LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

NEPĀRTRAUKTĀS INTEGRĀCIJAS KONCEPTS BANKAS KODOLA SISTĒMĀS.

MAĢISTRA DARBS

Autors: Kārlis Broks
Stud. Apl. Nr. kb17066
Darba vadītājs: Profesors Dr.dat. Jānis Bičevskis

RĪGA 2019

ANOTĀCIJA

Maģistra darbā detalizēti tiek analizēts esošais bankas kodola sistēmu izstrādes process, kas ir novecojis - nenodrošina pienācīgu versiju kontroli, testēšanu un piegādes ātrumu. Darba gaitā autors izskata nepārtrauktās integrācijas konceptu un iespēju to pielietot bankas kodola sistēmu izstrādes procesā. Šīs sistēmas ir būvētas izmantojot specifisku Progress Open Edge tehnoloģiju. Tiek apskatīti nepieciešamie rīki, kā arī aprakstītas ieviešanas vadlīnijas. Darba rakstīšanas gaitā autors jau daļēji ievieš nepatrauktās integrācijas rīkus un pielieto tos bankas kodola sistēmu piegādēm.

Veicot rezultātu analīzi autors secināja, ka konkrētajā situācijā izmantojot nepārtrauktās integrācijas kanālu sistēmas piegāžu uzstādīšanas laiku var samazināt līdz pat 12 reizēm. Procesu automatizēšana noved arī pie ievērojama resursu izmantojuma samazināšanas, būtībā likvidējot manuālu darbību nepieciešamību veidojot būvējumu un to uzstādot kādā no vidēm. Tiek nodrošināta arī pienācīga koda versiju atsekošana. Rezultātā piegādes ir ātrākas, lētākas un drošākas.

Atslēgvārdi: nepārtrauktā integrācija, vienībtesti, automātiskie būvējumi, Open Edge.

ANNOTATION

This Master's thesis analyzes in detail the existing development process of bank's core systems that is outdated – not providing proper version control, testing and delivery times. In thesis author examines the concept of continuous integration and the possibility to apply it in the process of developing bank's core systems. These systems are built using technology that is not so common – Progress Open Edge. The necessary tools are reviewed and the guidelines of the implementation are described. During the writing of the thesis author already partly introduces the tools of continuous integration in bank system deployments.

By analyzing the results, author concluded that in a given situation using continuous integration pipeline delivery time can be reduced up to 12 times. Process automation also leads to a significant reduction in resource utilization, by essentially eliminating the need for manual action when building and deploying new release. Also proper code version tracking is provided. As result deliveries are faster, cheaper and safer.

Keywords: continuous integration, unit testing, automatic builds, Open Edge.

Title: Continuous integration concept in bank core systems.

AUTOREFERĀTS

Autors maģistra darba ietvaros ir veicis literatūras izpēti par nepārtraukto integrāciju, tās pielietošanas praksēm un iespējām to izmantot bankas kodola sistēmu izstrādes procesā. Darba ietvaros autors detalizēti izpētīja esošo programmatūras izstrādes procesu, tikās ar sistēmu izstrādātājiem, kā arī ārējiem konsultantiem.

Lai iegūtu papildus informāciju par iespējām veikt vienībtestus un automātiskos būvējumus, ņemot vērā specifisko tehnoloģiju, autors arī tikās ar tehnoloģijas Open Edge pārstāvjiem.

Darba sākumā autors detalizēti apraksta esošo situāciju, sistēmas arhitektūru, izmantotās tehnoloģijas un rīkus. Tālāk tiek aprakstīts nepārtrauktās integrācijas koncepts, izvēlēti rīki, kā arī aprakstītas to izmantošanas vadlīnijas veidojot nepārtrauktās integrācijas kanālu.

Autors uzskata, ka izmantojot nepārtrauktās integrācijas konceptu var uzlabot esošo izstrādes procesu un panākt resursu ietaupījumu. Darba vadlīnijas ir nodotas uzņēmuma atbildīgajām personām un ir pieņemts lēmums ieviest šo konceptu ikdienas bankas kodola sistēmu izstrādes procedūrās.

Darbā autors ir izmantojis grāmatas un interneta avotus, no kuriem daļa ir arī apskatīto rīku, tehnoloģiju oficiālā dokumentācija. Darbā ir sniegtas atsauces uz izmantoto literatūru.

SATURS

APZĪMĒJUMU SARAKSTS

Apzīmējums	Skaidrojums
IT	Informācijas tehnoloģijas
DevOps	Izstrādes kultūra, kas apvieno programmatūras izstrādi (Development) un darbināšanu (Operations).
TDD	Testu vadīta izstrāde
PAS	Progress lietotņu serveris
CI	Nepārtrauktā integrācija
CI pipeline	Nepārtrauktās integrācijas kanāls (ietver sevī būvējuma sagatavošanu piegādei)
Build server	Serveris uz kura notiek programmatūras būvēšana
Back-end	Servera lietojumprogramma
Front-end	Klienta lietojumprogramma
SSH	Secure Shell - tīkla protokols, kurš nodrošina drošu datu apmaiņu starp divām tīkla ierīcēm.
DB	Datubāze
HTML	Hiperteksta iezīmēšanas valoda
JSON	JavaScript objektu notācija
API	Lietojumprogrammas saskarne
Vienībtests	Vienībtest ir tests, ko pielieto mazākai atdalāmai programmas daļai
UAT	Lietotāja akceptēšanas tests - programmatūras pēdējās pārbaudes, kas pārliecinās, ka produkts strādā kā tas ir līgumā paredzēts.
Integrācijas testi	Testi, kuros tiek apvienotas individuālas programmatūras daļās un pārbaudīts vai tās kopā strādā kā tas paredzēts.
Regresa testi	Testi, kuros pārliecinās, ka, pēc izmaiņu veikšanas, iepriekš izstrādātā programmatūras funkcionalitāte joprojām strādā kā tas ir paredzēts.
Dūmu testi	Programmatūras testi, kas ir paredzēti, lai pārbaudītu vai programmatūras svarīgākās funkcijas strādā korekti.
HTTP	Hiperteksta transporta protokols
Agile	Spējā izstrāde

PC	Personālais dators
CPU	Centrālais procesors.
RAM	Brīvpiekļuves atmiņa, kura kurā uzglabātām programmatūru instrukcijām ir tieša piekļuve no centrālā procesora.
Code review	Koda pārskatīšana
Branching	Programmatūras versiju zarojums versiju kontroles sistēmā
RHEL	Red Hat Enterprise Linux
Skripts	Instrukciju virkne, kas izpildās, lai veiktu kādu noteiktu darbu.
Programmatūras metrika	Kāds projekta standartizēts mērījums programmatūrai vai procesiem, kuriem ir iespējams izsekot.
Repozitorijs	Centrālā vieta, kurā organizēti tiek veidots un uzturēts datu sakopojums.
Git	Versiju kontroles sistēma, kas ir paredzēta izmaiņu izsekošanai datoru datnēs starp vairākiem cilvēkiem.

IEVADS

Mūsdienās informācijas tehnoloģiju laikmetā programmatūras izstrādei ir pieejami dažādi rīki un metodes. Neskatoties uz arvien pieaugošo informācijas tehnoloģiju produktu piedāvājuma apjomu, pieprasījums krietni pārsniedz piedāvājumu. Līdz ar to tiek meklēti veidi kā arvien ātrāk, kvalitatīvāk un efektīvāk izmantot resursus, lai apmierinātu šo pieprasījumu.

Pēdējos gados lielu popularitāti ir ieguvusi izstrāddarbināšanas (DevOps) metodika. Viens no šīs metodikas elementiem ir nepārtrauktā integrācija. Nepārtrauktā integrācija nodrošina ātrāku un drošāku programmatūras papildinājumu sagatavošanu, bet ietver sevī arī daudzus izaicinājumus. Lai to veiksmīgi izmantotu ir nepieciešams apzināt esošo situāciju un izvērtēt jomas, kurās ir nepieciešami uzlabojumi.

Ņemot vērā bankas kodola sistēmu ievērojamo vecumu un specifisko tehnoloģiju lietojumu, lai ieviestu nepārtraukto integrāciju, ir jāizpilda vairāki priekšnoteikumi.

Darba mērķis ir izpētīt nepārtrauktās integrācijas konceptu un izvērtēt tā pielietošanas iespējas saistībā ar Open Edge tehnoloģijām un sagatavot vadlīnijas šo izmaiņu veikšanai.

Darbs sastāv no 4 daļām. Pirmajā daļā tiek apskatīta esošais bankas kodola sistēmu izstrādes process. Otrajā daļā vispārīgi tiek apskatīts nepārtrauktās integrācijas koncepts un izvēlēti konkrēti rīki kas tiks izmantoti šī koncepta ieviešanai. Trešajā daļā tiek apskatīts ieviešanas process un detalizēti aprakstītas ieviešanas vadlīnijas. Ceturtajā daļā tiek analizēti pētījuma rezultāti un salīdzināti raksturlielumi ar vēsturiskajiem datiem, tiek apskatīti nepārtrauktās integrācijas ieviešanas ieguvumi vairākās jomās – piegāžu ātrums, resursu izmantojums un piegāžu kvalitāte.

1. ESOŠĀS SITUĀCIJAS APRAKSTS

Šajā nodaļā darba autors aprakstījis esošo situāciju finanšu pakalpojuma sniedzēja (bankas) kodola sistēmu uzbūvē, kā arī esošo piegāžu mehānismu. Kodola sistēmas spēlē ārkārtīgi nozīmīgu lomu bankas darbības nodrošināšanā. Tā kā bankas galvenie ienākumi veidojas no depozītiem un kredītiem, ir nepieciešams, lai šis sistēmas darbotos korekti un neradītu zaudējumus/neērtības ne bankai, ne klientiem. Šajā nodaļā apskatītā bankas kodola sistēma "Platon". Šī sistēma nodrošina tādas funkcijas kā – maksājumu apstrāde, kredītu, depozītu, karšu, kontu, grāmatvedības un citu pakalpojumu uzturēšana.

1.1. Sistēmas arhitektūra

Šajā apakšnodaļā aprakstīti galvenie sistēmas arhitektūras raksturlielumi.

