumentacija, kuri tūrėtų būti kuo abstraktesnė, tačiau aukštos kokybės žinučių rašymas yra sunki užduotis net ir patyrusiems programuotojams. Kadangi projektui augant ir užfiksavimų kiekiui didėjant, turi būti aišku, kas atsirado su pakeitimu. Geros pakeitimų užfiksavimo žinutės, taip pat, padeda greičiau išsiaiškinti, kada buvo padaryti pakeitimai, kuriuose atsirado klaidų ir jas ištaisyti. Viena iš galimų žinutės taisyklių yra pradžioje žinutės parašyti, kokia paskirtis šio pakeitimo, ar tai būtų pataisymas, ar pridėtas naujas funkcionalumas. Šių žinučių formato konvencijų yra gan daug ir visos jos priklauso nuo projekto ar komandos. Kadangi didelė dalis PĮ kuriama naudojantis projektų valdymo technikomis ir įrankiais, dažniausiai formuojamos užduotys projektų valdymo įrankiuose, todėl dažna praktika pranešime įterpti užduoties numerį ar identifikacinį numerį, kurio pagalba kiti programuotojai

galėtų rasti tą užduotį ir pamatyti, kokią problemą sprendžia šis pakeitimų užfiksavimas. Kiekvienas projektas turėtų nutarti ir aprašyti, kokiomis konvencijomis rašomos užfiksavimų žinutės ir koks jų formatas. Vienas iš pavyzdžių vadinamas „Git Karma“ užfiksavimo žinučių formatas.

2.2. Git eiga (anlg. *„Git flow“*)

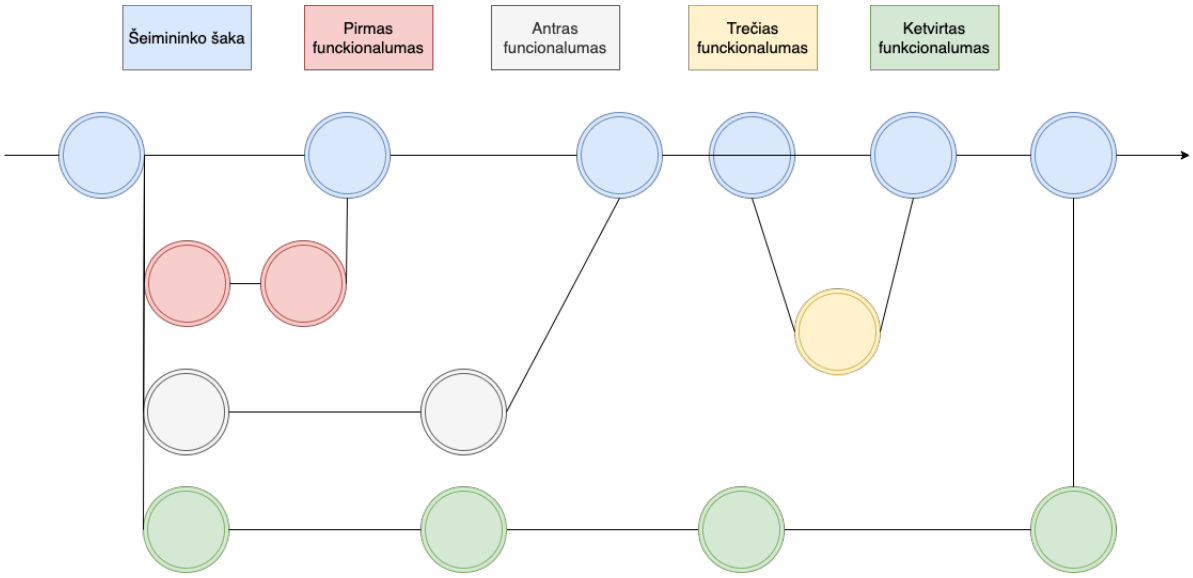
Git eiga pirma karta buvo paminėta Vincent Driessen jo straipsnyje „A successful Git branching model“ [Dri10]. Ši eiga paremta būdu, kur kiekvienas funkcionalumas turi savo atskirą šaką ir veliau ši šaka būna apjunginėjama su kitomis šakomis. Git eiga remiasi dviem pagrindinėmis šakomis. Viena pagrindinė šaka (angl. *„master“*), kurioje yra saugomi pakeitimai yra oficiali išleidimų (angl. *„releases“*) istorija ir saugo pakeitimus, kurie nukeliavo į produkcinę aplinką. Kita šaka yra kūrimo (angl. *„develop“*) šaka, kuri yra integracinė ir į ją sukeliami visi pabaigti funkcionalumai. Tai reiškia, kad kūrimo šaka savyje saugo visus projekto pakeitimus ir pilną istoriją. Naujo funkcionalumo kūrimo pradžioje programuotojas susikuria savo atšaką nuo kūrimo šakos. Atlikus visus pakeitimus ir juos užfiksavus ši funkcionalumo šaka būna sukeliama į serverį ir atiduodama peržiūrai. Po peržiūros šaka prijungiama atgal prie serveryje esančios kūrimo šakos. Taip dirba visi programuotojai su savo pataisymais ir naujais funkcionalumais. Kai tokių pakeitimų atsiranda daugiau ir pradedamas ruošti naujas išleidimas, sukuriama nauja išleidimo šaka nuo „kūrimo”. Paskutinė išleidimo šakos versija būna keliama į produkcinę aplinką ir po to prijungiama prie pagrindinės šakos, taip pat pagrindinė šaka prijungiama prie kūrimo. Skubus klaidų taisymas produkcinėje aplinkoje sprendžiamas kuriant naują šaką nuo pagrindinės. Tada pataisymai sukeliami į taisymo šaka ir galiausiai sukeliami į produkcinę aplinką. Tokiu būdu vėl prijungiami prie pagrindinės šakos ir taip pat prie kūrimo. Šią Git darbo eigą vaizduoja diagrama apačioje (2 pav.):



2 pav. Git eiga.

2.3. GitHub eiga

GitHub yra viena didžiausių programinio kodo saugojimo ir versijavimo platformų, todėl akivaizdu, kad šioje platformoje kodo versijavimo eigoms skiriamas didelis dėmesys. Vienas iš šios platformos įkūrėjų savo straipsnyje „GitHub Flow: the best way to use Git and Github“ [Cha13], kuriame aprašo šią eigą, teigia, kad ši eiga naudinga tuomet, kai norima kuo dažniau programinį kodą sukelti į produkcinę aplinką. Github eiga remiasi paprastu principu, kad turimas vienintelė ilgalaikė šaka - pagrindinė. Kiekvienam naujam funkcionalumui ar pataisymui kuriama nauja šaka, taigi taip pat kaip Git eigoje turime funkcionalumo šaką, tačiau šioje eigoje programuotojas užfiksavęs pakeitimus, savo lokalią šaką sukelią į serverį ir sukuria įtraukimo užklausą (angl. *„pull-request“*), kurią gali peržiūrėti kiti programuotojai dirbantys prie tos pačios kodo repozitorijos. Tokiu būdu pakeitimai būna taisomi, rašomi komentarai ir diskutuojama apie pakeitimų korektiškumą. Po to, kai užklausa būna patvirtinama, ši šaka prijungiama prie pagrindinės šakos ir pakeitimai diegiami į produkcinę aplinką. Autoriaus teigimu, ne visada pakeitimai turėtų iškart keliauti į produkcinę aplinką, bet įprastu atveju, tas yra rekomenduojama. Šios eigos trūkumas tas, kad turėjimas keletą skirtingų aplinkų, kuriose būtų galima dirbti testuotojams ir analitikams, yra gan sudėtinga, nes eiga akcentuota į greitą funkcionalumų kėlimą į produkcinę aplinką, bet ne kitas projekte naudojamas aplinkas. Taigi ši eiga yra gan paprasta ir reikalauja žymiai mažiau apjungimų nei Git eiga. Apačioje pavaizduotoje diagramoje parodomas Github eigos veikimas (3 pav.):



3 pav. Github eiga.

2.4. GitLab eiga

Kaip aprašyta kitos didelės atvirojo kodo platformos Gitlab, kurios vardu ir pavadinta ši eiga, GitLab eiga yra labai panaši į Github eigą tuo, kad nėra papildomų ilgalaikių šakų, kaip Git eigoje, o yra viena pagrindinė šaka. Skirtumas tarp Github ir GitLab eigų, kad kodas pagrindinėje šakoje ne visada atspindi, kas yra produkcinėje aplinkoje. Šioje eigoje pakeitimai apjungiami su pagrindine šaka, tačiau siekiant vykdyti diegimą, tam yra skirtos atskiros išleidimų šakos per aplinkas.

Tarkime programuotojas padarė pakeitimus, juos atidavė bendradarbių peržiūrai ir gavo patvirtinimą, kad jo pakeitimuose problemų nėra. Tada ši funkcionalumų šaka jungiama į pagridinę šaką ir ji apjungiama su aplinkos, į kurią norima diegti pakeitimus. Tai reiškia, kad kiekviena aplinka turi savo šaką ir joje atsiradus naujiems pakeitimų įrašams, jei būna diegiami į dedikuotą aplinką. Klaidų, kurios pamatomos toje aplinkoje, pataisymų šakos kuriamos, jau nebe nuo pagrindinės šakos, bet nuo tos aplinkos šakos ir išsprendus problemas vėl diegiama į tą pačią aplinką. Patikrinus PĮ funkcionalumų korektiškumą pakeitimai apjungiami su produkcinės aplinkos šaka ir ši šaka apjungiama atgal į pagrindinę šaką.

2.5. Kamienu paremtas (angl. ***„Trunk-based“***) programavimas su funcionalumų šakomis ir jungikliais (angl. ***„feature branches and flags“***)

Pasak šaltinio „Trunk Based Development: Introduction“ [Ham17] kamienu paremtas programavimas remiasi į tai, kad nėra jokių ilgo gyvavimo šakų, išskyrus pagrindinę kamieninę šaką. Šio versijavimo darbo eigos metu visi funkcionalumai apjungiami tik su pagrindine kamienine šaka, tačiau didelio masto projektams rekomenduojama vis tiek turėti savo atskiras funkcionalumų ir pataisymų šakas, tačiau jų egzistavimas turi būti kuo trumpesnis. Taigi, programuotojas savo lokalioje šakoje padaręs pakeitimus ir juos užfiksavęs sukelia į serverį, atiduoda kitiems komandos nariams peržiūrai. Po peržiūros pakeitimai apjungiami su pagrindine šaka ir funkcionalumo šakos ištrinamos. Kadangi, diegimai priešingai nei Github eigoje daromi ne iš karto, bet funkcionalumai išlieka kamieninėje kodo šakoje, ne visada norima šiuos funkcionalumus leisti naudoti vartotojams produkcinėje aplinkoje, todėl remiantis straipsniu „Software Development with Feature Toggles: Practices used by Practitioners“ [Rez20] naudoti funkcionalumų jungiklius, kurie konfigūracijoj indikuoja kad funkcionalumas yra išjungtas, yra vis populiarėjanti ir daug problemų išsprendžianti praktika. Norint produkcinėje aplinkoje matyti šiuos pakeitimus programuotojai gali pakeisti konfigūracijos nustatymus ir aktyvuoti funkcionalumus be jokių naujų kodo šakų apjungimų ar diegimų. Git įrankis palaiko tokį žymų sukūrimą. Tai reiškia, kad po bet kurio pakeitimų užfiksavimo galimą sukurti žymą kuri atspindėtų versijos numerį iki paskutinio (įskaitant paskutinį) užfiksavimo. Tokiu būdu kamienu paremtame programavime galima prieš kiekvieną diegimą sukurti žymą, kuri apibrėš diegiamą versiją ir tos žymos pakeitimus sudiegti į norimą aplinką (taip pat produkcinę). Ši kodo versijavimo kodo eiga užtikrina greitą ir efektyvų funkcionalumų tiekimą, be daug dažnu atveju nereikalingų šakų apjungimų, kurie dažnai sukelia apjungiamų šakų konfliktus. Šių konfliktų metu programuotojai turi nuspręsti, kurios besikertančių funkcionalumų eilutės yra teisingos ir tai sukelią didelę klaidų riziką. Minėto straipsnio autoriai įspėja, kad šios praktikos naudojimas prideda rizika laiku neištrinti jau išleistų funckionalumų.

3. Sistemos komponenčių artefaktų kūrimas naudojantis konteinerių technologija

3.1. Artefaktų kūrimas

Visa PĮ turi turėti vykdomuosius failus arba paketus, artefaktus norint ją vykdyti. Ne išimtis ir Java kalbos programos. Pačiame paprasčiausiame lygmenyje Java programinis kodas turi būti sukompiliuotas į mašininį kodą, kurį būtų galima vykdyti JVM aplinkoje. Kuriant kompleksiškesnes programas dažnai prireikia į programą įterpti išorines priklausomybes (angl. *„dependencies“*). Tokiu atveju neužtenką programinį kodą sukompiliuoti. Java kalbos atveju reikia sukurti programos paketą (angl. *„package“*), kuriame būtų įterpiamos bibliotekos ar kiti resursai, kuriuos galėtų pasiekti vykdoma programa. Tokie Java kalbos paketai, artefaktai vadinasi JAR (angl. *„Java archive“*). Java programų kontekste egzistuoja dar vienas paketų tipas - WAR (angl. *„Web application archive“*). Šis paketas skirtas serverinių programų paketavimui ir pagal jo struktūrą viduje turi būti JAR paketas ir kiti reikalingi failai, kurie įgalina programą veikti kaip internetinę programą. Šiai dienai egzistuoja daug įrankių padedančių automatizuotai vykdyti testus, kompiliuoti programinį kodą, prisidėti bibliotekas ir kurti artefaktus. Taip pat šie įrankiai dažniausiai palaiko įskiepių (angl. *„plugins“*) naudojimą, kurių pagalbą įrankio standartinius funkcionalumus galima praplėsti ir pridėti naujų. Įrankiai turi savo klientus ir komandas, kurių pagalba vykdomos artefaktų kūrimo operacijos. Knygoje „Continuous Delivery in Java“ [MB18] apžvelgiami keli populiariausi kūrimo įrankiai:

1. Apache Ant
2. Gradle
3. Apache Maven
4. Bazel, Buck, Pants it t.t.

Norint vykdyti artefaktų kūrimą su bet kuriuo iš šių įrankių reikalingi konfigūraciniai failai projekto repozitorijos viduje. Įrankių konfigūracinių failų formatas ir aprašymo stilius skiriasi. Pavyzdžiui, Apache Maven ir Apache Ant konfigūracijos apsirašomos XML formato failuose, o Gradle įrankis palaiko Groovy arba Kotlin kalbų sintaksę, naudojamos kalbos yra JVM programavimo kalbos. Jau minėtoje knygoje „Continuous Delivery in Java“ [MB18] padaryta įrankių apžvalga išskiria tokias išvadas:

1. Apache Ant naudojimas nerekomenduojamas, pradedant naują projektą, kuriame naudojami modernūs Java kalbos karkasai, pvz.: Dropwizard, Spring Boot.
2. Su Apache Maven įrankiu komplikuota kurti specifinius įskiepius bei sunku naviguoti po konfigūracinius failus, kuriuose nurodoma daug išorinių bibliotekų, tačiau patogus įrankis komandoms, kurios turi patirties ir aukštą kvalifikaciją naudojantis šiuo įrankiu.
3. Gradle įrankiu nesunku apsirašyti specifinį artefaktų kūrimo procesą ir savo įskiepius. Patogu naudoti Groovy kalba rašomus testavimo karkasus, pvz.: Spock, Geb. Taip pat gerai pritaikytas įrankis naudojant mikroservisų architektūriniam stiliui pritaikytus karkasus, pvz.: Spring Boot. Taip pat šis įrankis palaiko specifinių konfigūracijos šablonų naudojimą, kuris yra naudingas turint mikroservisų stiliaus sistemą.