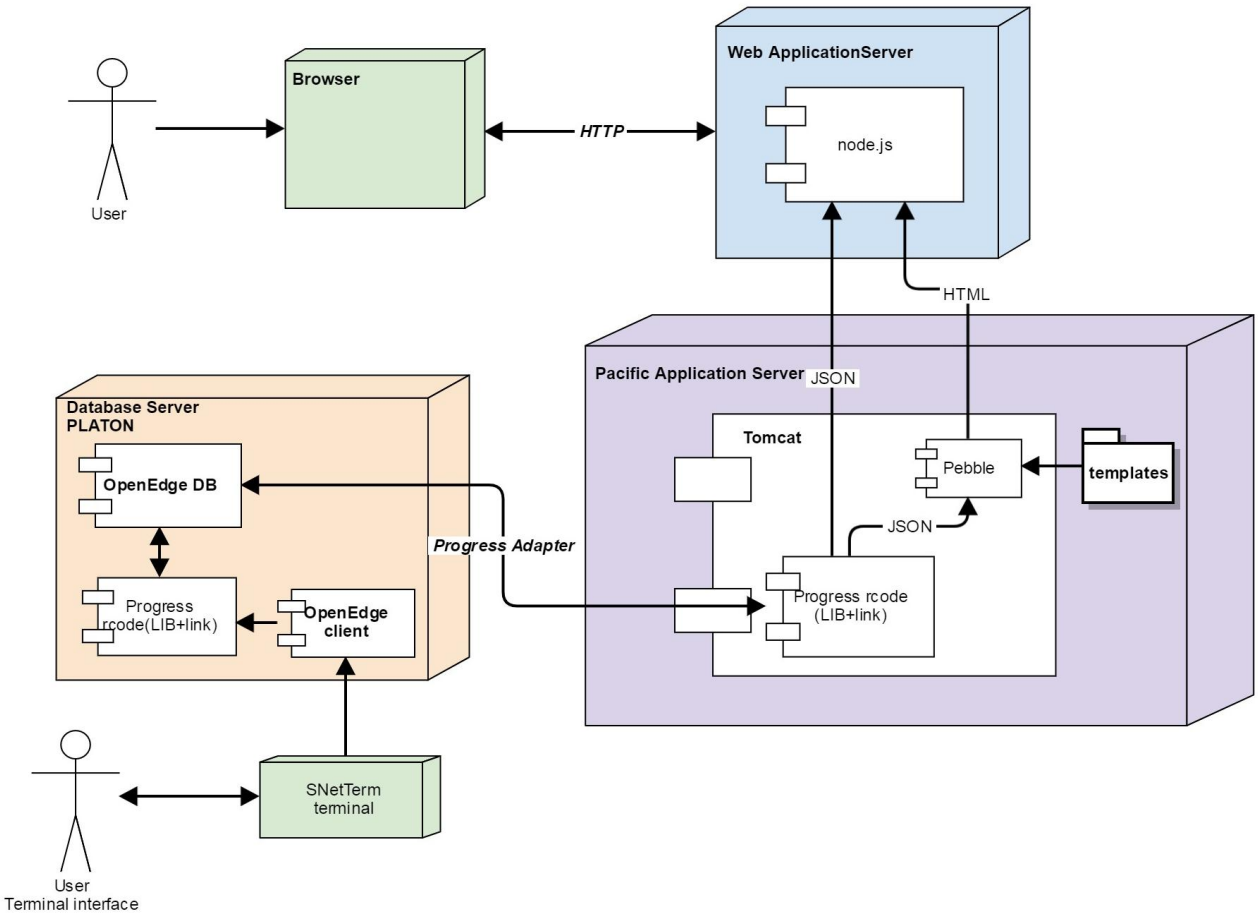
"Platon" ir divu līmeņu arhitektūrā (back-end, front-end) būvēta sistēma, kas sastāv no:

1. Back-end daļa:
    - Datu bāzes vadības sistēma – OpenEdge DB
    - Pacific Application Server (PAS) for OpenEdge - tīmekļa lietotņu serveris: Apache Tomcat, Progress, Pebble, JSON
2. Front-end daļa:
    - Lietotāja saskarne – node.js, HTML, JSON
    - SNetTerm termināla saskarne – Progress

Datubāzes un lietotņu serveri izvietoti uz dažādiem serveriem (fiziskajām un viruālajām mašīnām) *1.1. att* attēlo šo komponenšu izvietojumu.

Kā var redzēt no attēla, Open Edge programmatūras kods ir izvietots, gan uz datubāzes gan aplikāciju servera mašīnām, tas ir tādēļ, ka ar sistēmu ir iespējams strādāt no divu veidu saskarnēm, kas sasvā starpā lielākoties pārklājas:

1. Open Edge character saskarne (SNetTerm termināla saskarne) – vēsturiska saskarne, kurā vadība notiek navigējot tikai ar klaviatūras kombinācijām, bet kuras darbība pieredzējušam lietotājam ir ārkārtīgi parocīga un ātra.
2. OSIRIS web saskarne – jaunāka saskarne, kas daļēji pārklāj character saskarnes funkcionalitāti, šī saskarne ir vieglāk apgūstama un ir krietni intuitīvākā kā vēsturiskā character saskarne.



1.1. att. **Bankas kodola sistēmas "Platon" arhitektūra**

Progress rcode ir OpenEdge Progress avota koda nokompilētās bibliotekas, tās ir cieši integrētas ar Open Edge datubāzi izmantojot Progress savienotāju.

Front-end web saskarnei tiek izmantots Pebble šablonu ģenerators, uz tīmekļa serveri tiek nodots HTML šablons, kas tiek aizpildīts ar datiem kurus progress bibliotēka nosūta uz tīmekļa serveri JSON formātā. Tīmekļa servera pusē tiek konstruēta lietotne no saņemtās informācijas.

Platon sastāv no vairākiem savstarpēji saistītiem moduļiem:

- Statistikas modulis
- Transakciju ģenerators
- Līgumi
- Money Market
- Karšu modulis
- Depozīti
- Konti

- Maksājumi
- Nodrošinājumu modulis
- Kredītu modulis
- OSIRIS (WEB saskarne)

Platon izvietots 5 vidēs:

- Izstrādes vide – tiek izstrādāts programmas kods, sagatavotas relīzes, pārbaudīta koda kvalitāte (code review).
- Testa vide – relīzes tiek testētas, notiek lietotāju apmācības, lietotāju akcepttesti, integrācijas testi.
- Testa vide (2) – produkcijas datu iepriekšējās diena kopija ar maskētiem lietotāju personas datiem. (scrambeled data)
- Pirms produkcijas (staging) vide – notestētā relīzes pakotne tiek uzstādīta un notiek pēdējie integrācijas testi. Tiek notestēta sistēmas pamatfunkcionalitāte. Pirms produkcijas vides konfigurācija, programmatūras kods un datubāzes struktūra pilnībā atbilst produkcijas videi.
- Produkcijas vide:
    - Front-end pieejama gala lietotājam – bankas darbiniekam, kas strādā ar bankas pakalpojumu piešķiršanu.
    - Back-end veic dažādus aprēķinus, apstrādā transakcijas, maksājumus utt.

1.2.Izmantotās tehnoloģijas

1.2.1 OpenEdge ABL

OpenEdge Advanced Business Language – OpenEdge ABL ir biznesa lietotņu izstrādes valoda pazīstama arī kā "Progress" izstrādes valoda. Programmēšanas valodas izstrādātājs un attīstītājs ir "Progress Software Corporation" (PSC). Valoda tiek klasificēta kā ceturtās paaudzes programmēšanas valoda, kas izmanto sintaksi līdzīgu Angļu valodai, lai vienkāršotu programmatūras izstrādi. Valoda tika sauta par PROGRESS vai Progress 4GL līdz tās 9. versijai, bet 2006. gadā tika nosaukta par OpenEdge ABL.

OpenEdge ABL palīdz izstrādātājiem izstrādāt lietotnes izmantojot tajā integrēto relāciju datubāzi un programmēšanas rīku. Šajā valodā izstrādātās lietotnes ir viegli pārnesamas starp dažādu infrastruktūru vidēm un iespējo piekļuvi dažādiem populāriem datu avotu veidiem. Lai to darītu nav nepieciešamības apgūt pamata datu piekļuves metodes. Tas nozīmē, ka produkta gala lietotājs var nebūt kursā par pamatā esošo arhitektūru.

Kombinējot ceturtās paaudzes programmēšanas valodu un relāciju datubāzi, OpenEdge ABL palīdz izmantot ātro lietojumprogrammu izstrādes modeli (Rapid application development) programmatūras izstrādē. Programmētājs un pat galalietotājs var veikt strauju prototipēšanu izmantojot izstrādes vides integrētos grafiskos lietotāja saskarni (GUI) rīkus.[1]

1.2.2 OpenEdge datubāzes dzinis

OpenEdge lietotņu izstrādes vidē ir integrēta arī datubāzes vadības sistēma. Šis datubāzes dzinis nav tik populārs kā citi tirgū pieejamie, iespējams, tāpēc, ka atbalsta tikai ABL programmēšanas valodu un tam ir tikai komerciālā licence. Lai arī ir atbalstīts SQL, tā funkcionalitāte ir ierobežota un pārsvarā tiek izmantota specifiska vaicājumu valoda. (sk. 1.1. tabula.)[2]

1.1. tabula

Vienkāršs SQL un OpenEdge vaicājumu valodas salīdzinājums

Vaicājumu valoda	Sintakse
SQL	**SELECT** * **FROM** customer;
OpenEdge/ABL	FOR EACH customer NO-LOCK: DISPLAY customer. END.

To atbalsta ierobežots skaits operētājsistēmu:

- AIX
- HP-UX
- Linux
- Solaris
- Windows

1.2.3 PAS – Pacific lietojumprogrammu serveris

Pacific Application Server (PAS) ir kodola lietojumprogrammu serveris, kas balstīts uz Apache Tomcat, tas ir pamats OpenEdge lietojumprogrammu serveriem kā arī citiem progress produktiem. PAS ir īpaši pielāgots, lai atbalstītu OpenEdge lietojumprogrammas. Šis serveris tiek izmantots kā web back-end serveris.[3]

1.2.4 Pebble

Pebble ir java veidņu veidošanas dzinis, kas izceļas ar mantošanas funkcionalitāti, bagātīgu iebūvēto tagu un filtru kompleksu un viegli lasāmu sintaksi. Pebble ir iebūvēta drošības automātiska atlase un tajā ir integrēts internacionalizācijas atbalsts.[4]

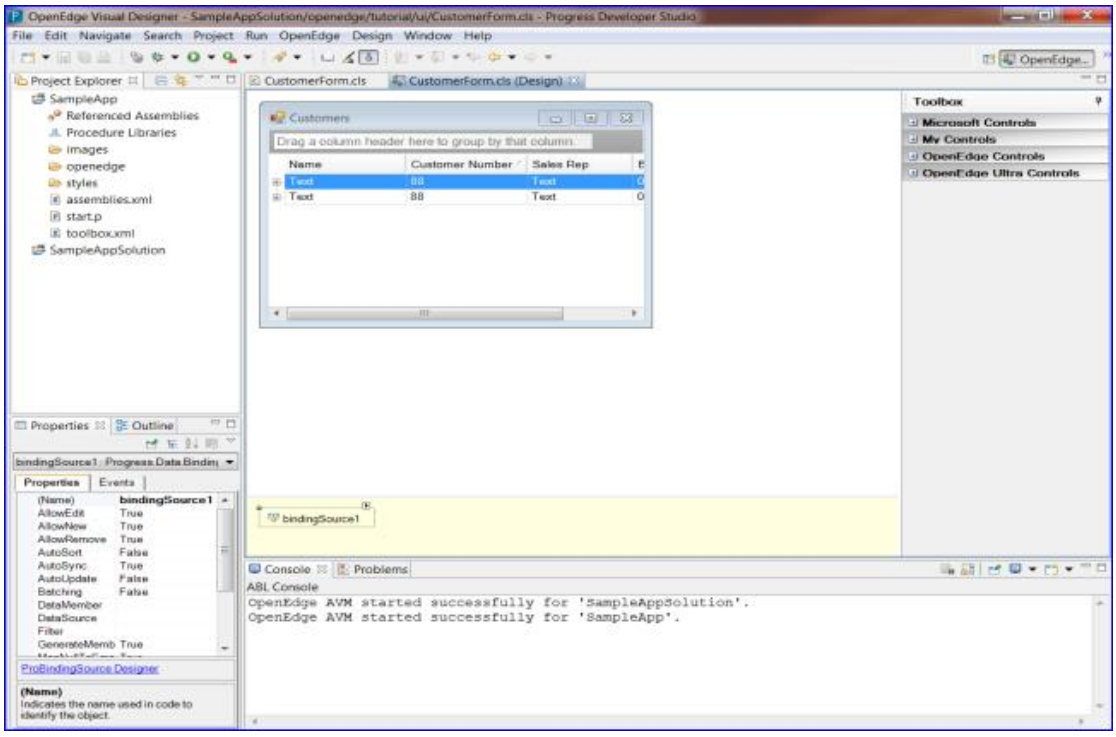
1.2.5. Node.js

Node.js ir atvērtā koda starpplatformu Java Script run-time vide, kas izpilda JavaScript kodu ārpus pārlūka. JavaScript galvenokārt tiek izmantots klienta puses skriptu rakstīšanai, kur JavaScript ir iebūvēts tīmekļa vietnes HTML un darbināts izmantojot klienta puses pārlūkā iebūvēto JavaScript dzini. Node.js ļauj izstrādātājiem izmantot JavaScript lai rakstītu komandrindas rīkus un servera puses skriptus, lai izveidotu dinamiskas tīmekļa vietnes pirms lapa ir nosūtīta uz lietotāja pārlūku.[5]

1.3. Izstrādē izmantotie rīki

1.3.1. Progress Developer Studio

Vispopulārākais OpenEdge ABL izstrādes rīks ir Progress Developer Studio (sk. *1.2. att.*), tas ir eclipse bāzēts izstrādes rīks, kurā var izstrādāt, atkļūdot, testēt un uzstādīt OpenEdge ABL lietotnes.



1.2. att. **Progress ABL Developer Studio**

1.3.2. ABLUnit

ABLUnit ir testēšanas ietvars ABL programmām. Tas ir līdzīgs JUnit XUnit bāzētiem testēšanas ietvariem. ABLUnit paplašinājums Progress Developer studio palīdz rakstīt un darbināt atkārtojamus testus un testu kompleksus.

Šis ietvars palīdz:

- Identificēt kļūdas kodā
- Rakstīt modulāru un brīvi savienotu kodu
- Testēt koda daļas

ABLUnit ietvars atbalsta divu veidu testu scenārijus:

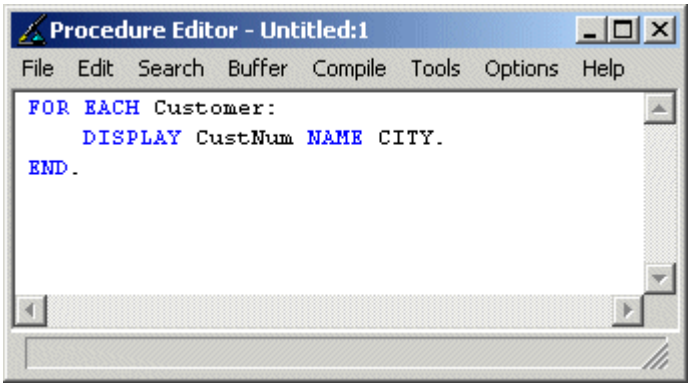
- Testa klase un testa kompleksa klase (.cls fails)
- Testa procedūra un testa kompleksa procedūra (.p fails)

ABLUnit testēšanas ietvars ietver sekojošas komponentes:

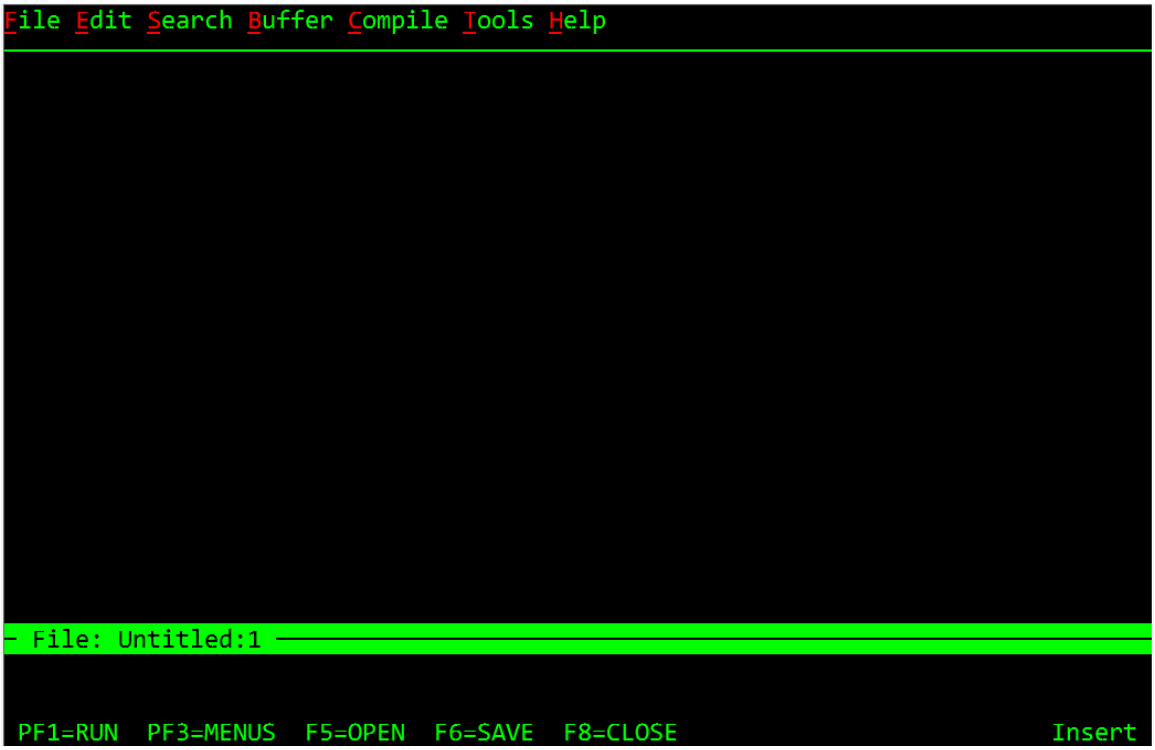
- Anotācijas testa gadījumu atzīmēšanai
- Apgalvojumi par paredzamajiem rezultātiem
- Testa kompleksi, lai viegli organizētu un palaistu testa scenārijus
- Testu vai testu kompleksa darbinātāji no komandrindas vai izpildīšanas konfigurācijas
- Rezultātu skats, kas attēlo un analizē testu rezultātus.[6][20]

1.3.3. OpenEdge procedure editor

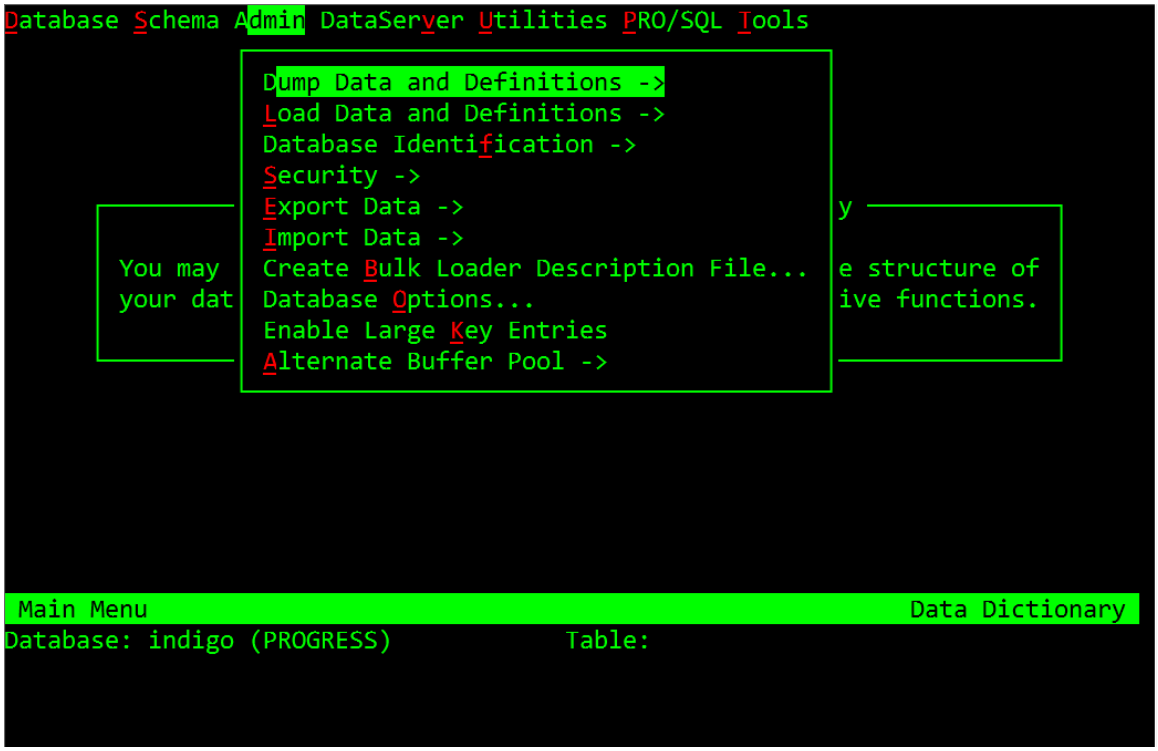
Procedūru redaktors ir OpenEdge ABL koda redaktors. (sk. *1.3., 1.4. att.*) Ar šo redaktoru var izstrādāt un kompilēt OpenEdge programmu kodu. Šis redaktors sevī ietver arī vairākus citus noderīgus rīkus, kā datu vārdnīcu un OS shell un lietotņu kompilatoru. Izmantojot datu vārdnīcas rīku ir iespējams darboties ar datubāzes pieslēgumiem, apskatīt, mainīt datubāzes struktūru, importēt, eksportēt datus kā arī veikt citas darbības ar datubāzi.(sk *1.5. att.*)