4. Bazel, Buck, Pants patogūs įrankiai naudojant dideles vienos repozitorijos tipo programas, tačiau nėra jokių privalumų kuriant tradicinę mikroservisų stiliaus sistemą su nedideliu kiekiu programų.

3.2. Konteinerių atvaizdų kūrimas

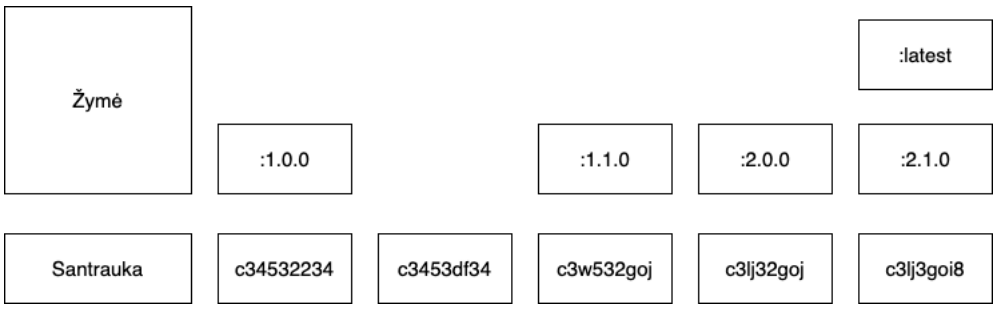
Remiantis straipsniu „Automatic Observability for Dockerized Java Applications“ [Misc11], paviešintu 2021 metais, konteinerių technologijos panaudojimas Java programų kūrime tapo viena dažniausiai naudojamų praktikų debesų kompiuterija paremtose Java sistemose. Ši praktika suteikia galimybę turėti programą izoliuotoje aplinkoje, konteineryje, kurį galima būtų diegti į bet kokią kitą aplinką, kurioje sudiegtas konteinerių valdymo įrankis. Tokiu būdu galime turėti sistemą, kurioje egzistuotų daug skirtingų technologijų ir mikroservisų stiliaus programų, tačiau nereikėtų kiekvienos technologijos diegti į serverį, kuriuo naudojasi sistemos vartotojai. Plačiausiai naudojamas konteinerių valdymo įrankis remiantis straipsniu „Container-based Cluster Orchestration Systems:A Taxonomy and Future Directions“ [Mar18] yra atvirojo kodo įrankis Docker. Siekiant konteinerizuoti Java programą pasinaudojant Docker reikalingas failas pavadinimu Dockerfile, kurio pagalba aprašoma instrukcija kaip kurti Docker atvaizdą (angl. *„image“*). Atspindys susideda iš programos artefakto ir aplinkos konfigūracijos, kurios reikia, kad ši programa galėtų būti vykdoma. Java artefaktų kontekste pagrindinis reikalavimas aplinkai yra sudiegtas JDK (angl. *„Java Development Kit“*) arba JRE (angl. *„Java Runtime Environment“*). Taip pat Dockerfile nurodomas ir tinklo prievadas (angl. *„port“*), kuriuo galima pasiekti ir vykdyti užklausas į programą. Docker atvaizdo kūrimą taip pat galima automatizuoti kūrimo įrankio įskiepio pagalba, pvz.: Gradle įrankis turi įskiepį, kurį susikonfigūravus yra galimybė inicijuoti atskirą užduotį ir sukurti naują Docker atvaizdą.

3.3. Konteinerių atvaizdų versijavimas

Sukurtam atvaizdui Docker įrankis pritaiko santraukos funkcija (angl. *„digest function“*) ir taip sugeneruojamas atvaizdo identifikatorius. Taip pat atvaizdui galima suteikti žymę. Žymos nepriskyrus laikoma, kad atvaizdas jos neturi. Ši žymė tampa Docker atvaizdo versija. Žymos priskyrimui ir atvaizdų versijavimui remiantis rastomis rekomendacijomis, pavyzdžiui, straipsniu „Docker Tagging: Best practices for tagging and versioning docker images“ [Las18]. Egzistuoja dvi dažnai naudojamos praktikos:

1. Semantinis versijavimas.
2. Unikalios žymės kūrimas.

Naudojant semantinio versijavimo praktiką atvaizdui yra suteikiama stabilios versijos numeris (plačiau apie semantinį versijavimą kitame skyriuje). Jeigu sukurtas atvaizdas pasirodo, kad yra nestabilus, tada ištaisius kilusias problemas priskiriamas tas pats numeris, tačiau su sekančiais pakeitimais arba programinio kodo pataisymais versijos numeris būna pakeliamas. Taip pat kiekviena Docker repozitorija palaiko naujausios (angl. *„latest“*) versijos žymę. Ši žymė visada priskiriama aukščiausiam stabilios versijos numeriu pažymėtam atvaizdui. Toks versijavimo pavyzdys pavaizduotas apačioje (4 pav.):



4 pav. Docker atvaizdų semantinis versijavimas.

Pasak autoriaus semantinis žymės versijavimas nėra geriausia praktika, nes versijos numeriai skirtingu metu gali būti žymimi prie skirtingų atvaizdų ir tai yra nestabilu. Kitas dažnai naudojamas žymių kūrimo ir priskyrimo būdas yra unikalios žymės priskyrimas. Problema su santraukos naudojimu kaip versijos numeriu yra ta, kad santraukos numeris sukuriamas automatiškai nesuteikia jokios atsekamos informacijos apie tai, kokia PĮ versija yra atvaizdo viduje. Toliau pateikiama, kokios dažnai naudojamos unikalios reikšmės būna priskiriamos žymei kartu su komentarais apie jas:

1. ***Git užfiksavimo identifikacinis numeris***. Šis identifikacinis numeris yra geras tol, kol kiekvienas atvaizdas yra naujas užfiksavimas, tačiau kartais reikia keisti atvaizdo aplinkos konfigūracija. Jeigu konfigūracija yra atskirai nuo kodo, tai ji nebūtinai turi savo atskira užfiksavimą.
2. ***Git apibūdinančios komandos reikšmė***. Ši reiškmė gaunama iškviečiant Git įrankio komandą „describe“. Iškvietus šią komandą gaunamas rezultatas sudaromas iš paskutinės versijų istorijos žymos ir nuo žymos sukūrimo velėsnių užfiksavimų skaičių, bei paskutinio užfiksavimo identifikacinį numerį. Jeigu paskutinis užfiksavimas yra naujos žymos sukūrimas, tai reiškmę sudaro tik žyma.
3. ***Datos ir laiko antspaudas***. Šis būdas yra informatyvus. Aišku, kada buvo sukurtas atvaizdas, tačiau tai niekaip nekorealiuoja su informacija apie tai, kokia PĮ versija yra šiame atvaizde.
4. ***Artefakto unikalus numeris***. Šis unikalus numeris yra informatyvus apie PĮ versiją atvaizde ir yra unikalus.
5. ***Artefakto sistemos ir artefakto unikalus numeris***. Šis būdas skirtas tada, kai egzistuoja keletas sistemų, kuriuose gali būti kuriami artefaktai.

3.4. Konteinerių atvaizdų saugojimas

Sukūrus Docker atvaizdą ir jį pažymėjus, atvaizdas iš lokalios mašinos gali būti sukeliamas į nuotolinį atvaizdų serverį. Tai gali būti privatus serveris arba debesų kompiuterija paremta platforma, kurioje saugomi projekto atvaizdai. Iš nuotolinio serverio kiti programuotojai savo lokalioje aplinkoje ar dedikuotoje sistemos aplinkoje, pavyzdžiui produkcinėje, gali parsiųsti ir vykdyti programą nurodę konkrečią versijos žymę.

4. Sistemos versijavimas ir konfiguracijų valdymas

4.1. Versijavimo praktikos

Mikroservisų stiliaus architektūrose dažnai atsiranda funkcionalumų, kurie kuriami pasinaudojant ne tik vienu mikroservisu. Tokiais atvejais dažnai kuriamos atskiros bibliotekos bendroms duomenų struktūroms, modeliams aprašyti ar kitu būdu sprendžiama bendrų struktūrų problema. Kartais atsiranda poreikis, kad išorinės sistemos naudotų sukurtus funkcionalumus. Taip pat, prisimenant vieną iš pagrindinių mikroservisų stiliaus architektūrų privalumų - perpanaudojamumą, programa gali būti naudojama keleto skirtingų sistemų ir darant pakeitimus servise, kitos sistemos apie juos nežino. Tokiais atvejais atsiranda poreikis visus servisus versijuoti. Versijavimas greitai padeda suprasti, kad servise atsirado pakeitimų ir naudojama ne paskutinė jo versija arba atvirksčiai, pasitikrinti ar naudojamas servisas yra naujausios versijos. Yra daug versijavimo praktikų, tačiau remiantis Matthew Setter straipsniu „Best Practices When Versioning a Release“ [Pat20] ir knyga „Continuous Delivery in Java“ [MB18] šiuo metu industrijos geroji praktika ir versijavimo standartas jau yra tapęs Semantinis versijavimas, todėl detaliau pakalbėsime būtent apie jį.

Remiantis šaltiniu „Semantic Versioning 2.0.0“ [Pre] Semantinis (angl. *„Semantic“*) versijavimas, dar vadinamas „Semver“, apibrėžia versiją pagal tris pagrindinius veiksnius „x.y.z“, kur:

1. X - stabili verija, kuri keičiasi atsiradus nesuderinamiems pakeitimams (angl. *„breaking changes“*)
2. Y - nauji funkcionalumai
3. Z - pataisymai

Šie trys faktoriai apibrėžia ar kokie pakeitimai atsirado su nauja versija. Kuriant naujus pakeitimus šių faktorių reikšmės išreikštos skaičiaus pavidalu keliamos per vienetą. Kartais paskutinis faktorius apie pataisymus yra praleidžiamas. Tokio versijavimo naujausios versijos pavyzdys šio darbo rašymo metu vienoje iš populiariausių interneto naršyklių „Mozilla Firefox” yra 88.0. Tai reiškia, kad dabartinė versija, kurios ne visa programavimo sąsaja (angl. *„Application Programming Interface“* arba API) palaiko tai, ką palaikė senesnes versijos yra 88, tačiau antras indikatorius 0 reiškia, kad nuo naujos sąsajos išleidimo nėra naujų funkcionalumų ir taip pat pataisymų, nes praleistas pataisymų indikatorius. Diegiant tik pataisymus „Firefox“ naršyklės atveju nauja versija atrodytu taip - 88.0.1. Kadangi pataisymų versijos skiltis apibrėžia kiek jų buvo padaryta naujausiai pakeitimų skilčiai, pakėlus pakeitimų skirties versijos numerį pataisymų versija tampa lygi 0. Tokiu pačiu principu naujų funkcionalumų versija apibrėžia pakeitimus atliktus stabiliai versijai, dėl to pakėlus stabilios versijos reikšmę, funkcionalumų ir pataisymų reikšmės tampa lygios 0 ir sekanti „Firefox“ su ankstesnėmis nesuderinama versija atrodys taip - 89.0.

4.2. Konfiguracijų valdymas

Mikroservisų programų valdymas yra svarbi problema nuolatinio tiekimo kontekste. Turint keletą skirtingų aplinkų atsiranda poreikis turėti skirtingus programų nustatymus. Vienas iš pavyzdžių būtų duomenų šaltinis. Turėti bendrą duomenų šaltinį produkcinei ir testinei aplinkai

nepatartina praktikų, todėl prisijungimų prie šaltinio konfigūracijos skirsis tarp aplinkų. Kitas pavyzdys būtų funkcionalumų jungiklių nustatymai (angl. *„feature flags“*), apie kuriuos kalbėjome PĮ kodo versijavimo skyrelyje. Kartais norime, kad programinis kodas įgalinantis naujus funkcionalumus atsidurtų produkcinėje aplinkoje, tačiau vartotojai to funkcionalumo nematytų tol, kol suprogramuotas funkcionalumas iki galo neištestuotas arba jis būtų pasiekiamas tik testinėse aplinkose. Tokiu atveju reikalingos skirtingos konfigūracijos tarp aplinkų.

Pagal knygą knyga „Continuous Delivery in Java“ [MB18] yra du pagrindiniai būdai kaip valdyti konfigūracijas:

1. Vidinės servisų konfiguracijos.
2. Išorinės servisų konfiguracijos.

Modernūs Java kalbos programų karkasai, tokie kaip „Spring Boot”, palaiko keletą skirtingų konfigūracijų. „Spring Boot“ karkaso atveju yra tokia sąvoka kaip profiliai. Programos konfigūracinis failas yra vienas pagrindinis, tačiau skirtingų profilių konfigūraciniai failai savyje turi kitas konfigūracijos reikšmes. Serviso paleidimo metu kaip sisteminis kintamasis nurodomas profilis ir tokio tipo profilio konfigūracijos reikšmės užrašomos ant viršaus numatytųjų. Tokiu būdu skirtingose aplinkose nurodomi profiliai, pvz.: „test“, „prod“, su skirtingomis konfigūracijomis. Renkantis tokį būdą valdyti konfigūracijas, jos laikomos kartu su bendra kodo baze ir versijuojami, taip pat kaip kodas.