1.3. att. **OpenEdge procedure editor windows vidē**



1.4. att. **OpenEdge procedure editor UNIX vidē**



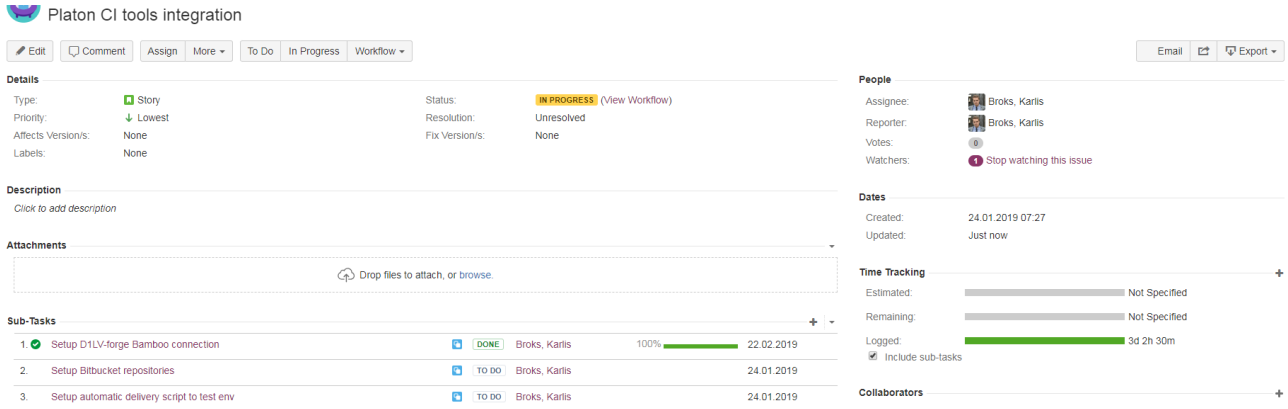
1.5. att. **OpenEdge procedure editor data dictionary skats**

1.3.4. JIRA

Jira ir izstrādes uzdevumu izsekošanas produkts, ko ir izstrādājis programmatūras izstrādes kompānija Atlassian. Tas ļauj veikt kļūdu izsekošanu un ātru projektu vadību. Jira ir vispopulārākais spējās izstrādes komandu izmantotais programmatūras izstrādes rīks pasaulē.

Tam ir pieejamas vairāk nekā 3000 dažādas pievienojumprogrammas (addons), viena no populārākajām izstrādes procesam ir manuālās un automātiskās testēšanas vadības lietotne - Xray.

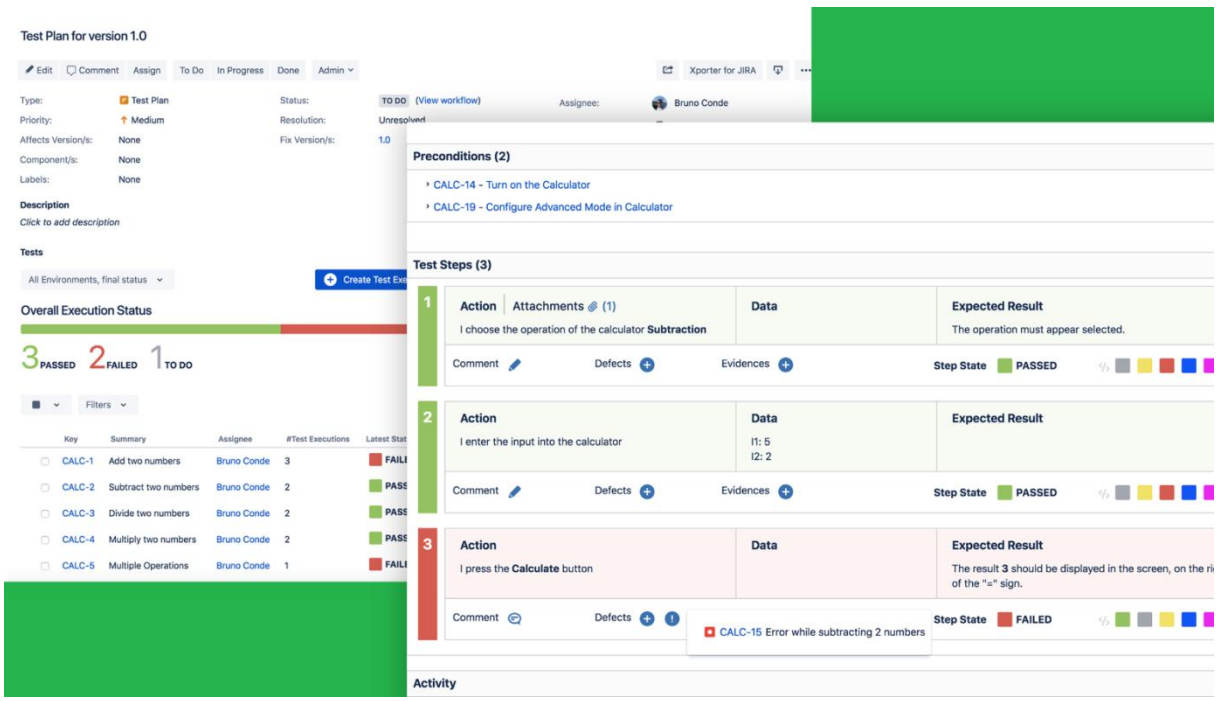
Jira funkcionalitāte ir ārkārtīgi plaša, bet šī darba kontekstā tiks apskatīta tikai funkcionalitātes daļa, kas saistīta ar nepārtraukto integrāciju. *1.6. attēlā* redzams Jira saskanes piemērs.[7]



1.6. att. **Jira saskarne**

1.3.5. Jira Xray

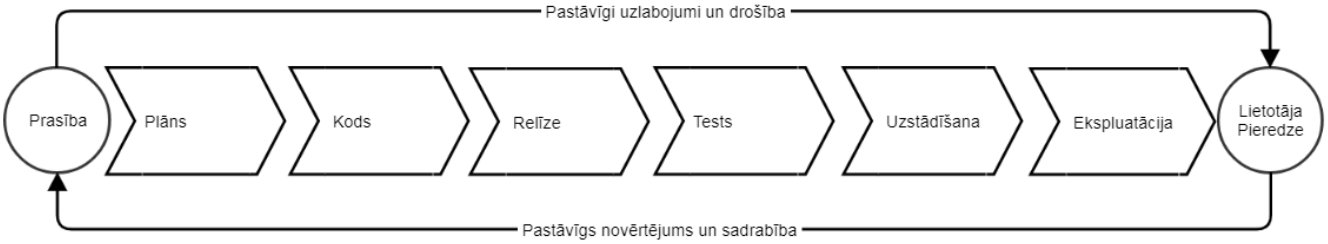
Xray ir vispopulārākā manuālās un automātiskās testēšanas vadības lietotne Jira paplašinājumu katalogā. Xray palīdz pārvaldīt testus kā Jira uzdevumus, kā arī plānot izpildīt un integrēt testus. Šī lietotne nodrošina plašu atskaišu daudzveidību kā arī palīdz nodrošināt prasību pārklāšanu. (sk. *1.7. att.*)[8]



1.7. att. **Xray JIRA saskarne**

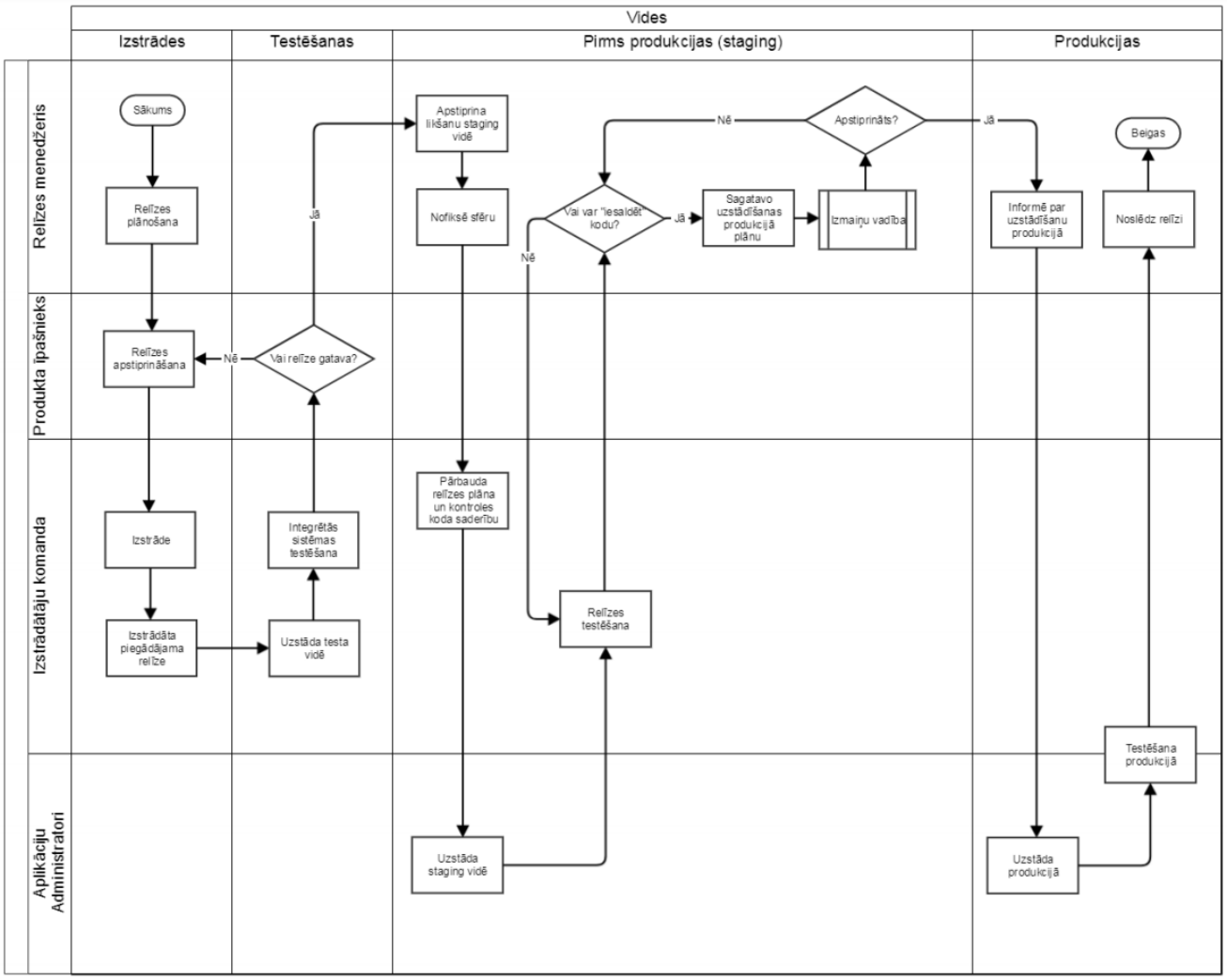
1.4. Relīžu procedūra

Šajā apakšnodaļā aprakstīts spēkā esošais bankas relīzes process. Bankas relīžu process ir konkrēti definēts relīžu procesa procedūru dokumentācijā. Vienkāršots relīzes izstrādes procedūras apraksts redzams *1.8. att.* Pieprasījums pēc jaunas funkcionalitātes vai kļūdas labojuma parasti pienāk no biznesa puses. Tiek izvirzīta prasība, tiek sagatavots relīzes plāns, veikta izstrāde, sagatavota relīze un veikta testēšana. Pēc veiksmīgas testēšanas relīze tiek uzstādīta produkcijā un nodota gala lietotāja rīcībā.



1.8. att **Vienkāršota relīzes procesa aktivitāšu shēma**

Par relīzes procesa ievērošanu, piemērošanu ir atbildīgs relīzes menedžeris, tieši viņš norīko ar relīzi saistītās aktivitātes atbildīgajām personām un seko līdzi relīzes implementēšanai produkcijā. Detalizēta relīzes procesa aktivitāšu shēma redzama *1.9. att.*



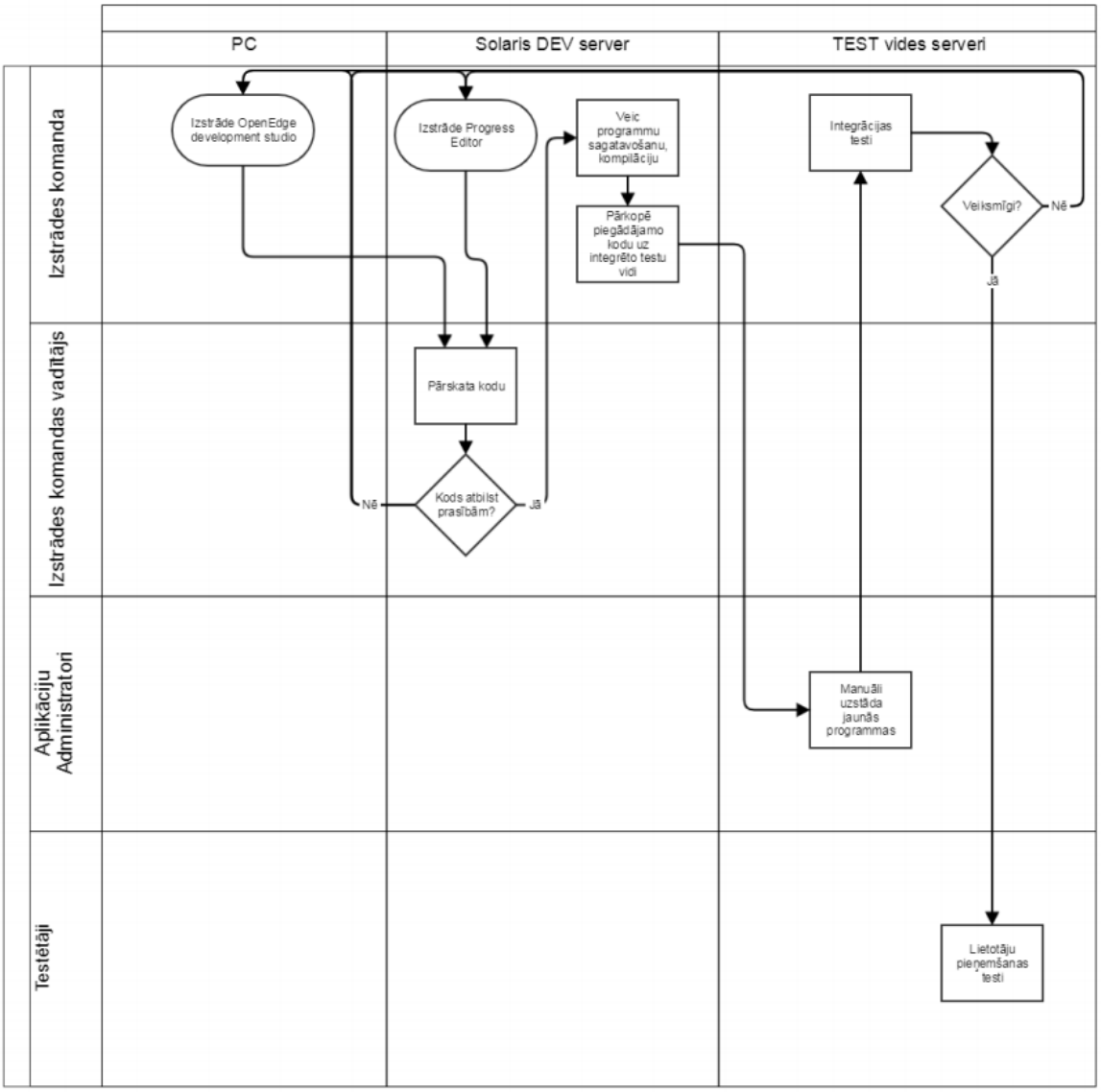
1.9. att **Detalizēta relīzes procesa aktivitāšu shēma**

1.5 Izstrādes procedūra

Šajā apakšnodaļā ir aprakstīta procedūra, kas darbojās šī darba rakstīšanas sākumā.

Izstrādātāji galvenokārt izmanto divas izstrādes vides un rīkus – personālos datorus (izmantojot OpenEdge Development Studio) un UNIX Solaris darbamašīnu (izmantojot OpenEdge procedure editor). Kad izstrādātāju komanda saņem izstrādes uzdevumu, funkcionalitāte vai labojums tiek izstrādāts un nodots izstrādes vadītājam uz koda pārskatīšanu (code review), parasti vadītājs tiek informēts epasta formā un kods tiek kopēts attiecīgajā direktorijā izstrādes vidē. Pēc koda pārbaudes izstrādātāju komanda saņem apstiprinājumu sagatavot izstrādes pakotni un pārkopē to uz integrēto testu vidi. Saņemot pieprasījumu aplikāciju administratori uzstāda attiecīgās programmas testa vidē, kur izstrādātāji attiecīgi veic integrācijas testus. Ja testi izpildās veiksmīgi, programmatūra tiek nodota testēšanai testētājiem. Komunikācija pārsvarā notiek e-pastos un JIRA komentāros. Programmatūras uzstādīšana parasti notiek manuāli pārkopējot failus. Bieži rodas problēmas

ar nekorektu failu pārkopēšanu kā arī ar nekorektām faila piekļuves tiesībām. Esošais izstrādes procedūras modelis attēlots *1.10. attēlā*. Ir attēlotas iesaistīto lomas, kā arī instances kurās notiek konkrētās darbības.



1.10. att **Esošais programmatūras izstrādes process.**

Manuālā programmas būvēšana

Vislielākā apzinātā problēma, kas saistīta ar izstrādes procesu ir manuālā programmas būvēšana, tā kā Platon koda kompilēšanai un bibliotēkas izveidošanai ir nepieciešams atbilsts daudzām atkarībām, šis var izrādīties grūts uzdevums. Sistēma sastāv no vairāk kā 6000 programmām, kas ir savstarpēji saistītas, bet visgrūtākais faktors kompilēšanas procesā ir tas, ka kompilatoram ir jābūt pieslēgtam dažādu datubāžu kombinācijām. Tas ir nepieciešams, jo kompilators tabulas un laukus, kas minēti kodā meklē attiecīgajās datubāzēs. Šis ir manuāls

process, jo datubāžu skaits pret kurām ir jākompilē ir vairāk kā 10 un ir nepieciešamas dažādas to kombinācijas.

1.6. Problēmas formulējums

Pastāvot pašreizējajai programmatūras izstrādes un integrācijas procedūrai, tiek patērēti ievērojami resursi, lai manuāli sagatavotu lietotni darbībām dažādās vidēs. Sakarā ar dažādu apjomīgu funkcionalitāšu ieviešanu ir nepieciešamas arvien jaunas vides lietotnes testēšanai, kā arī vide kurā pārbaudīt kļūdu labojumus. Tā kā vižu un piegāžu apjoms pieaug, manuāla aplikācijas būvēšana un integrācijas vairs nav racionāla, arvien vairāk un vairāk laika tiek patērēts, lai meklētu risinājumus problēmām, kuru cēlonis ir nekorekti uzbūvēta, uzstādīta, vai konfigurēta lietotne. Izmantojot manuālos piegādes mehānismus nav iespējams sekmīgi atsekot piegādes, kā arī atbilst bankas relīzes procesam.