Vienas pagrindinių mikroservisų stiliaus programų privalumų yra jų perpanaudojamumas [Oma18]. Tai reiškia, kad vieną programą galime turėti keliose skirtingose sistemose ir tokiu atveju konfigūraciją laikyti kartu su programiniu kodu nebūtų geras sprendimas, nes keičiant konfigūraciją potencialiai bus daromi pakeitimai ir kitoms sistemoms. Taip pat, vidinių konfigūracijų saugumo iššūkiu tampa slaptažodžių konfigūracijų valdymas, nes prie kodo bazės laikyti produkcinės aplinkos slaptažodžius nėra saugu.

Kitas būdas valdyti konfigūracijas yra jas laikyti atskirai nuo PĮ kodo. Šis būdas tampa pravartus sistemoje naudojant daug komponenčių, mikroservisų. Tokiu atveju visa servisų konfigūracija būtų laikoma vienoje vietoje. Taip pat, kartais keli servisai turi bendrus konfigūracijos nustatymus, pavyzdžiui duomenų šaltinio nustatymus. Juos būtų galima valdyti vienoje vietoje, tai reiškia vienu kartu keičiant sistemos nustatymus, tie pakeitimai įvyktų visuose šį nustatymą naudojančiuose servisuose. Taip pat, informacinėse sistemose gali atsirastai poreikis konfigūracijas valdyti ne programuotojams, o komandos, atsakingos už sistemų infrastruktūrą, nariams. Tokiu atveju rekomenduojama turėti atskirą repozitoriją konfigūracijoms. Atsiradus poreikiui keisti konfigūraciją, pavyzdžiui, funcionalumų veliavų nustatymus analitikams ar kitiems komandos nariams, kurie negali arba nemoka prieiti prie kodo, dirbti su kodo versijų kontrolės įrankiais, siūloma susikurti atskiras programas arba naudotis jau sukurtais sprendimais, kurie tūrėtų grafinę sąsają, būtų patogūs naudoti ir naudojami, ne kaip atskiras mikroservisas sistemoje, bet kaip paslauga. Tokių egzistuojančių sprendimų yra nemažai, pavyzdžiui:

1. „ConfigCat“.
2. „Launch Darkly“.
3. „Unleash Hosted“.

5. Nuolatinė integracija ir automatizavimas

Nuolatinė integracija (angl. *„Continuous Integration“* arba trumpiau *„CI“*), kaip minėta pirmame skyrelyje yra praktika, kurios esmė skirtingų PĮ kodo versijų apjungimas ir integravimas į pagrindinę versijų šaką. Taip pat į šią praktiką įeiną ir kodo versijos testavimas ir artefaktų kūrimas.

Automatizuotas artefaktų kūrimas yra viena iš pagrindinių nuolatinės integracijos atsakomybių. Martin Fowler straipsnyje pavadinimu „Continuous Integration“ [Fow06], pabrėžia, kad vienas iš nuolatinės integracijos reikalavimų yra jos efektyvumas. Automatizuotas kodo versijų apjungimas yra integravimas turi būti greitas, nes kitaip ši praktika netenka prasmės. Nuolatinės integracijos įgyvendinimas yra grandinė veiksmų, kuriuos visus sėkmingai įvykdžius laikoma, kad grandinė konkrečiai versijai yra sėkminga.

Kaip teigiama knygoje „Continuous Delivery in Java“ [MB18], šiais laikais standartinė praktika tapo nuolatinės integracijos realizavimas pasinaudojant egzistuojančiais įrankiais, kurie veikia kaip atskiras serveris arba beserverinė paslauga. Šis serveris būtų atsakingas už versijų tarpusavio integravimas, kodo kokybės verifikavimą pagal nustatytus žingsnius, pvz.: automatizuotų testų vykdymą ar statinės analizės įrankių ataskaitą. Taip pat, šie įrankiai dažnai atlieka ir versijos diegimo į aplinkas funkciją, todėl tokie įrankiai dažnai vadinami ne tik nuolatinio tiekimo, bet „nuolatinės integracijos/nuolatinio tiekimo“ įrankiai (angl. *„CI/CD tools“*). Siekiant naudotis vienu įrankiu tiekimui ir integravimui toliau nagrinėsime tik tokio tipo įrankius.

5.1. Nuolatinės integracijos ir tiekimo įrankiai

Visus nuolatinės integracijos ir įrankius būtų galima suskirstyti į dvi grupes:

1. Serveriniai įrankiai.
2. Paslaugų įrankiai.

Plačiau apibrėžiant, komerciniai serveriniai įrankiai arba atvirojo kodo (angl. *„open-source“*) gali būti diegiami lokaliame serveryje ir valdyti resursų išskyrimą įrankio serveriui. Paslaugų įrankiai dažniau būna komerciniai ir jų serveriai kaip paslauga parduodami vartotojams. Tai reiškia, kad su tokio tipo įrankiais nereikia valdyti resursų, skirtų serveriui. Taip pat, egzistuoja įrankių kuriuos galima naudoti ir kaip serverius ir kaip paslaugas. Straipsnyje „Continuous Integration on Cloud Versus on Premise: A Review of Integration Tools“ [Mak20] teigiama, kad pasirinkimas tarp šių skirtingų įrankių gali priklausyti nuo organizacijos numatytų kaštų dydžio, saugumo reikalavimų, lankstumo ir pasitikėjimo paslaugų tiekėjais. Toliau pateikiami dažniausiai naudojamų įrankių sąrašas remiantis įmonės ActiveState apklausos „State Of CI/CD 2020 Survey Results“ [Act20] rezultatais:

1. Jenkins.
2. AWS CodeBuild.
3. Microsoft Visual Studio Team Services.
4. GitLab CI.
5. BitBucket.

6. Azure Pipelines.
7. GitHub Actions.
8. Circle CI.

Kiekvienas įrankis turi skirtingų funkcionalumų ir privalumų, trūkumų, tačiau visi gali padėti realizuoti pagrindines nuolatinės integracijos praktikas. Pasak knygos „Continuous Delivery in Java“ [MB18] autorių renkantis tarp šių įrankių svarbiausia atsižvelgti į norimus rezultatus. Šie įrankiai, taip pat, pateikia žingsnių vykdymo laiko matus, bei vykdymo žinynus (angl. *„logs“*).

Įrankiuose nurodomos projektų konfigūracijos, todėl dažnai reikalaujama prie projekto programinio kodo laikyti konfigūracinius failus. Šiose konfigūracijose aprašomi žingsniai, kurie bus vykdomi integracijos ar diegimo metu, bei interaktyviai arba deklaratyviai galima pasirinkti, ar vykdyti sukonfigūruotus žingsnius.

5.2. Statinės kodo analizės įrankiai

Pakeitimų programinio kodo peržiūra yra svarbi praktika siekiant užtikrinti kodo kokybę. Siekiant užtikrinti kuo paprastesnį PĮ palaikymą ilgalaikiu laikotarpiu labai svarbu užtikrinti, kad projekte bus naudojama kuo mažiau skirtingų kodo rašymo konvencijų. Dėl šios priežasties projektų dokumentacijose dažnai aprašomos konvencijos ir praktikos, naudojamos tame projekte. Siekiant pagreitinti kodo peržiūras ir užtikrinti kodo skaitomumą galima pasitelkti statinės kodo analizės įrankius. Šių įrankių pagalba projekto programinis kodas būna analizuojamas ir tikrinama, ar nėra sukonfigūruotas kodo rašymo taisykles pažeidžiančių vietų. Taip pat, šių įrankių naudojimas gali padėti ankstyvoje stadijoje surasti potencialias klaidas ir jas ištaisyti dar prieš diegiant pakeitimus į vartotojams skirtas aplinkas. Knygoje „Continuous Delivery in Java“ [MB18] pateikiami keli pavyzdžiai tokių įrankių:

1. *PMD*.
2. *CheckStyle*.
3. *FindBugs*.

Egzistuoja dar vienas plačiai naudojamas įrankis „SonarQube“, kurio paskirtis tikrinti PĮ kodo kokybe ir saugumą. Šio įrankio pagalba galima gauti matus nusakančius įrankyje apibrėžtų taisyklių pažeidimus. Šiam įrankiui reikalingas serveris, kurio pagalba atliekama statinė kodo analizė. Siekiant nekurti naujo serverio dedikuoto kodo analizei, egzistuoja ir šio įrankio, kaip paslaugos alternatyva pavadinimu „SonarCloud“. Šio įrankio, kaip paslaugos naudojimas prideda papildomo darbo programuotojams, nes norint atlikti analize privalu, kad kodas būtų vykdomas per nuolatinės integracijos grandinę. Siekiant to išvengti ir rasti klaidas dirbant lokaliai, galima į savo integruotoje kūrimo aplinkoje (angl. *„IDE“*) įsirašyti „SonarLint“ įskiepį. SonarLint palaiko plačiai naudojamas kūrimo aplinkas, pavyzdžiui, „InteliJ IDEA“, „Visual Studio“, „Eclipse“ ir t.t. Šis įskiepis pagal tas pačias taisykles, kaip SonarCloud tikrintų kodą, prieš jam atsiduriant nuolatinės integracijos grandinėje. Tas leistų greičiau pamatyti klaidas ir jas ištaisyti.

6. Automatizuotas PĮ testavimas nuolatinėse integracijose

Vienas iš svarbiausių nuolatinės integracijos dalių yra automatizuotų testų vykdymas. Remiantis knyga „Continuous Delivery“ [HF10] automatizuotų testų vykdymas yra viena iš prielaidų, būtinų nuolatiniam tiekimui įgyvendinti. Testavimas yra vienas iš būdų užtikrinti, kad PĮ tiekiama vartotojams atitinka reikalavimus ir veikia numatytai. Taip pat, automatizuoti testai ne tik gali programuotojams padėti įsitikinti, ar jų kodas veikia taip, kaip numatyta, bet ir padėti ateityje kuo greičiau įspėti apie senų, veikiančių funkcionalumų sugadinimą. Svarbu nepamiršti Edsger W. Dijkstra žodžių, pateiktų jo darbe „Notes On Structured Programming“ [Dij70], kad programų testavimas gali parodyti klaidų buvimą, bet niekada neparodys jų nebuvimo.

Visus sistemų testavimus galime išskirti į dvi grupes:

1. Nefunkcinių reikalavimų testavimas. Skirtas testuoti tokiųs PĮ kodo reikalavimus, kaip tvarkingumą, sistemos efektyvumą ir atsparumą.
2. Funkcinių reikalavimų testavimas. Skirtas testuoti verslo logikos įgyvendinimą sistemoje.

6.1. Nefunkcinių reikalavimų testavimas

Nefunkcinių reikalavimų testavimai, kaip teigiama knygoje „Continuous Delivery in Java“ [MB18], yra dažnai praleidžiamas žingsnis PĮ tiekimo grandinėse, ypač mažose komandose su nedideliais ištekliais. Įmonės dažnai nėra linkusios investuoti laiko į nefunkcinių reikalavimų testavimą, nes tai neatneša trumpuoju laikotarpiu daug pridėtinės vertės jų klientams, nebent to reikalauja dalykinė sritis, pavyzdžiui, bankininkystė. Šiai dienai egzistuoja daug įrankių ir praktikų, kaip testuoti nefunkcinius reikalavimus. Toliau pateiksime keletą tokių testavimo įrankių pavyzdžių naudojamų Java projektuose:

1. *„Checkstyle“* arba *„PMD“*. Šie įrankiai buvo aprašyti statinės analizės skyrelyje ir apibrėžti jų naudojimo būdai.
2. „ArchUnit“. Šios bibliotekos pagalba rašomi testi skirti mikroserviso architektūros kokybės validavimui. Su juo galima testuoti priklausomybes tarp klasių ar nėra ciklinių priklausomybių ir t.t. Įrankio pagalba testai rašomi šalia PĮ kodo ir vykdomi kūrimo (angl. *„build“*) įrankio pagalba.
3. *„Gatling“*. Tai yra atvirojo kodo įrankis skirtas testuoti, kaip sistema susidoros su dideliu kiekiu užklausų. Šio įrankio pagalba galima apsirašyti testavimo scenarijus su JVM kalbų grupės kalba Scala. Gatling įrankio biblioteka turi savo testų scenarijų apsirašymo programavimo sąsaja, bet norint vykdyti testus reikalinga, kad programa būtų vykdoma. Tai reiškia, kad testavimo scenarijai su šiuo įrankiu turi būti paleidžiami po to, kai programa sudiegta į aplinką. Taip pat, testai vykdomi su atskirai nuo kitų, tačiau yra galimybė juos vykdyti pasinaudojus kūrimo (angl. *„build“*) įrankio pagalba. Po testų scenarijų vykdymo įrankis gali sugeneruoti ataskaitą, kokie tarpiniai rezultatai buvo gauti vykdant testus.
4. *„SpotBugs“*. Šis įrankis taip pat buvo aprašytas statinės kodo analizės skyrelyje. Jo pagalba galima skenuoti PĮ kodą ir tokiu būdu testuoti ar programoje nėra potencialių saugumo spragų. Įrankis turi didele apsirašytų taisyklių aibę, apie kurių pažeidimus įrankis sugene-

ruoja HTML ataskaitą. Šis įrankis gali būti konfigūruojamas kūrimo (angl. *„build“*) įrankio konfigūracijoje.

Nefunkcinių reikalavimų testavimo kiekis priklauso nuo projekto apimties, vartotojų ir dalykinės sistemos srities, todėl sunku apibrėžti pakankamą testavimo praktikų kiekį.

6.2. Funkcinių reikalavimų testavimas

Egzistuoja daug skirtingų funkcinių reikalavimo būdų testavimo būdų ir daug skirtingų testų tipų, tačiau jų visų aptarimas ir gero testų kiekio apibrėžimas projekte nėra šio darbo kontekste. Čia aptarsime tris funkcinių reikalavimų testų tipus, kurie remiantis anksčiau minėta knyga visada turėtų būti vykdomi nuolatinės integracijos grandinėje:

1. Vienetų testai (angl. *„Unit tests“*).
2. Komponentų testai (angl. „Component tests“).
3. Priėmimo testai (angl. *„Acceptance tests“*).