Manuāli būvējot lietotni, kā arī neizmantojot centralizētus versiju kontroles rīkus vidējais programmas būvēšanas laiks atkarībā no prasībām var sasniegt līdz pat 6 stundām. Tam ir vairāki iemesli:

- Izstrādes vadītājam jāpārbauda būvējamās lietotnes bibliotēkas saturs un konflikti manuāli pirms būvējuma veikšanas
- Ja ir dažādas prasības attiecībā par mērķa vidēm (dažāda datubāzes struktūra, nepieciešama atšķirīga programmu konfigurācija), izstrādes vadītājs apspriežas ar izstrādātājiem, lai noskaidrotu kuras programmas ir jāievieto būvējumā
- Būvējumam nepieciešama mērķa vides struktūra, nav centralizētas un versiju kontrolētas struktūru kombinācijas
- Tikai izstrādes vadītājam ir piekļuves tiesības veikt būvējumu un tā kā darba dienas gaitā ir daudz tekošo darbu, ir liela iespējamība pieļaut kļūdu vai aizkavēt būvējuma veikšanu.
- Būvējumi nav versiju kontrolēti, līdz ar to ir iespējams pieļaut kļūdu būvējuma procesā.

Izmantojot esošo izstrādes metodiku, izmaiņas ir grūti atsekojamas un kļūdu gadījumā ir grūti atgriezties pie strādājošas sistēmas versijas.

2. NEPĀRTRAUKTĀ INTEGRĀCIJA

Šajā nodaļā tiks apskatīts nepārtrauktās integrācijas koncepts, kā arī izvēlēti rīki tā ieviešanai bankas kodola sistēmās.

Nepārtrauktā integrācija ir programmatūras izstrādes prakse, kurā izstrādātāji regulāri veic izmainās versiju kontroles rīkā, pēc tam tiek veikti automātiski programmas būvējumi un darbināti automātiski testi. Nepārtrauktā integrācija parasti apraksta būvēšanas, kompilācijas un integrācijas posmu programmatūras izstrādes procesā un ietver gan komponenšu automatizāciju, gan kulturālo komponenti (mācīšanos integrēt bieži). Galvenie nepārtrauktās integrācijas mērķi ir atrast kļūdas ātrāk, uzlabot programmatūras kvalitāti un samazināt laiku, kas ir vajadzīgs lai apstiprinātu relīzi. [9]

Ieviešanas prasības:

- Izstrādes komandai ir jāraksta automātiski testi katrai jaunajai funkcionalitātei, kļūdas labojumam un uzlabojumam
- Ir nepieciešams nepārtrauktās integrācijas serveris, kas monitorētu galveno repozitoriju un darbinātu automātiskos testus katru reizi, kad kods tiek nodots galveno repozitorija zaru.
- Izstrādātājiem jāapvieno (merge) izmaiņas pēc iespējas biežāk, vismaz reizi dienā

Ieguvumi:

- Mazāk kļūdas tiks piegādātas produkcijā, jo regresijas rezultātā automātisko testu laikā tās tiks atklātas
- Būvēt relīzi ir viegli un visas integrācijas problēmas tiek atrisinātas laicīgi
- Mazāk konteksta maiņa, jo izstrādātāji tiek brīdināti tiklīdz tie salauž būvējumu un var to salabot pirms sāk strādāt pie nākošā uzdevuma
- Testēšanas izmaksas krasi samazinās, nepārtrauktās integrācijas serveris dažu sekunžu laikā var palaist simtiem testu
- Kvalitātes nodrošināšanas komanda tērē mazāk laika testēšanai un var fokusēties uz būtiskiem kvalitātes uzlabojumiem.[10]

2.1. Izvēlētie nepārtrauktās integrācijas rīki

2.1.1. Versiju kontroles sistēma

Par versiju kontroles sistēmu tika izvēlēts GIT, ņemot vērā šis sistēmas funkcionalitāti un vienkāršību un intuitivitāti.

Git ir versiju kontroles sistēma, kas kontrolē izmaiņas failos un koordinē to cilvēku darbu, kas darbojas ar šiem failiem. Šobrīd Git ir vispopulārākais versiju kontroles rīks, kas tiek izmantots programmatūras izstrādē. Git ir decentralizēta versiju kontroles sistēma. Katras lietotnes izstrādei ir nepieciešams izveidot zarošanās modeli.

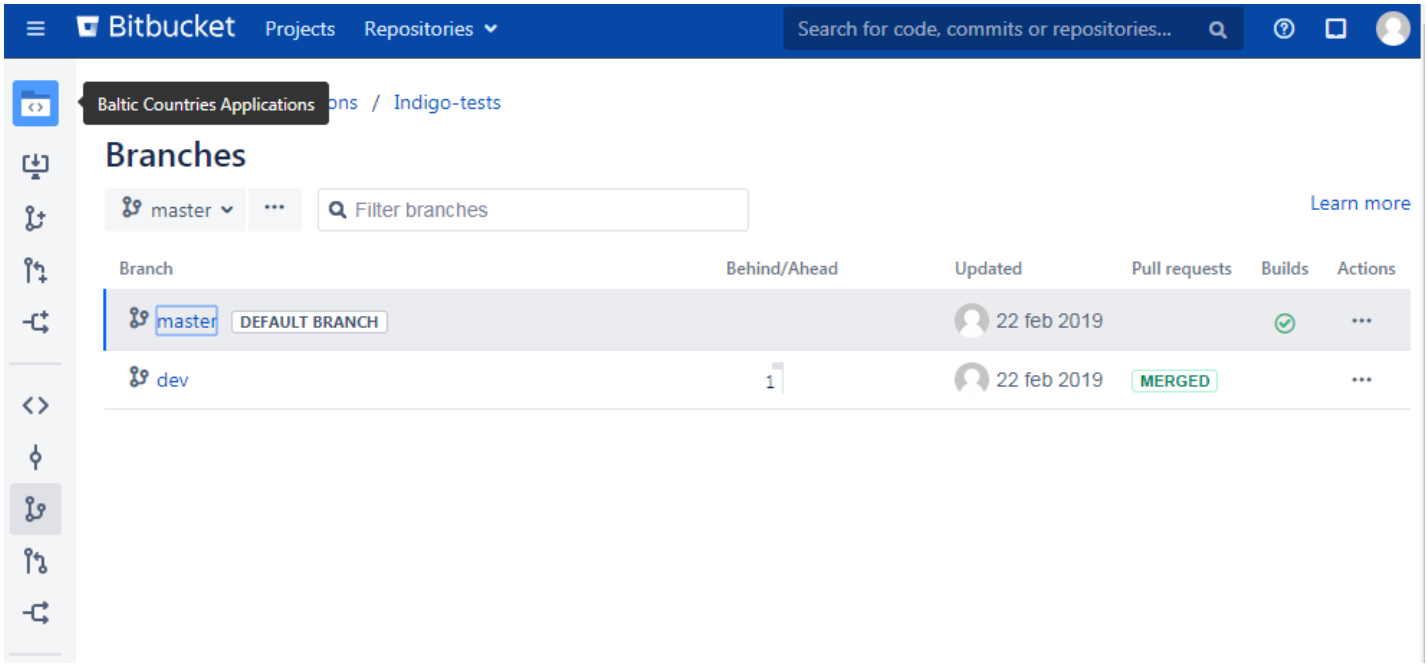
Katra Git jeb repozitorijs ietver pilnu izmaiņu vēsturi un nodrošina vadības iespējas, neatkarīgi no tīkla savienojuma pieejamības vai centrālā servera. Git ir brīvā programmatūra, ko izplata saskaņā ar GNU Vispārējās publiskās licences 2. versiju.

Lai strādātu ar GIT, ir nepieciešams uzstādīt GIT klienta programmatūru. Tas ļauj operētājsistēmas komandrindā izpildīt GIT komandas.[11]

2.1.2. Versiju kontroles serveris

Par versiju kontroles serveri tika izvēlēts Bitbucket, ņemot vērā, ka tas ir viegli savietojams ar citiem programmatūras izstrādes rīkiem un ir viegli pārvaldāmas.(sk. *2.1. att.*)

Bitbucket ir tīmeklī bāzēts versiju kontroles repozitorija hostēšanas serviss, ko izstrādājis Atlassian. Tas ir paredzēts avota kodam un izstrādes projektiem, kas lieto Mercurial vai GIT versiju kontroles sistēmas. Bitbucket ir integrēts ar tādiem citiem Atlassian programmatūras izstrādes rīkiem kā – JIRA, Bamboo, Confluence.



2.1. att **Bitbucket repozitorija saskarne**

Bitbucket ļauj veikt tādas pamata GIT operācijas kā pārskatīt un apvienot kodu, līdzīgi kā tas tiek darīts GitHub. [12]

2.1.3. Nepārtrauktās integrācijas serveris

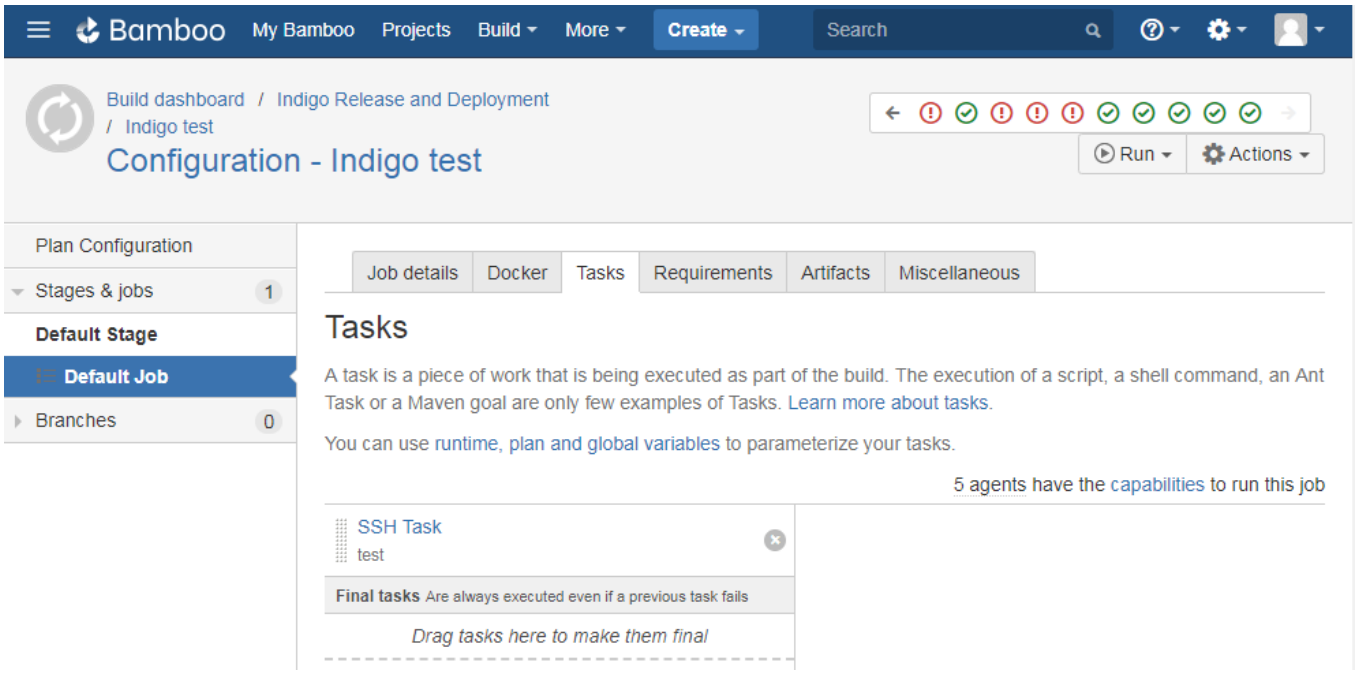
Vispopulārākais nepatrauktās integrācijas serveris ir "Jenkins", bet šajā darbā autos izvēlējās Bamboo, jo tas tāpat kā citi izvēlētie rīki ir Atlassian produkts un ir viegli savietojams ar citiem nepārtrauktās integrācijas rīkiem, kā arī tā lietošana ir relatīvi vienkārša.(sk. *2.2 att.*)

Bamboo ir CI serveris, kas var tikt izmantots, lai automatizētu relīžu vadību, izveidojot automātisku piegādes mehānismu (delivery pipeline). OpenEdge kontekstā Bamboo var tikt izmantots, lai darbinātu Ant skriptus, kas automātiski būvētu un uzstādītu lietotni.

Bamboo sadala aplikāciju hierarhiskā līmenī:

1. **Projekts** – specifiska aplikācija, kas iever sevī vairākus plānus, tas arī savieno savstarpēji saistītas lietotnes.
2. **Plāns** – tajā ir iekļauts visu saskaņā ar to veikto uzdevumu kopums. Plānā ir informācija par:
    - Avota koda krātuve
    - Izstrādātājs izveido būvētājus
    - Kas var apskatīt / mainīt plānu
    - Plānot mainīgos

    Sākotnēji plānam ir viens posms, bet vairākus posmus var izveidot, grupējot dažādus darbus. Visi plāna posmi notiek secīgi.
3. **Fāze** - Sākotnēji tai ir viens darbs, bet zem tās var izveidot vairākus darbus, sadalot uzdevumu sarakstu dažādos darbos. Katram posmam ir jābeidzas, pirms pāriet uz nākamo posmu. Viens no iemesliem, kāpēc tas ir nepieciešams, ir tāpēc, ka reizēm posms var būt atkarīgs no iepriekšējā posma attiecībā uz dažiem artefaktiem.
4. **Darbs** - Darbs vienā posmā notiek paralēli. Piemēram, mēs varam vēlēties, lai funkcionālie un vienību testi darbotos vienlaicīgi. Katru darbu var piešķirt citam aģentam. Darbs tiek piešķirts aģentam atkarībā no tā spējas izpildīt darba prasības. Darbs ņem artefaktus no iepriekšējiem posmiem, rada jaunus, kā arī iezīmē tos.
5. **Uzdevums** - Tas ir mazākais vienības darbs, ko veic aģents. Tā kā visi darba uzdevumi tiek veikti ar vienu aģentu, tie ir jāapstrādā secīgā secībā. Uzdevuma piemēri ietver koda gabala veidošanu, lietojumprogrammas ieviešanu utt.[13]

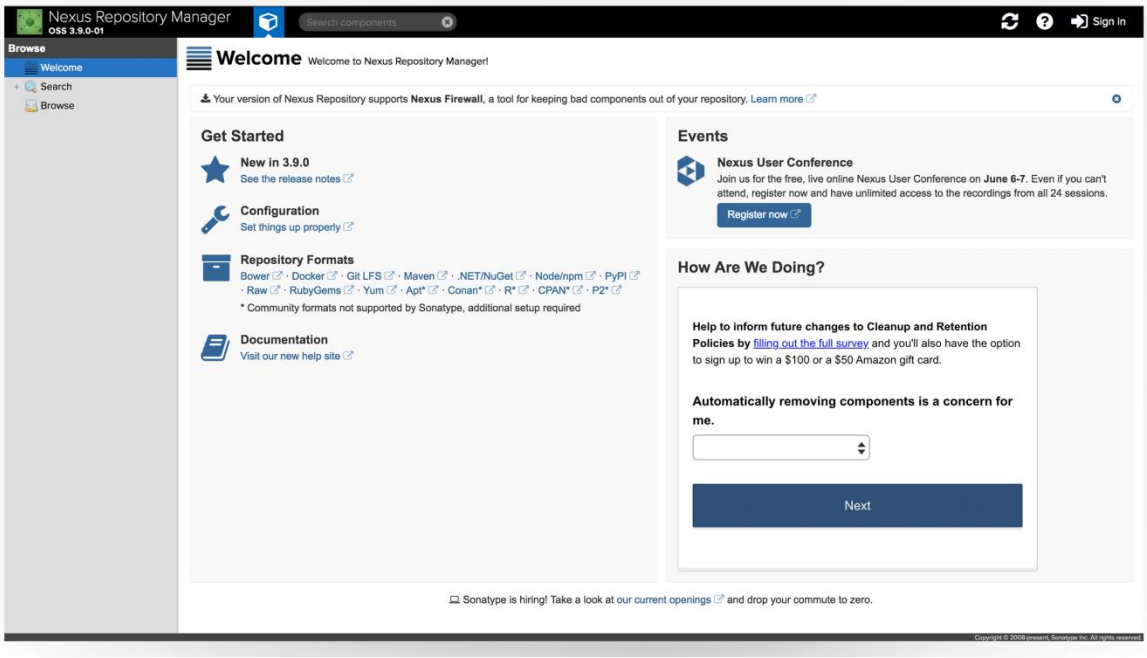


2.2. att **Bamboo saskarne**

2.1.4. Artefaktu pārvaldnieks

Šī darba veikšanai autors izvēlējās Nexus artefaktu repozitoriju, ņemot vērā tā funkcionalitāti un bezmaksas licenci.

Nexus ir repozitoriju pārvaldnieks, tas ļauj pilnvarot, apkopot un pārvaldīt programmas atkarības, kā arī īstenotu versiju kontroli binārajiem vai citiem programmas failiem, kas nav avota faili (programmas kods nav izgūstams). Tas ļauj viegli izplatīt programmatūru. Nepārtrauktās integrācijas kanāls būvē lietojumprogrammu un artefakti tiek publicēti Nexus un pēc tam pieejami citiem izstrādātājiem. Nexus saskarnes piemērs *redzams 2.3. att*. [14]



2.3. att **Nexus repository manager saskarne**

2.2. Nepārtrauktās integrācijas principi

Šajā apakšnodaļā tiks apskatīti nepārtrauktās integrācijas pamatprincipi, kā tos aprakstījis Gatis Jezuns bakalaura darbā – "Pastāvīgā integrācija Java programmatūras izstrādē".[15] Lai arī šis darbs attiecās uz Java tehnoloģijas izmantošanu, tajā minētie nepārtrauktās integrācijas pamatprincipi ir aktuāli arī Open Edge kontekstā.

2.2.1. Viena repozitorija uzturēšana

Gatis Jezuns savā darbā apskatījis, ka programmatūras izstrādes projektos parasti ir iesaistīti dažādu veidu resursi, kuri jāpārvalda, lai uzbūvētu produktu. Lai to visu atsekotu, jāiegulda liels darba apjoms, it īpaši, ja ir daudz iesaistīto pušu. Tāpēc programmatūras izstrādes komandas ir izveidojušas rīkus, lai pārvaldītu resursus. Versiju kontroles sistēmas, pirmkoda pārvaldības rīki, konfigurācijas pārvaldības rīki, repozitoriji un citi ir kļuvuši par neatņemamu sastāvdaļu izstrādes projektos. [15]

Lai veiksmīgi realizētu pastāvīgo integrāciju, ir jāizmanto versiju kontroles repozitorijs,. Versiju kontroles sistēmas uzdevums ir pārvaldīt izmaiņas pirmkodā un citos programmatūras resursos (dokumentos, būvējuma skriptos), izmantojot kontrolētu piekļuvi repozitorijam. Tas nodrošina iespēju piekļūt vecākām resursu versijām, lai vajadzības gadījumā atgrieztos pie tām. Repozitorijā būtu jāievieto pilnīgi visi resursi, kas nepieciešami projekta uzbūvēšanai no nulles. Tie varētu būt būvējumu skripti, konfigurācijas faili un datu bāzu izveidošanas skripti un izmantojamās bibliotēkas. [15]

Katram jaunam izstrādātājam, kas iesaistās projektā, būtu jāvar uzbūvēt visu sistēmu pēc pēdējās versijas iegūšanas. [15]

2.2.2. Būvējuma automatizācija

Process ar kura palīdzību nonāk no pirmkoda līdz strādājošai sistēmai parasti ir sarežģīts un ietver sevī dažādas darbības, tādas kā kompilācija, failu pārvietošana, datubāzes shēmas ielādēšana u.c. Tomēr kā vairums uzdevumu šajā izstrādes daļā to var automatizēt un tas ir jāautomatizē. Būvēšanas procesam jābūt pilnīgi automatizētam, pēc tā uzsākšanas nav nepieciešama nekāda iejaukšanās no lietotāja puses. Tam jābūt arī elastīgam ne vienmēr ir nepieciešams izpildīt pilnībā visu procesu, piemēram, nelielu izmaiņu gadījumā. Procesam jāspēj automātiski noteikt, kādus soļus nepieciešams izpildīt, kā arī jāpiedāvā iespēja to specificēt lietotājam. Sevišķi svarīgi tas ir pie lieliem būvējumiem, kur pilns process aizņem daudz laika. [15]