Vienetų testai yra rašomi tam, kad patikrintų mažų programos dalių veikimą izoliuotoje aplinkoje, pavyzdžiui vienos funkcijos ar kelių funkcijų sąveikos. Java kalbos kontekste, tai dažniausiai būna testavimas vienos klasės viduje. Tokio tipo testų vykdymas nereikalauja visos programos paleidimo. Taip pat, jie neturėtų naudoti duomenų bazės, failų sistemos ar interneto tinklo. Duomenys, skirti tokiems testams, nebūtinai yra tokie patys, kokie produkcinėje aplinkoje. Labai svarbu kad šio tipo testai būtų įvykdomi per trumpą laiko tarpą, nes tokio tipo testų, remiantis knygos „Continuous Delivery in Java“ [MB18] autorių nuomone, turi būtų daugiausia ir tai yra viso testavimo pagrindas.

Komponentų testų paskirtis yra patikrinti keleto programos komponentų veikimą kartu. Kaip ir vienetų testai, šio tipo testavimas ne visada reikalauja visos programos paleidimo, tačiau jie gali naudoti tokius resursus kaip duomenų bazė, failų sistema ar interneto tinklas. Resursų naudojimas testavimo metu turi užtikrinti šalutinių poveikių nebuvimą, tai reiškia, kad po jų įvykdymo negali likti pakeitimų realiuose resursuose.

Priėmimo testavimo metu yra tikrinama, ar PĮ kodas užtikrina verslo taisyklių įvykdymą programoje. Tokio tipo testavimas tikrina programos funkcionalumus. Priėmimo testavimai reikalauja, kad programa būtų paleista ir gyvuotų aplinkoje panašioje į produkcinę aplinką. Dėl didelio resursų naudojimo, šių testų vykdymas gali užtrukti ilgai.

7. Diegimai ir jų valdymas

Diegimo metu į dedikuotą sistemos aplinką yra sukeliami nauji programinės įrangos paketai, Docker atveju atvaizdai, iš kurių kuriami konteineriai.

Ankstesniuose skyriuose kalbėjome kaip iš Java paketų galima sukurti Docker konteinerių atvaizdus ir juos sukelti į atvaizdų serverius, todėl toliau kalbėsime apie atvaizdų diegimą į aplinkas.

Konteinerių technologija įgalina konteinerių sukūrimą iš atvaizdų po vieną, tačiau mikroservisų architektūrinio stiliaus sistemose konteinerių tikriausiai bus daugiau nei vienas ir taps sudėtinga konteinerius grupuoti, valdyti tinklo nustatymus ir pasiekiamumą tarp jų bei bendrus resursus. Šiai problemai spręsti reikalingi orkestratoriai, kurių pagalba egzistuotų bendras aplinkos konfigūravimas, bei atskirai būtų galima konfigūruoti kiekvieną mikroservisą. Taip pat, nuolatinis tiekimas turėtų įgalinti sistemos atnaujinimus vykdyti dažnai, pavyzdžiui, kelis kartus per dieną, o tam reikalingas toks diegimo būdas, su kuriuo prastovos (angl. *„downtime“*) trukmė būtų lygi nuliui, kad sistemos vartotojai galėtų pasiekti sistemą bet kuriuo metu. Tam egzistuoja diegimų valdymo strategijos.

7.1. Applikacijos sveikatos sąsaja

Prieš pradedant nagrinėti diegimo strategijas svarbu paminėti, kad egzistuoja mikroservisų sveikatos patikrinimai (angl. *„health checks“*). Remiantis knyga „Continuous Delivery in Java“ [MB18] prieš atsirandant nuolatiniam tiekimui ir vykdant diegimus rankiniu būdu patikrinimas, ar diegimo procesas pavyko sėkmingai, būdavo rankinis procesas. Atsiradus automatizuotiems diegimams atsirado ir automatinio patikrinimo, ar mikroservisas veikia, poreikis. Atsakymą į klausimą, ar mikroservisas veikia, galima gauti turint sveikatos patikrinimo sąsaja. Šios sąsajos principas, kad ji nevykdo jokių pašalinių veiksmų, tik grąžina teigiamą atsakymą, kai programa pilnai pasileidusi. Daugelis Java kalbos karkasų, pavyzdžiui, Spring Boot, šią sąsają sukuria ir atskirai kurtis naujos nereikia. Konteinerių orkestratorius, naudodamas šią sąsają, per sukonfigūruotą laiką tikrina ar sistema yra stabili. Atsitikus incidentams su konteineriais, juos orkestratorius iš naujo paleidžia arba grąžina senesnes versijas.

7.2. Diegimų valdymo strategijos

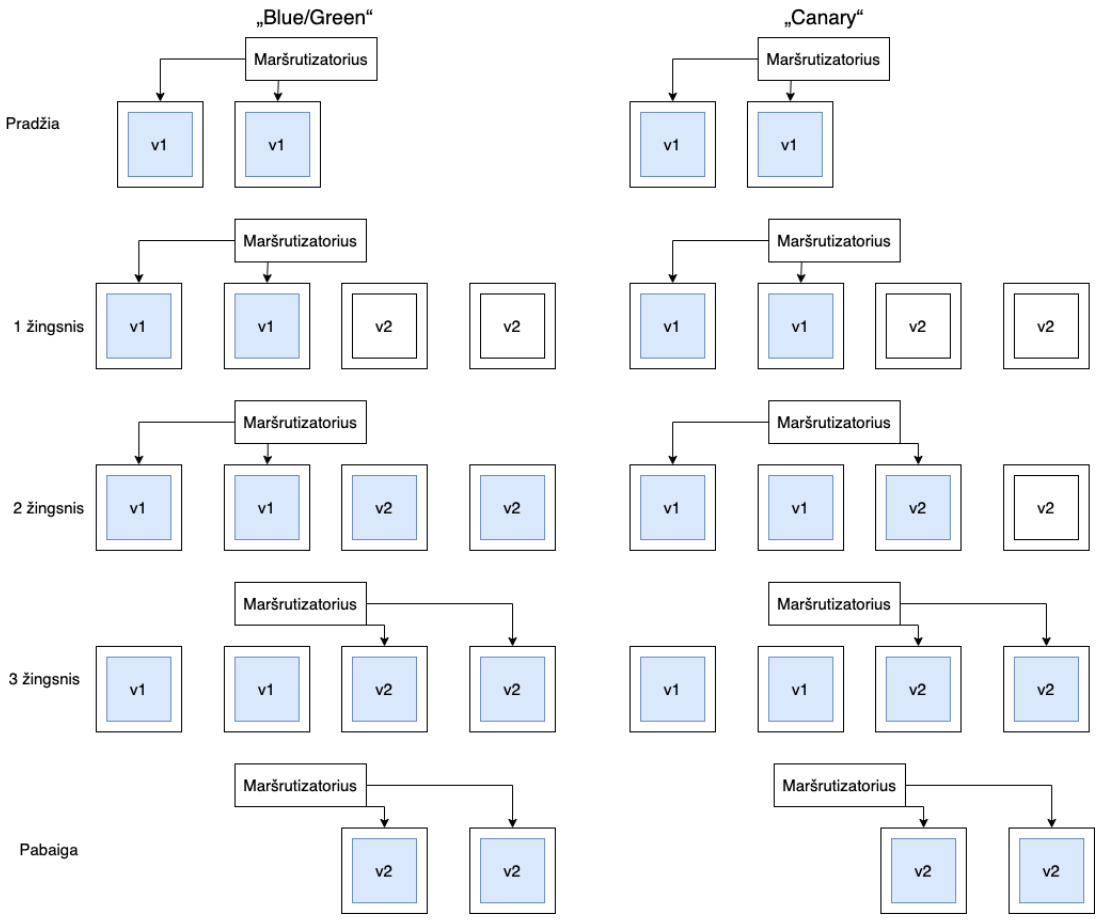
Šiame darbe siekiant užtikrinti, kuo aukštesnį brandos lygį nagrinėsime tik tokias diegimų strategijas, kurių pagalba prastovos laikotarpis diegimo metu būtų lygus nuliui.

Konteinerių technologijos pagalba mikroservisų architektūrinio stiliaus sistemose yra galimybė aplinkoje esančius mikroservisus leisti per kelis konteinerius. Tai reiškia, kad aplinkoje egzistuoja keli konteineriai, kuriuose vykdoma ta pati programa ir užklausų padalijimu tarp konteineriu rūpinasi apkrovos skirstytojas (angl. *„load balancer“*). Šiame darbe tirsime tik tokias praktikas, kuriose diegimai vyksta be senų mikroservisų versijų ir naujų buvimo kartu aplinkoje, nes dėl to gali atsirasti nenumatytų sistemos veikimo sutrikimų tarp skirtingų nesuderinamų

versijų. Knygoje „Continuous Delivery“ [HF10] yra pateikta dvi diegimo strategijos be prastovos laikotarpio.

1. *„Blue/Green Deployment“*.
2. *„Canary Deployment“*.

Pagrindinis skirtumas tarp šių strategijų yra skirtingų versijų diegimo metu naudojimas. „Blue/Green Deployment“ strategijos metu nauja versija pradedama naudoti tik tada, kai visų naujos versijos konteinerių sveikatos sąsajos grąžina teigiamą rezultata ir konteineriai yra paruošti naudoti. Sulaukus teigiamų atsakymų iš sveikatos sąsajų, visass senos mikroserviso versijos maršrutizavimas perkeliamas nuo senos versijos konteinerių ant naujų. Galiausiai seni konteineriai ištrinami iš aplinkos. „Canary Deployment“ strategijos atveju naujos versijos konteineriai pradedami naudoti palaipsniui. Nauji konteineriai paleidžiami ir po vieną konteinerį maršrutizavimas perduodamas bei ištrinamas senos versijos konteineris. Taigi, „Blue/Green Deployment“ strategija užtikrina senų ir naujų versijų konteinerių nevykdymą lygiagrečiai. Toliau pavaizduotas šių dviejų strategijų veikimas (5 pav.):



5 pav. „Blue/Green“ ir „Canary“ diegimų diagrama.

7.3. Diegimo orkestratoriai

Šiai dienai egzistuoja daug konteinerių orkestravimo įrankių, kurių pagalba valdomi diegimai. Visi orkestravimo įrankiai turi skirtingų būdų valdyti konteinerius ir šiame darbe vieno ar kito įrankio pranašumų ir trūkumų neaptarinėsime, tačiau įvardinsime pora populiarių įrankių

paminėtų knygoje „Continuous Delivery in Java“ [MB18]:

1. *„Amazon Elastic Container Service“* (trumpinamas *„Amazon ECS“*). Debesų kompiuterijos sprendimas sistemų aplinkų valdymui.
2. *„Kubernetes“*. Atvirojo kodo orkestratorius, kurio debesų kompiuterijos mokamus sprendimus jau yra įgyvendinę ir Google Cloud, Amazon AWS, Microsoft Azure. Remiantis „CNCF's Cloud Native Landscape“ [Fou21] apklausų duomenimis šio darbo rašymo metu „Kubernetes“ yra populiariausias konteinerių orkestravimo įrankis rinkoje.

8. Nuolatinio tiekimo brandos modelis

Projektuojant nuolatinio tiekimo grandinės šabloną svarbu įsitikinti, ar naudojamų praktikų rinkinys tikrai užtikrina aukštą gradinės kokybę. Šiam tikslui pasiekti pasinaudosime knygoje „Continuous Delivery“ [HF10] apibrėžtu nuolatinio tiekimo brandos modeliu. Šio modelio pagalba galima identifikuoti, kuriame lygmenyje yra sistemoje naudojama grandinė bei galima identifikuoti vietas, kurias reikėtų tobulinti. Knygos autorių nuolatinio tiekimo brandos modelis apibrėžia penkis brandos lygius įvairiuose kriterijuose. Lygiai prasideda nuo minus pirmo, kur aprašomos regresinio brandos lygio praktikos, o toliau seka siektinos praktikos keliant brandos lygį. Aukščiausias modelio lygmuo apibrežia procesų optimizavimą. Šiame darbe siekiama techninėmis galimybėmis įgyvendinti nuolatinio tiekimo grandinės šablono sprendimą, todėl pasiekti trečio lygmens nesieksime. Aptariamas brandos modelis apibrėžtas tokiuose kriterijuose: artefaktų kūrimas ir nuolatinė integracija, aplinkos ir diegimai, naujų versijų išleidimas, testavimas, duomenų valdymas, konfigūracijų valdymas.

Kaip jau minėta, šiame darbe nenagrinėjamas duomenų valdymo kriterijus, dėl to projektuodami grandinės šabloną į šį kriterijų dėmesio nekreipsime. Įgyvendinus visus kriterijaus reikalavimus lygmenyje, laikysime, kad šis lygis pasiektas nurodytame kriterijuje. Toliau pateikiama lentelė, kurioje apibrėžti pirmo ir antro brandos lygio reikalavimai (1 lentelė):

1 lentelė. Nuolatinio tiekimo brandos modelis.

Kriterijus	1 lygis (Nuoseklus)	2 lygis (Kiekybiškai valdomas)
Artefaktų kūrimas ir nuolatinė integracija.	Automatizuotas artefaktų kūrimas ir testavimas vykdomas po kiekvieno pakeitimų užfiksavimo. Valdomos priklausomybės. Perpanaudojami įrankiai ir skriptai	Kūrimo matai renkami ir vaizduojami. Nepavykusios integracijos grandinės tvarkomos.
Aplinkos ir diegimai.	Pilnai automatizuotas ir grafinės sąsajos pagalba naudojamas diegimo procesas. Toks pat procesas naudojamas visoms aplinkoms.	Orkestruojami diegimai. Naujų versijų diegimas ir ankstesnių versijų grąžinimas ištestuotas.
Naujų versijų išleidimas.	Pakeitimų valdymas ir peržiūros vykdomos. Reguliacinės ir atitikties sąlygos patenkintos.	Aplinkų ir programų sveikata tikrinima ir valdoma. Diegimų laikas stebimas.
Testavimas.	Automatizuoti vienetų ir priėmimo testai. Priėmimo testai rašomi su testuotojais.	Kokybės matai kaupiami ir stebimi. Nefunkciniai reikalavimai apibrėžti ir matuojami.
Konfigūracijų valdymas.	Bibliotekos ir priklausomybės valdomos. Versijų kontrolė apibrėžta pagal pakeitimų valdymo procesą.	Programuotojai apjunginėja pakeitimus į pagrindinę šaką bent kartą per dieną. Papildomos šakos kuriamos tik naujų versijų išleidimams.

9. Nuolatinio tiekimo grandinės šablono projektavimas

Projektuojant nuolatinio tiekimo grandinės šabloną, reiktų pradėti nuo pradinių reikalavimų, be kurių negalėtų egzistuoti nuolatinis tiekimas. Šie techniniai reikalavimai, kaip jau minėta, yra programinio kodo versijavimas ir automatizuotas artefaktų kūrimas.

Versijų kontrolės šakų strategija. Programinio kodo versijavimui reikalingas įrankis, kuris palaikytų versijų žymų funkcionalumą bei būtų galima patogiai kurti naujas versijų šakas. Tokius reikalavimus pilnai išpildo anksčiau aptartas įrankis Git, kurį panaudosime savo kuriamoje grandinėje. Brandos modelio primas lygis reikalauja, kad versijų kontrolė būtų apibrėžta pagal pakeitimų valdymo procesą. Šiam reikalavimui įgyvendinti reikia apsibrėžti pakeitimų valdymo procesą, tai yra pasirinkti šakų naudojimo strategiją.

Remiantis straipnsiu „Predicting Merge Conflicts in Collaborative Software Development" [Moe19] naudojant Git įrankį, didžiausias sunkumas su kuriuo susiduria programuotojai yra šakų apjungimas, kai atsiranda konfliktų tarp kodo versijų. Tokie konfliktai atsiranda tada, kai du skirtingi pakeitimai pritaikomi tai pačiai vietai PĮ kode. Kuo toliau šakos nutolsta viena nuo kitos, tuo didesnė rizika atsiranda konfliktams. Siekiant sumažinti konfliktų riziką grandinės šablonui pasirinksime tokią strategija, kuri skatintų kuo mažesnį kiekį papildomų šakų. Tokias praktikas užtikrina kamienu paremtas programavimas naudojamas kartu su pakeitimų jungikliais, siekiant kuo greičiau į produkcinę aplinką sukelti funkcionalumus. Taip pat, būtent ši kodo verijų kontrolės strategija padeda įgyvendinti ir konfigūracijų valdymo kriterijaus antro lygio reikalavimus, tai yra „programuotojai apjunginėja pakeitimus į pagrindinę šaką bent kartą per dieną." Aišku, techninėmis priemonėmis negalima užtikrinti, kad tai vyktų kartą per dieną, tačiau pasirinkta strategija suteikia priemones tai įgyvendinti. Vienas iš antro lygio reikalavimų - „Papildomos šakos kuriamos tik naujų versijų išleidimams". Remiantis pasirinktos strategijos dokumentacija [Ham17], kamienu paremtas programavimas palaiko atskiros šakos naujų versijų išleidimams praktiką, tačiau pabrėžta, kad galima alternatyva yra versijos išleidimas pasinaudojant Git įrankio žymos funkcionalumu. Tokiu būdu išvengiama šakų apliejimo ir išvengiama konfliktų rizikos, todėl šis būdas yra rekomenduojamas, jei įmanomas. Projektuojamame šablone naudosime dvi aplinkas, todėl papildomos šakos išleidimams galime išvengti, tačiau atsiradus poreikiui turėti daugiau aplinkų, rekomenduojama naudoti papildomą šaką diegimams. Dėl šios priežasties darysime išvadą, kad renkantis kamienu paremto programavimo strategiją, šį modelio reikalavimą galime ignoruoti.

Nors pagal pateiktą brandos modelį šakos tūretų būti kuriamos tik naujų versijų išleidimams, naujų versijų išleidimo kriterijus reikalauja pakeitimų peržiūrų vykdymo. Naudojantis kodo versijų kontrolės platformomis, tokiomis kaip GitHub ar GitLab, pakeitimų peržiūros vykdomos prieš apjungiant šakas. Dėl šios priežasties, mūsų sudarytame pakeitimų valdymo procese bus naudojamas šakų kūrimas funkcionalumams, kad juos, prieš apjungiant į pagrindinę šaką, galėtų peržiūrėti kiti komandos nariai ir pateikti pataisymų rekomendacijas. Pats Git įrankis neturi tokio funkcionalumo kaip peržiūros. Tai yra PĮ kodo talpinimo platformų funkcionalumas, todėl šiame darbe naudosimės kodo saugojimo platforma GitHub.

Kūrimo įrankis. Apsibrėžus pakeitimų valdymo procesą, kitas žingsnis yra automatizuotas

artefaktų kūrimas. Šiam žingnsiui reikalingas įrankis, kurio pagalba galėtume įgyvendinti tokius funkcionalumus:

1. Automatizuotas artefaktų kūrimas.
2. Automatizuotų testų vykdymas.
3. Bibliotekų ir kitų priklausomybių valdymas.
4. Docker atvaizdų automatizuotas kūrimas.

Visus šiuos funkcionalumus mūsų grandinėje galėtų įgyvendinti kūrimo įrankis Gradle, kurį įsitrauksime į savo grandinės šabloną. Šio įrankio pagalba galima apsibrėžti užduotis, kurios būtų vykdomos iš komandinės eilutės ir be papildomų konfigūracijų galėtų vykdyti artefaktų kūrimą, automatizuotų projekte aprašytų testų vykdymą. Taip pat, šis įrankis palaiko papildomus įskiepius, kurie įgalintų sukurti užduotį, kuri automatizuotai iš sukurto artefakto pagamintų Docker atvaizdą be papildomo Dockerfile apsirašymo. Turint tokį įskiepį, mūsų kuriamų atvaizdų konfiguracija būtų laikoma toje pačioje repozitorijoje. Tai reikštų, kad šios konfigūracijos pakeitimai visada patektų į užfiksavimus ir Docker atvaizdo žymai būtų galima naudoti užfiksavimo identifikatorių arba „git describe“ komandos reikšmę. Ši reikšmė suteikia daugiau informacijos apie vėliausią iki užfiskavimo egzistuojančią versijų kontrolės žymą, todėl atvaizdų žymoms naudosime apibūdinančios komandos reikšmę. Artefaktų kūrimo įrankio panaudojimas ir apibrėžtų įrankiui reikalavimų tenkinimas išpildo konfigūracijų valdymo pirmo lygio likusią sąlygą - „bibliotekos ir priklausomybės valdomos“. Po šios sąlygos įvykdymo gradinės šablonas pasiektų antrą brandos lygį konfigūracijų valdymo kriterijuje.

Įrankis Gradle palaiko projekto versijavimą, tačiau siekiant išvengti naujų pakeitimų užfiksavimų naujos versijos sukūrimui, Gradle projekto versijos nekeisime ir paliksime standartine *„0.0.1“*. Pagrindinis privalumas versijuoti Gradle projektą yra tas, kad paleidžiant programą, ši versija gali būti atspindima programos žinyne, tačiau aplinkose versijas bus galima identifikuoti diegimo orkestratoriaus pagalba bei papildomai bus pridėtas versijos atvaizdavimas žinyne, kur versijos reikšmė bus tokia pati kaip projekto.

Testavimas. Kita svarbi gradinės dalis yra funkcinių reikalavimų automatizuotų testų vykdymas. Šios dalies reikalavimai apibrėžti brandos modelio testavimo kriterijaus pirmame lygmenyje. Testų rašymą ir naudojamas bibliotekas plačiau aptarsime grandinės realizavimo skyriuje. Kol kas darysime tokią prielaidą, kad vienetų, komponentų ir priėmimo testai yra parašyti ir vykdomi artefaktų kūrimo įrankio pagalba. Vienas iš pirmo lygio reikalavimų - „priėmimo testai rašomi su testuotojais“. Šio reikalavimo techninėmis priemonėmis užtikrinti negalime, todėl jį ignoruosime.

Svarbu paminėti, kad priėmimo testai tūrėtų būti vykdomi su paleista programa, todėl grandinės šablone juos vykdysime po Docker atvaizdo sukūrimo.

Gradle įrankio artefakto kūrimo užduoties metu, prieš vykdant patį artefakto surinkimą, vykdomi ir automatizuoti testai, kurių vykdymo nesekmės atveju, grandinė sustos ir artefaktas nebus kuriamas. Taip pat, prieš artefakto kūrimo žingsnį galima vykdyti nefunkcinių reikalavimų testavimą. Nefunkcinių reikalavimų kiekis ir tipai priklauso nuo reikalavimų iškeltų informacinei sistemai. Šiame darbe kuriamai grandinei ir projektui ypatingų reikalavimų nekeliama.

Šiame žingsnyje apsiribosime tokiais reikalavimais, kurių tikrinimą užtikrina paslauga SonarCloud. Šioje paslaugoje sukonfigūruotas kokybės standartas (angl. *„Quality Gate“*) užtikrina tokius reikalavimus, kaip programinio kodo padengimą testais procentas, dublikuojamų kodo eilučių procentas, įrankio sudaromas kodo palaikymo reitingas, kodo patikimumo reitingas bei saugumo reitingas. Taip pat, apsibrėšime bendrą projekto konvencija savo integruotoje kūrimo aplinkoje naudoti įskiepį SonarLint. Šių įrankių pagalba, bus galima matuoti kodo kokybę ir greičiau rasti potencialias klaidas. Analizės rezultatų kaupimas užtikrintų brandos modelio testavimo kriterijaus antro lygio reikalavimus - „kokybės matai kaupiami ir stebimi“ bei „nefunkciniai reikalavimai apibrėžti ir matuojami“. Tokiu būdu projektuojamas grandinės šablonas testavimo kriterijuje pasiektų antrą lygmenį.

Nuolatinės integracijos įrankis. Sekantis nuolatinio tiekimo grandinės šablono sudarymo žingsnis būtų nuolatinės integracijos įrankio pasirinkimas. Kaip jau minėta ankstesniuose skyriuose, kuriamame grandinės šablone stengsimės nuolatinės integracijos įrankį pritaikyti ir diegimams, todėl prie reikalavimų įrankiui priskirsime ir diegimų kontekste, bet ne orkestratoriams, pritaikomus reikalavimus. Remianti pateiktu brandos modeliu nuolatinės integracijos įrankio reikalavimams galėtume priskirti šiuos:

1. Automatizuotas artefaktų kūrimas ir testavimas vykdomas po kiekvieno pakeitimų užfiksavimo.
2. Artefaktų kūrimo matai renkami ir vaizduojami.
3. Kodo kokybės matai kaupiami ir stebimi.
4. Pilnai automatizuotas ir grafines sąsajos pagalba naudojamas diegimo procesas.
5. Toks pat procesas naudojamas visoms aplinkoms.
6. Diegimų laikas stebimas.

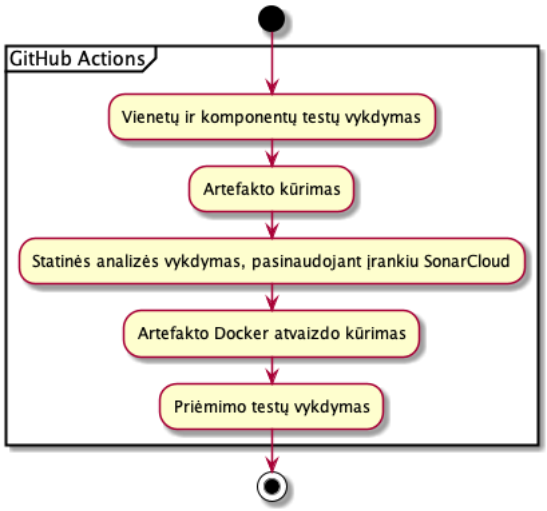
Šiuos reikalavimus įgyvendinti, kartu su jau apibrėžtų šablone naudojamų praktikų pagalba, turbūt pavyktų su bet kuriuo nuolatinės integracijos skyriuje paminėtu įrankiu, tačiau siekdami pasigilinti į įrankius giliau, išsikelsime papildomus reikalavimus. Pasirenkant nuolatinės integracijos įrankį, svarbu nuspręsti, ar rinksimes serverinį įrankį, ar įrankį naudosime kaip paslaugą. Šiame darbe stengsimes vengti įrankių, kuriems reikalingi papildomi šablono naudotojų ištekliai. Dėl šios priežasties rinksimes iš tokių nuolatinės integracijos įrankių, kuriuos būtų galima naudoti kaip paslaugą. Toliau, siekiant neturėti nuolat veikiančio serverio, skirto integravimui, stengsimės ieškoti vis populiarėjančių beserverinės technologijos sprendimų. Tai reiškia, kad už nuolatinės integracijos įrankio paslaugą mokama tik tada, kai ji būna naudojama, nes beserverinė technologija įgalina neturėti visą laiką veikiančio dedikuoto serverio, o integraciją vykdyti tik tada, kai tas reikalinga. Vienas iš tokių sprendimų yra pasirinktos kodo saugojimo platformos GitHub sukurtas įrankis „GitHub Actions“. Taigi, šį įrankį ir pasirinksime, kaip kuriamame šablone naudojamą nuolatinės integracijos įrankį. „Github Actions“ įrankis palaiko daug integracijų su kitais įrankiais, įskaitant ir SonarCloud, ir diegimus į debesų kompiuterijos technologija pagrįstas aplinkas. Taip pat, šis įrankis turi grafinę sąsają, kurios pagalba galima aiškiai matyti, kiek užtruko žingsniai, kaupti žingsnių vykdymo kuriamus žinynus bei kitus matus. Įrankis, taip pat, palaiko taisyklių apsibrėžimą, kurių pagalba galima vykdyti grandinę po kiekvieno pakeitimų

užfiksavimo. Siekiant įgyvendinti pirmo lygio reikalavimą - „perpanaudojami įrankiai ir skirptai“, pabandysime perpanaudoti apsirašytas nuolatinės integracijos įrankio eigas.

Nuolatinės integracijos įrankio grandinėje nepavykus įgyvendinti žingsnio, grandinė sustabdoma ir kiti žingsniai nevykdomi. Mūsų grandinės šablono atveju diegimas bus paskutinis žingsnis, todėl nepavykus bent vienam žingsniui PĮ nebus diegiama į aplinkas. Vienas iš brandos modelio artefaktų kūrimo ir nuolatinės integracijos kriterijaus antro lygio reikalavimų - „nepavykusios integracijos grandinės tvarkomos.“ Šio reikalavimo negalime pilnai užtikrinti techninėmis priemonėmis, tačiau su naudojamais įrankiais nepavykus bent vienam žingsniui diegimas nebus vykdomas, o tai reiškia, kad negalimi tolimesni diegimai, kol grandinė nebus sutvarkyta. Tokiu būdu, kiek įmanoma techninėmis priemonėmis užtikriname šio reikalavimo įgyvendinimą.

Kuriant programinę įranga svarbiausia, kad kodas įgyvendintų norimus pakeitimus, bei būtų galima sukurti artefaktą, todėl statinė kodo analizė, kuri tikrina koko kokybė, bus vykdoma po funkcinių testų ir artefakto kūrimo žingsnių.

Taigi, su šiuo įrankiu galime įgyvendinti visus išsikeltus reikalavimus ir su kuriamu grandinės šablonu pasiekti antrą brandos lygį kriterijuje artefaktų kūrimas ir nuolatinė integracija. Toliau pateikta nuolatinės integracijos žingsnių vykdymo diagrama (6 pav.):



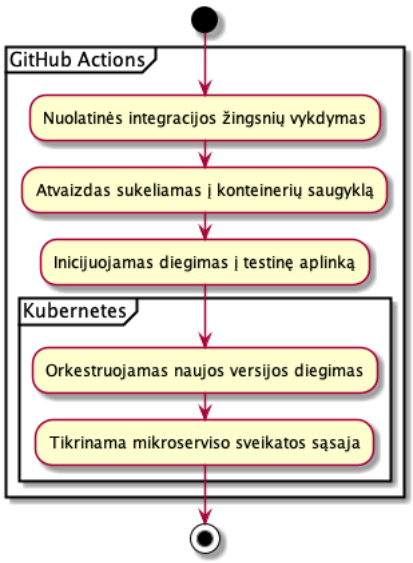
6 pav. Nuolatinės integracijos žingnsių vykdymo diagrama.

Konfigūracijų valdymas. Kaip jau minėta skyriuje apie konfiguracijų valdymą, išorinių ir vidinių konfigūracijų naudojimas priklauso nuo reikalavimų, keliamų projektui. Mūsų kuriamame šablone išorinės konfigūracijos bus funkcionalumų jungikliai, o kita konfigūracijos dalis bus laikoma toje pačioje repozitorijoje. Tokios reikšmės kaip duomenų bazių slaptažodžiai ir kita slapta informacija, bus saugomos GitHub repozitorijos slaptažodžių saugykloje ir reikšmės paduodamas diegimo metu į konteinerius. Grandinės šablone pasirinktas naudoti kamienu paremtas programavimas su funckionalumų jungikliais, todėl siekiant įgalinti tuos jungiklius keisti ne tik programuotojams, panaudosime įrankiais, leidžiančiais per grafinę sąsają valdyti funkcionalumų jungiklius. Tokiu būdu ši konfigūracija bus išorinė. Šiuos nustatymus periodiškai galėtų gauti kuriama programa ir su jais valdyti naujų funkcionalumų prieinamumą. Įrankis turėtų nenaudoti sistemos išteklių ir funkcionalumų jungiklius (kurie neturi jokios jautrios informacijos)

saugoti įrankio tiekėjo duomenų bazėje. Tokius reikalavimus atitinkantis įrankis yra „ConfigCat“, kurį panaudosime kuriamame projekte.

PĮ diegimas į aplinkas. Paskutinis grandinės šablono žingsnis yra artefaktų diegimas į aplinkas. Prieš diegiant sukurtus Docker atvaizdus, svarbu juos sukelti į atvaizdų saugyklą. Iš saugyklos aplinkose šiuos atvaizdus orkestratorius galėtų parsisųsti pagal nurodytą žymą. Sukėlus šiuos atvaizdus galima pradėti diegimą. Mūsų pasirinkto įrankio „GitHub Actions“ pagalba galima pasidaryti papildomą žingsnį artefaktų diegimui. Taip pasinaudojant grafine sąsaja, diegimą galėsime automatizuoti ir valdyti. Nuolatinės integracijos įrankyje apsirašius skirtingas aplinkas, galima suvienodinti diegimus į visas aplinkas. Tokiu būdu patenkinamos visos pirmo lygio aplinkos ir diegimų kriterijaus brandos modelyje sąlygos. Siekiant užtikrinti antro lygio brandą būtina diegimus orkestruoti, bei ištestuoti naujų ir senų versijų diegimą. Testavimus atliksime šablono realizavimo skyrelyje, todėl dabar darysime prielaidą, kad diegimai ištestuoti. Dar vienas svarbus reikalavimas naujų versijų išleidimo kriterijuje - „aplinkų ir programų sveikata tikrinama ir valdoma“. Šiems paminėtiems reikalavimams įgyvendinti reikalingas diegimų orkestratorius, kurio pagalba galėtume orkestruotai diegti konteinerius į aplinkas. Diegimui pasinaudosime įrankiu „Kubernetes“. Šis orkestratorius palaiko mum tinkamą „Blue/Green Deployment“ diegimo strategiją ir pastoviai tikrina konteinerių sveikatą. Taip pat, siekdami nenaudomi savų resursų ir neturėti savo infrastruktūroje dedikuotų serverių, pasinaudosime įmonės Google debesų kompiuterijos sprendimu ir naudosime jų siūlomą „Kubernetes“ orkestratoriaus paslaugą. Tokių budu galėsime susikurti aplinkas ir jas valdyti.

Kuriamame gradinės šablone diegimai į testinę aplinką bus vykdomi automatiškai, pakeitimams atsidūrus pagrindinėje šakoje. Toliau pavaizduoti diegimo į testinę aplinką žingsniai (7 pav.):



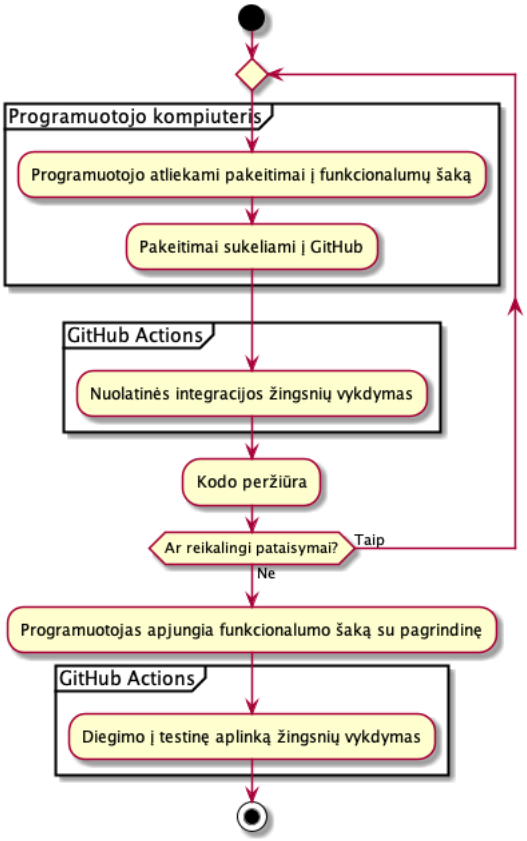
7 pav. Diegimo į testinę aplinką žingnsiai.

Paskutinis likęs brandos modelio naujų versijų išleidimų reikalavimas yra „reguliacinės ir atitikties sąlygos patenkintos.“ Remiantis brandos modelio šaltiniu [HF10], šis reikalavimas yra aktualus, kai priklausomai nuo projekto dalykinės srities keliami teisiniai reikalavimai sistemai,

pavyzdžiui ne visiem darbuotojam prieinamos visos aplinkos arba norint atlikti diegimą, reikalaujama vadovų patvirtinimo. Mūsų kuriamame šablone tokių atitikties ir reguliacinių reikalavimų nėra keliama, tačiau tokiam poreikiui atsiradus, platformoje GitHub egzistuoja funkcionalumas, leidžiantis apriboti teises vartotojams. Siekiant grandinės šabloną išlaikyti lengvai perpanaudojamu, šį reikalavimą priklausantį nuo dalykinės projekto srities, praleisime.

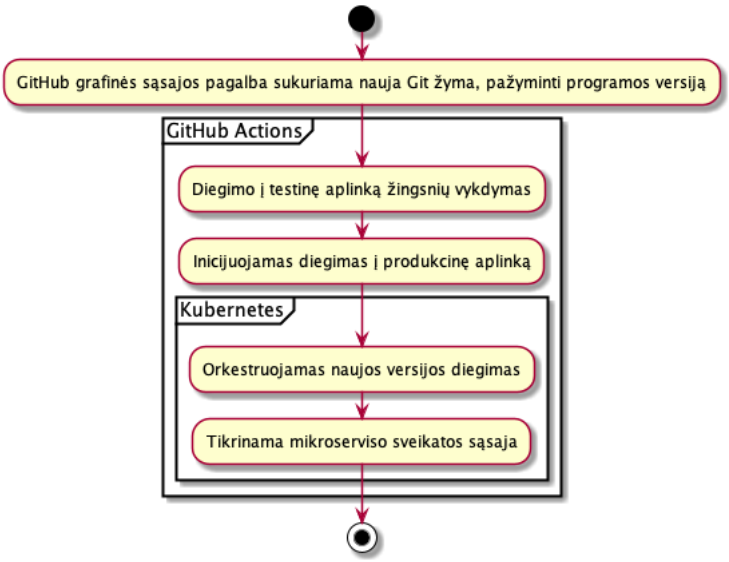
Apibendrinimas.

Apsibrėžus kodo versijų kontrolės šakų strategiją, nuolatinės integracijos žingsnius bei diegimų į testinę aplinką eigą, pateikiamas bendra diagrama, vaizduojanti pakeitimų grandinę nuo programuotojo kompiuterio iki pagrindinės šakos (8 pav.):



8 pav. Kodo pakeitimų prijungimo į pagrindinę šaką eiga.

Diegimai į produkcinę aplinką inicuojami sukūrus naują Git žymą, pažyminčią naują programos versiją, GitHub platformoje pasinaudojant grafine sąsaja. Po žymos sukūrimo vykdomi diegimo į testinę aplinką žingsniai su nauja versijos žyma bei tas pats Docker atvaizdas diegiamas į produkcinę aplinką. Tokia eiga vaizduojamas diagramoje (9 pav.):



9 pav. Diegimo į produkcinę aplinką žingsnių vykdymo eiga.

Projektuojamas grandinės šablonas atitinka visų brandos modelyje apibrėžtų kriterijų antrą lygmenį.

10. Suprojektuoto gradinės šablono realizavimas

Šablono realizacija viešai prieinama GitHub platformoje. Visas PĮ kodas patalpintas repozitorijoje `https://github.com/mila973/CD-pipeline-app`.

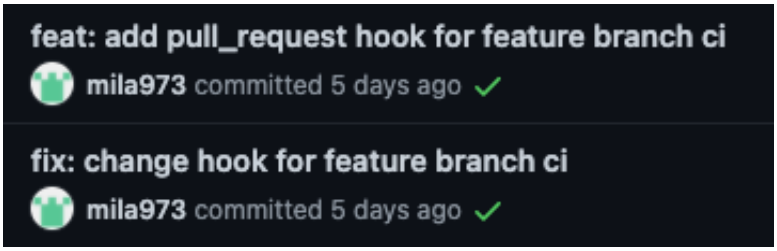
10.1. Šablono įgyvendinimo aprašymas

Šablono įgyvendinimui sukurta saityno paslaugų programa pasinaudojant Spring Boot karkasu.

Git pakeitimų užfiksavimo žinutės. Žinučių rašymui pasirinktas „Git Karma“ formatas. Įprastu atveju formatas susideda iš trijų dalių:

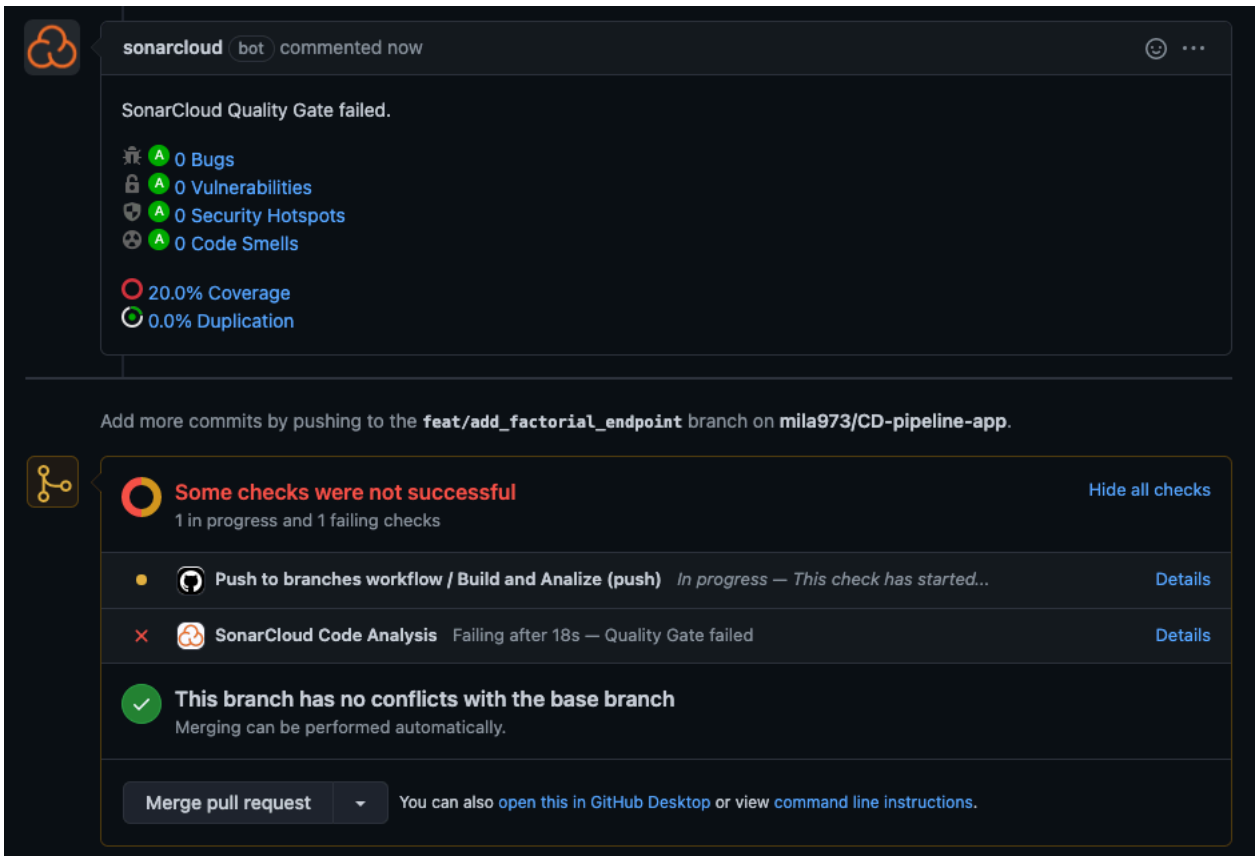
1. Pakeitimo tipas (naujas funkcionalumas, pataisymas, kodo pertvarkymas ir t.t.)
2. Pakeitimo užduoties numeris, jeigu naudojama projektų valdymo sistema.
3. Pakeitimo aprašymas.

Šioje realizacijoje nenaudojama jokia projektų valdymo sistema, todėl antras punktas praleidžiamas (10 pav.):



10 pav. Pakeitimo užfiksavimų žinučių pavyzdys.

Nefunkcinių reikalavimų testavimas. Kaip jau minėta projektavimo skyriuje nefunkcinių reikalavimų testavimas buvo įgyvendintas pasinaudojant įrankiu „SonarCloud“. Nuolatinės integracijos įrankio pagalba artefaktų kūrimo įrankis vykdo Gradle užduotį „build“, kurios metu vykdomi projekte aprašyti automatizuoti testai ir surenkamas artefaktas. Taip pat, pasinaudojant Gradle įskiepio „JaCoCo“ pagalba, sugeneruojama kodo testavimo ataskaita, kuri siunčiama į „SonarCloud“. Šio įrankio puslapyje atsidarius projekto paskyrą, galima matyti visas statinės analizės metu rastas problemas Java klasėse, bei kodo testavimo analizės rezultatus. „SonarCloud“ paskyroje sukonfigūruotas kokybės standartas (angl. *„Quality Gate“*). Šiame šablone pasirinktas rekomenduojamas paslaugos tiekėjų standartas, kuris reikalauja programinio kodo padengimo testais bent aštuoniasdešimt procentų, nedaugiau trijų procentų kodo dublikavimo, bei projektavimo dalyje paminėtų reitingų aukščiausio įvertinimo. Šio standarto neatitikimas indikuojamas GitHub Actions įrankyje ir grandinė sustabdoma. Taip pat, pagrindiniai faktai atvaizduojami GitHub platformoje (11 pav.):



11 pav. Nefunkcinių reikalavimų testavimo rezultatų atvaizdavimas GitHub platformoje.

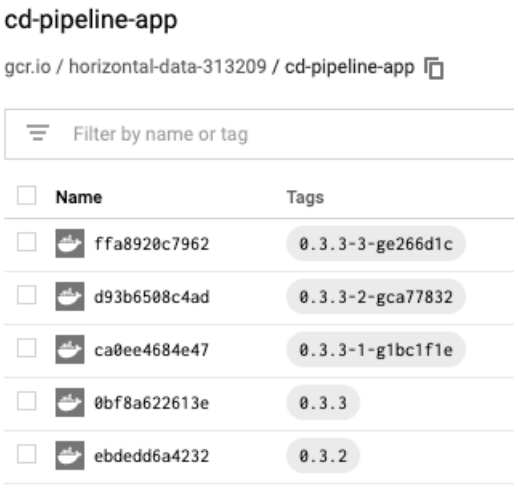
Šablono projektui nustatytas numatytasis kokybės standartas, kurio vienas iš reikalavimų automatizuotais testais padengti bent aštuoniasdešimt procentų kodo.

Funkcinių reikalavimų testavimas. Šablone įgyvendinti visi trys teorinėje darbo dalyje aprašyti automatizuotų testų tipai. Vienetų ir komponentų testai realizuoti su „Junit5“, „Mockito“ ir „WireMock“ bibliotekomis. Su „WireMock“ biblioteka sukuriama pamėgdžiojanti saityno paslauga, į kurią kai kreipiamasi su testo scenarijuje apibrėžtais parametrais, gražinamas nurodytas atsakymas. Tokiu būdu patikrinamas integracijos su kita saityno paslauga veikimas. Priėmimo testai realizuoti, atskirtame nuo programos kodo aplanke. Siekiant užtikrinti priėmimo testų vykdymą su paleista kuriama programa, dedikuotame aplanke buvo apsirašytas Docker atvaizdo paleidimas. Pasinaudojant Gradle įrankio įskiepiu „Avast Docker compose“, kurio pagalba prieš vykdant apsirašytus testus nuolatinės integracijos serverio aplinkoje automatiškai paleidžiamas programos Docker konteineris su atvaizdu. Taip pat, konteineris ištrinamas po testų vykdymo. Turėdami Docker konteineryje veikiančią ir prieinamą programą, ją galime testuoti tiesiogiai bendraudami su konteineriu pasinaudojant „Awaitility“ ir „Rest-assured“ bibliotekomis. Su „Awaitility“ biblioteka apsirašytuose testuose tikrinama programos konteinerio sveikatos sąsaja ir programai pasileidus sėkmingai testuojami apsirašyti scenarijai kviečiant kuriamos programos teikiamas sąsajas per HTTP protokolą.

Artefakto, Docker atvaizdo kūrimas ir sukėlimas į saugyklą. Kaip jau minėta anksčiau, programos artefakto kūrimas vykdomas pasinaudojant Gradle įrankiu. Kartu su artefakto kūrimu vykdoma ir statinė analizė bei automatizuoti testai, išskyrus priėmimo testus. Projekte naudo-

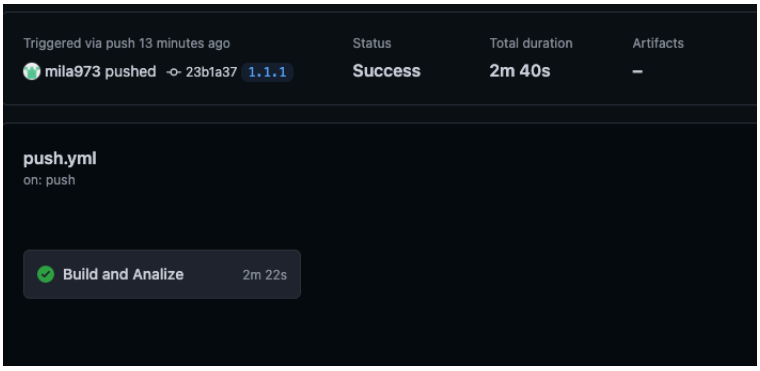
jamas Gradle „Bmuschko Docker“ įskiepis. Šis sukonfiguruotas įskiepis įgalina naują Gradle užduotį, su kuria iš sugeneruoto artefakto sukuriamas Docker atvaizdas.

GitHub Actions įrankio serveryje po artefakto kūrimo ir priėmimo testų vykdymo prisijungiama prie „Google Cloud“ konteinerių registro ir sukeliamas sukurtas atvaizdas (12 pav.).



12 pav. Docker atvaizdų saugykla.

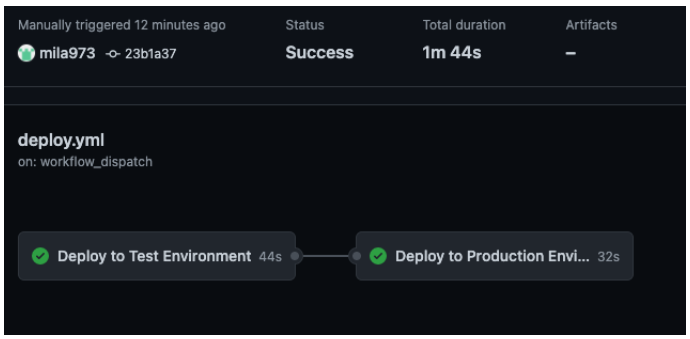
Nuolatinės integracijos įrankio konfigūracija ir matų atvaizdavimas. GitHub Actions įrankyje buvo sukurtos trys darbų vykdymo eigos (angl. *„workflow“*). Pirmoji darbų eiga atsakinga už artefaktų sukūrimą ir analizę. Taip pat, ši eiga sukelia sukurtą Docker atvaizdą į saugyklą (13 pav.):



13 pav. Pirmosios eigos vykdymas.

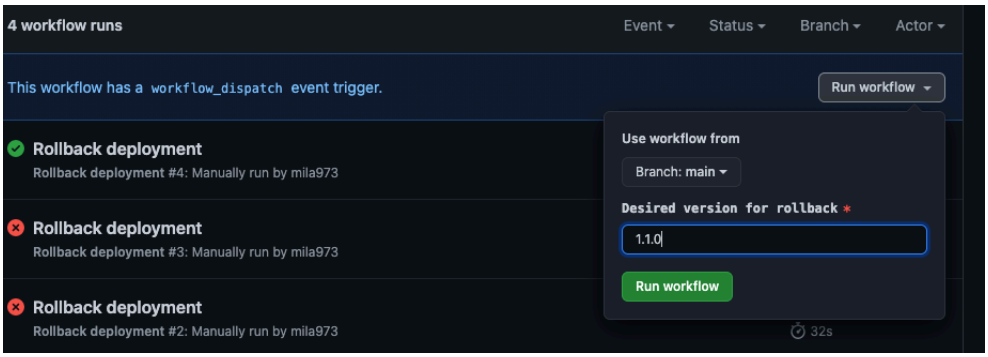
Antroji eiga atsakinga už diegimus į testinę ir produkcinę aplinkas. Sukėlus pakeitimus į ne pagrindinę šaką, inicijuojamas tik pirmosios eigos vykdymas. Sukėlus pakeitimus į pagrindinę kodo šaką inicijuojami pirmos eigos vykdymas, kuris iškviečia ir antrąją eigą. Šiuo atveju atroji eiga diegimą vykdo tik į testinę aplinką.

Sukūrus naują Git žymą, vykdoma pirmoji eiga bei diegimai į abi aplinkas (14 pav.):



14 pav. Diegimų eigos vykdymas.

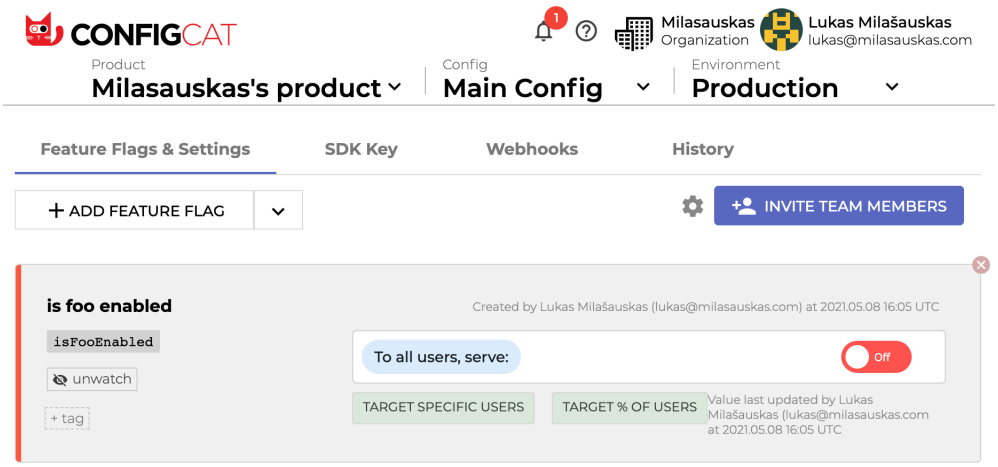
Trečioji eiga, atsakinga už senų versijų grąžinimą į produkcinę aplinką. Šioje eigoje egzistuoja tik vienas diegimo darbas, kadangi ankstesnės versijos jau buvo ištestuotos ir sudiegtos. Eiga vykdoma, inicijuojant iš GitHub Actions grafinės sąsajos, ir reikalauja vieno parametro - žymos, kuri nurodo norimą sudiegti veriją. Vykdant šią ir antrąją eigą patikrinama ar Docker atvaizdas su nurodyta žyma egzistuoja atvaizdų saugykloje. Jeigu toks atvaizdas egzistuoja, inicijuojamas diegimas. Toliau pavaizduotas rankinis trečios eigos iškvietimas pasinaudojant GitHub Actions grafine sąsaja (15 pav.):



15 pav. Ankstesnių versijų gražinimo eigos iškvietimas.

Siekiant perpanaudoti apsirašytas eigas GitHub platformoje susikūrus orgnanizacijos profilį, panaudotos eigos buvo aprašytos kaip šablonai, kuriuos galima naudoti visose orgnazicijos repozitorijose. Naujose repozitorijose grafinės sąsajos pagalba kuriant naujas eigas, galima pasirinkti iš jau sukurtų šablonų.

Funkcionalumų jungiklių valdymas. Jungiklių valdymas įgyvendintas pasinaudojant „ConfigCat“ įrankiu. Šio įrankio klientinė biblioteka sudiegta į programą ir sukonfiguruota taip, kad kas penkias minutes naujausia aplinkos jungiklių konfigūracija būtų gaunama iš įrankio duomenų bazės. Jungiklius aplinkose galima valdyti grafinės sąsajos pagalba „ConfigCat“ platformoje (16 pav.):

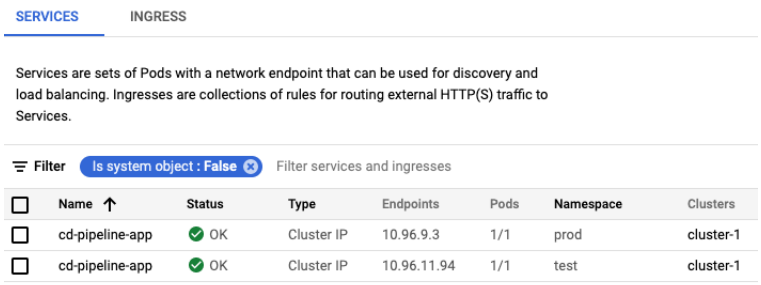


16 pav. „ConfigCat“ funkcionalumų jungiklių valdymo grafinė sąsaja.

Taip pat, programoje galimas jungiklių konfigūravimas pasinaudojant Spring Boot karkaso nustatymų failu. Jame galima nurodyti „ConfigCat“ konfigūracijos išjungimą ir naudoti sukonfigūruotus lokalius nustatymus. Tokiu būdu programuotojai gali nesunkiai valdyti nustatymus savo lokalioje aplinkoje testuodami programą.

Programuotojai projekte apsirašę naujus jungiklius, po savo šakos apjungimo su pagrindine projekto šaka privalo sukurti naujus jungiklius „ConfigCat“ grafinėje sąsajoje ir palikti juos išjungtus. Išimtis taikoma, siekiant patiems ištestuoti naują funkcionalumą. Vėliau jungiklius administruoja analitikai ar kiti sistemą administruojantys darbuotojai. Rašant priėmimo testus, Docker konteinerių konfigūracijoje visos funkcionalumų vėliavos nurodomos kaip įjungtos ir „ConfigCat“ nustatymų gavimas yra išjungtas, siekiant užtikrinti stabilų testų vykdymą, nepriklaausantį nuo aplinkų nustatymų.

Docker atvaizdų diegimas į aplinkas. Kaip jau minėta, Docker atvaizdų saugykla šablioniame projekte yra „Google Cloud“ konteinerių registras. Siekiant įgyvendinti „Blue/Green Deployment“ diegimo strategiją, pasinaudota „Google Cloud Kubernetes Engine“ paslauga. Kubernetes orkestravimo įrankio pagalba sukurtas telkinys (angl. *„cluster“*) ir dvi erdvės (angl. *„namespace“*) testinei ir produkcinei aplinkai. Šiose erdvėse buvo sukurtos paslaugos (angl. *„services“*), kuriose sukuriami programų diegimai. Paslaugų konfigūracijoje nurodoma, kuri sudiegta programos versija turi būti naudojama. Norint sudiegti naują programos versiją, sukuriamas naujas diegimas su visa konteinerio konfigūracija. Tada paslaugoje sukuriami nauji konteineriai su nurodytais Docker atvaizdais. Kai konteinerių sveikatos sąsaja grąžina teigiamą atsakymą apie jo pasiekiamumą ir programos pasiledimą, paslaugos konfigūracijoje apkeičiama naudojama versija į naujai sudiegto konteinerio. Šis tikrinimas ir automatinis konteinerių naudojimo apkeitimas įgyvendintas pasinaudojus įrankiu „Helm“. Kubernetes įrankis per nustatytą laiko periodą tikrina sveikatos sąsają ir programai išsijungus, konteineriai būna perkraunami pagal jų diegimo konfigūraciją (17 pav.):

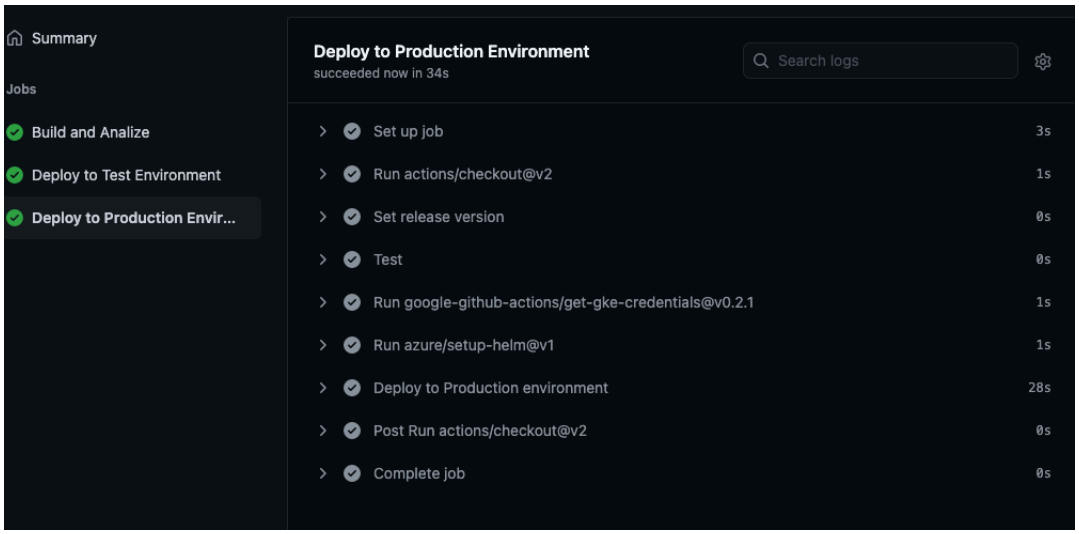


17 pav. Kubernetes telikinyje sukurtos paslaugos.

Naujos Git žymos sukūrimo procesas ir Docker atvaizdų diegimas į produkcinę aplinką. Norint sudiegti pakeitimus į produkcinę aplinką, GitHub platformoje, pasinaudojant grafine sąsaja sukuriama nauja Git žyma, kuri mūsų šablone apibrėžiama kaip programos versija. Naujos versijos kūrimas vykdomas pasinaudojant semantinio versijavimo praktika (18 pav.).

18 pav. Naujos programos versijos sukūrimas.

Sukūrus naują žymą, inicijuojamas diegimas į produkcinę aplinką. Diegimo metu vykdomi visi trys GitHub Actions darbai paeiliui (19 pav.).



19 pav. Inicijuojamas diegimas į produkcinę aplinką.

10.2. Grandinės šablono palyginimas su brandos modeliu

Taigi, suprojektavus ir įgyvendinus nuolatinio tiekimo grandinės šabloną galime apžvelgti įgyvendintus brandos modelio reikalavimus. Lentelėje pateikami įgyvendinti brandos modelio reikalavimai ir praktikos ar įrankiai, padedantys juos realizuoti (2 lentelė):

2 lentelė. Komunikavimo tipų palyginimo rezultatai.

Reikalavimas	Įrankis, praktika
Automatizuotas artefaktų kūrimas ir testavimas vykdomas po kiekvieno pakeitimų užfiksavimo	GitHub Actions
Valdomos priklausomybės	Gradle
Perpanaudojami įrankiai ir skriptai	GitHub Actions
Pilnai automatizuotas ir grafinės sąsajos pagalba naudojamas diegimo procesas.	GitHub Actions
Toks pat procesas naudojamas visoms aplinkoms.	GitHub Actions
Pakeitimų valdymas ir peržiūros vykdomos.	Įrankio Git ir platformos GitHub peržiūrų grafinės sąsajos pagalba
Automatizuoti vienetų ir priėmimo testai.	Junit, Mockito, Awaitility, Rest-assured
Bibliotekos ir priklausomybės valdomos.	Gradle
Versijų kontrolė apibrėžta pagal pakeitimų valdymo procesą.	Kamienu paremtas programavimas su funkcionalumų jungikliais
Kūrimo matai renkami ir vaizduojami.	GitHub Actions
Orkestruojami diegimai.	Kubernetes, Helm
Naujų versijų diegimas ir ankstesnių versijų grąžinimas ištestuotas.	GitHub Actions, Helm, Kubernetes
Aplinkų ir programų sveikata tikrinima ir valdoma	Kubernetes
Diegimų laikas stebimas.	GitHub Actions
Kokybės matai kaupiami ir stebimi.	SonarCloud
Nefunkciniai reikalavimai apibrėžti ir matuojami.	SonarCloud
Programuotojai apjunginėja pakeitimus į pagrindinę šaką bent kartą per dieną.	Kamienu paremtas programavimas su funkcionalumų jungikliais

Taip pat, dėl minėtų priežasčių projektavimo skyriuje nebuvo siekta įgyvendinti šių kriterijų:

1. Priėmimo testai rašomi su testuotojais.
2. Nepavykusios integracijos grandinės tvarkomos.
3. Papildomos šakos kuriamos tik naujų versijų išleidimams.
4. Reguliacinės ir atitikties sąlygos patenkintos.

Atsižvelgus į įgyvendintus kriterijus galima teigti, kad suprojektuotas ir realizuotas nuolatinio tiekimo grandinės šablonas pasiekė antrą, kiekybiškai valdomą brandos lygį.

Rezultatai ir išvados

Rezultatai

1. Aprašytos pagrindinės nuolatinio tiekimo grandinės dalys bei būdai jas įgyvendinti.
2. Suprojektuotas ir įgyvendintas nuolatinio tiekimo grandinės šablonas, kuris:
   (a) įgyvendina nuolatinio tiekimo brandos modelio antro lygio reikalavimus;
   (b) paremtas debesų kompiuterijos sprendimais.

Išvados

1. Egzistuoja brandos modelio reikalavimų, kurie ne visada atitinka naudojamų praktikų rekomendacijas.
2. Egzistuoja grandinės dalių, kurių brandos modelio reikalavimų įgyvendinimui tinkamos kelios skirtingos praktikos ar įrankiai, todėl būtina išsikelti papildomus reikalavimus, siekiant atsirinkti tinkamiausią įrankį ar praktiką.
3. Projekto versijavimas gali būti įgyvendinamas skirtingais įrankiais, projektuojant nuolatinio tiekimo grandinę ne visada praktiška versijuoti visomis priemonėmis, todėl svarbu pasirinkti, kas bus projekto versijos tiesos šaltinis.
4. Suprojektuotas grandinės šablonas tinkamas naudoti turint dvi aplinkas kuriamai sistemai. Atsiradus daugiau aplinkų, rekomenduojama koreguoti diegimo į aplinkas eigą, pavyzdžiui naujų versijų išleidimams naudoti išleidimo šakas, bei naujam procesui pritaikyti nuolatinės integracijos įrankį.

Summary

This paper analyzes Continuous Delivery (CD) practices and their usage in order to implement pipeline template for Java microservices. The maturity model of CD is used to evaluate the quality of the template. This paper attemps to satisfy only technical requirements of a CD pipeline maturity. After the analysis of practices and tooling for CD pipeline, the template is designed and qualifies for a second level of the maturity model. In order to be sure if the chosen practices do not conflict and they can be used together, the designed template is implemented and tested for multiple scenarios.

Šaltiniai

[Act20] ActiveState. State of ci/cd 2020 survey results. https://www.activestate.com/resources/datasheets/ci-cd-survey-results/, 2020. tikrinta 2021-04-22.

[Cha13] Scott Chacon. Github flow: the best way to use git and github. https://githubflow.github.io/, 2013. tikrinta 2021-04-22.

[Dij70] Edsger W. Dijkstra. Notes on structured programming. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF, 1970. tikrinta 2021-05-10.

[Dri10] Vincent Driessen. A successful git branching model. https://nvie.com/posts/a-successful-git-branching-model/, 2010. tikrinta 2021-04-22.

[FFdC+19] Wagner Felidre, Leonardo Furtado, Daniel A. da Costa, Bruno Cartaxo, and Gustavo Pinto. Continuous integration theater. https://arxiv.org/pdf/1907.01602.pdf, 2019. tikrinta 2021-04-22.

[Fit09] Timothy Fitz. Continuous deployment. http://timothyfitz.com/2009/02/08/continuous-deployment/, 2009. tikrinta 2021-04-22.

[Fou21] Cloud Native Computing Foundation. Cncf's cloud native landscape. https://landscape.cncf.io/, 2021. tikrinta 2021-05-01.

[Fow06] Martin Fowler. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, 2006. tikrinta 2021-05-10.

[Fow13] Martin Fowler. Continuousdelivery. https://martinfowler.com/bliki/ContinuousDelivery.html, 2013. tikrinta 2021-04-15.

[Ham17] Paul Hammant. Trunk based development: introduction. https://trunkbaseddevelopment.com/, 2017. tikrinta 2021-04-22.

[HF10] Jez Humble ir David Farley. *Continuous Delivery*. Addison-Wesley, Upper Saddle River, United States of America, 2010. tikrinta 2021-05-01.

[Hum10] Jez Humble. Continuous delivery vs continuous deployment. https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, 2010. tikrinta 2021-04-22.

[Las18] Steve Lasker. Docker tagging: best practices for tagging and versioning docker images. https://stevelasker.blog/2018/03/01/docker-tagging-best-practices-for-tagging-and-versioning-docker-images/, 2018. tikrinta 2021-05-10.

[Mak20] Vasanth K. Makam. Continuous integration on cloud versus on premise: a review of integration tools. http://article.sapub.org/10.5923.j.ac.20201001.02.html, 2020. tikrinta 2021-05-10.

[Mar18] Rajkumar Buyya Maria A. Rodriguez. Container-based cluster orchestration systems: a taxonomy and future directions. https://arxiv.org/pdf/1807.06193.pdf, 2018. tikrinta 2021-05-10.

[MB18] Abraham Marín-Pérez ir Daniel Bryant. *Continuous Delivery in Java: Essential Tools and Best Practices for Deploying Code to Production*. O'Reilly Media, Sebastopol, United States of America, 2018. tikrinta 2021-04-20.

[Moe19] Julia Rubin Moein Owhadi-Kareshk Sarah Nadi. Predicting merge conflicts in collaborative software development. `https://arxiv.org/pdf/1907.06274.pdf`, 2019. tikrinta 2021-05-10.

[Oma18] Peter Martinek Omar Al-Debagy. A comparative review of microservices and monolithic architectures. `https://arxiv.org/pdf/1905.07997.pdf`, 2018. tikrinta 2021-05-10.

[Pat20] David J. Pearce Patrick Lam Jens Dietrich. Putting the semantics into semantic versioning. `https://arxiv.org/pdf/2008.07069.pdf`, 2020. tikrinta 2021-05-10.

[Pre] Tom Preston-Werner. Semantic versioning 2.0.0. `https://semver.org/`. tikrinta 2021-05-10.

[Rez20] Laurie Williams Rezvan Mahdavi-Hezaveh Jacob Dremann. Software development with feature toggles: practices used by practitioners. `https://arxiv.org/pdf/1907.06157.pdf`, 2020. tikrinta 2021-05-10.