2.2.3. Būvējuma automātiskie testi

Kā minēja Gatis Jezuns, tradicionāli būvēšana ietver kompilēšanu un citas darbības, kas nepieciešamas izpildāmas programmas iegūšanai. Tas, ka programma izpildās nenozīmē, ka tā darbojas bez kļūdām. Modernās, statiski tipizētās programmēšanas valodas palīdz atrast daudz kļūdu jau pirmkoda rakstīšanas laikā, taču liela daļa no tām paliek nepamanītas. Labs veids, kā ātri un efektīvi atrast kļūdas, ir būvēšanas procesā iekļaut automātiskos testus. Automātiskā testēšana, protams, nav perfekta, taču ar tās palīdzību var atrast tik lielu daudzumu kļūdu, lai automatizācija atmaksātos vairāk kārt. Būvējuma paštestēšanās plaši tiek izmantota ekstrēmajā programmēšanā un testu orientētā izstrādē (test driven development). Abi no šiem paņēmieniem uzsver testu izveidi pirms koda izstrādes. Kad testi gatavi, tiek rakstīts kods, līdz tas iziet testus - šāda veidā testi ne tikai palīdz atrast kļūdas, bet arī apraksta sistēmas prasības un specifiku. Tā ir laba prakse, tomēr nav obligāta pastāvīgās integrācijas gadījumā. [15] [16][20]

Būvējumu var uzskatīt par paštestējošu tikai tādā gadījumā, ja tajā iekļauto automātisko testu izgāšanās nozīmē arī paša būvējuma izgāšanos. Testiem jābūt viegli izpildāmiem no izstrādes vides ar vienas komandas vai pāris peles klikšķu palīdzību, kā arī jāprot attēlot rezultātus skaidrā un vienkāršā veidā. Ātra kļūdu atklāšana nav vienīgā automātisko vienībtestu lietošanas priekšrocība. Vienībtestu rakstīšana palīdz autoram paskatīties uz kodu „no otras puses" – novērtēt izveidotās saskarnes lietošanas ērtumu un uzreiz izlabot radušās nepilnības projektējumā, pirms kāds kolēģis jau paspējis to sākt izmantot. Automātiskie testi

izstrādātājam dod drošības sajūtu izdarīt izmaiņas, nebaidoties par kādiem „blakus efektiem" – izmaiņas vienā sistēmas komponentā var izraisīt kļūdu rašanos citā. [15]

2.2.4. Regulāras izmaiņas repozitorijā

Pēc Gata Jezuna darbā minētā - integrācijas pamatā ir komunikācija. Ar integrācijas palīdzību izstrādātāji paziņo cits citam par veiktajām izmaiņām. Regulāra komunikācija komandas locekļiem ļauj uzzināt par izmaiņām tiklīdz tās parādās. Galvenais nosacījums, lai izstrādātājs drīkstētu ievietot savas izmaiņas repozitorijā, ir tas, ka kodam jābūt pareizam. Tas, protams, ietver testu pareizu izpildi. Pirms jebkuras ievietošanas izstrādātājiem vispirms jāiegūst pēdējā versija, jāatrisina visas nesaderības un jāpārliecinās, ka testi izpildās bez kļūdām. Tikai tādā gadījumā drīkst ievietot savas izmaiņas repozitorijā. Darot to regulāri, izstrādātāji var ātri konstatēt, ja ir radies konflikts starp viņu veiktajām izmaiņām. Lai ātri atrisinātu kādu problēmu, tā ātri jāidentificē. Ja izstrādātāji veic izmaiņas repozitorijā ik pēc dažām stundām, konflikts visdrīzāk tiks konstatēts pēc pāris stundām kopš tā rašanās. Šajā laika posmā izdarīto izmaiņu skaits ir salīdzinoši neliels, līdz ar to vieglāk atrast izmaiņas, kas kļūdu izraisījušas. Jo ilgāk kļūda netiek atrasta, jo grūtāk atrast tās cēloņus. Tā kā būvējums ir paštestējošs, tā izpildes laikā tiek noskaidrotas ne tikai kompilācijas kļūdas, bet arī izpildes laika kļūdas. Regulāra ievietošana panāk to, ka izstrādātājam jāsadala savs darbs mazākos vienumos, kas aizņem pāris stundas darba. Tas palīdz sekot līdzi progresam un rada progresēšanas sajūtu. [15]

2.2.5. Regulāras sistēmas būves

Darbā arī bija minēts, ka tā kā būvējums ir paštestējošs, tad būvējot pie katras izmaiņas tiek pārbaudīta izdarīto izmaiņu pareizība. Ja būvējums izgāžas, uzreiz ir skaidrs, kuras izmaiņas pie tā ir vainīgas. Tas palīdz ātri atklāt problēmas un atrisināt tās. Būvēšanu pie katrām izmaiņām var realizēt divos veidos:

- Izstrādātājs, kas izdarījis izmaiņas, pats manuāli uzsāk integrācijas būvējumu uz tam paredzētas darbstacijas. Sājā gadījumā jābūt kādām mehānismam, kas nodrošina, ka šīs integrācijas laikā neviens cits komandas loceklis neizdara izmaiņas repozitorijā;
- Izmanto integrācijas serveri, kas automātiski detektē izmaiņas repozitorijā un uzsāk būvējumu. [15]

2.2.6. Pēdējie būvējumi ir viegli pieejami

Kā minēja iepriekšminētā darba autors, viena no sarežģītākajām lietām programmatūras izstrādē ir būt pārliecinātam, ka tiek izstrādāta programma, kas atbilst klienta prasībām. Ir grūti pareizi aprakstīt sarežģītu lietu sīkumos, bieži vien klientam ir daudz vieglāk redzēt kaut ko nepilnīgu vai nepareizu un pateikt kas tajā ir jāizlabo. Lai to varētu izdarīt, katram izstrādē iesaistītajam ir jābūt iespējai saņemt pēdējo programmatūras versiju un jāvar to izmantot. Tas dod iespēju to izmantot demonstrācijai, izpētes testiem vai arī vienkārši, lai redzētu notikušās izmaiņas. Izdarīt to ir samērā vienkārši – jāuzglabā pēdējie veiksmīgie integrācijas būvējumi un jāpārliecinās, ka būvējumu uzglabāšanas vieta ir labi zināma visiem. [15]

2.2.7. Komunikācija

Rezumējot, Gata Jezuna darbā tika bija minēts, ka pastāvīgās integrācijas pamatā ir komunikācija - katram iesaistītajam jābūt iespējai apskatīt sistēmas stāvokli un tajā veiktās izmaiņas. Galvenā lieta, kas interesē visus komandas locekļus, ir būvējuma statuss. Ja tiek lietots integrācijas serveris, tad parasti, tas ne tikai var attēlot informāciju par būvējuma statusu, bet arī testu rezultātus un citu ar būvējumu saistītu informāciju. Manuālās integrācijas gadījumā būvējuma izpildītājam jebkādā veidā jāpaziņo komandai par tā statusu. [15]

2.3. Nepārtrauktās integrācijas izmantojuma piemērs

Grāmatā “DevOps Handbook: How to create world class agility, reliability, & security in technology organizations” ir uzskatāmi aprakstīts, kā ar nepārtraukto integrāciju ir iespējams atrisināt daudz problēmas. To pierāda Gerija Grūvera (Gary Gruver) pieredze inženierijas direktora amatā uzņēmuma Hewlett-Packard, “LaserJet Firmware” nodaļā, kas veido aparātprogrammatūru (firmware). [18]

Šajā apakšnodaļā ir apskatīts Rūdolfa Svikla maģistra darba “DevOps ieviešanas IT uzņēmumā” aprakstītais šī pētījuma kopsavilkums. Rūdolfs apraksta, ka grāmatā minētā izstrādes komanda sastāvēja no 400 izstrādātāju, kas atrodas vairākās valstīs. Neskatoties uz lielo izmēru, komanda gadiem ilgi nebija spējīgi piegādāt jaunu funkcionalitāti pietiekami ātri.[19]

Tika izlaisti divi jauni programmatūras izlaidumi gadā, un darbs pārsvarā sastāvēja no koda pārnešanas, lai atbalstītu jaunus produktus. Grūvers aprēķināja, ka aptuveni tikai 5% laika tika patērēti, lai veidotu jaunu funkcionalitāti, jo lielākais laiks tika patērēts veicot koda pārnešanu, detalizētu plānošanu, koda integrācijas veikšanu un manuālu testēšanu.

Tika izvirzīts mērķis daudzkārtīgi palielināt pavadīto laiku uz inovācijām un jaunu funkcionalitāti.

Komanda cerēja to panākt ar:

- Nepārtraukto integrāciju;
- Testu automatizācija;
- Neveiksmīgu testu atkārtošanu uz izstrādātāju darbstacijām;

Viņu nepārtrauktās integrācijas sistēma veica pilnu automātisko vienību, akceptēšanas un integrācijas testēšanu.

Automātiskie testi izveidoja ātru rezultātu iegūšanu, kas ļāva izstrādātājiem apstiprināt, ka nosūtītais kods tiešām strādā. Vienībtesti izpildījās minūšu laikā trīs līmeņos priekš katra iesūtītā koda, kā arī papildus ik pa divām stundām. Ik pa 24 stundām tika veikti pilni regresijas testi, lai pārliecinātos, ka visa sistēma strādā korekti.

Nepārtrauktās integrācijas ieviešanas ieguvumi:

- Veikt kompilēšanu līdz vienai reizei dienā, un vēlāk jau desmit līdz piecpadsmit reizes dienā;
- Nonāca no divdesmit koda iesūtīšanām dienā, ko veica kompilēšanas atbildīgais, līdz vairāk kā 100 koda daļu iesūtīšanai dienā, ko veica programmētāji paši;
- Deva iespēju izstrādātājiem veikt izmaiņas vai pievienot 75 – 100 tūkstošiem koda rindiņu dienā;
- Samazināja regresa testu veikšanas laiku no sešām nedēļām uz vienu dienu.

Aprakstītie biznesa ieguvumi:

- Patērētais laiks uz jaunām inovācijām un funkcijām palielinājās no 5% uz 40%;
- Kopējās izstrādes izmaksas samazinājās par aptuveni 40%;
- Programmu skaits, kas tika izstrādātas palielinājās par 140%;
- Izstrādes izmaksas par katru programmu samazinājās par 78%.

Grūvera pieredze rāda, ka nepārtrauktā integrācija ir viena no vissvarīgākajām pieejām, kas ļauj veikt ātru darba plūsmu. [18] [19]

3. NEPĀRTRAUKTĀ INTEGRĀCIJA BANKAS KODOLA SISTĒMĀS

Šajā nodaļā tiks apskatīts nepatrauktas integrācijas rīku izmantojums automatizējot sistēmas būvējumu un piegādes procesus. Tiks detalizēti aprakstītas autora veiktās darbības ieviešot nepatrauktās integrācijas konceptu bankas kodola sistēmās.

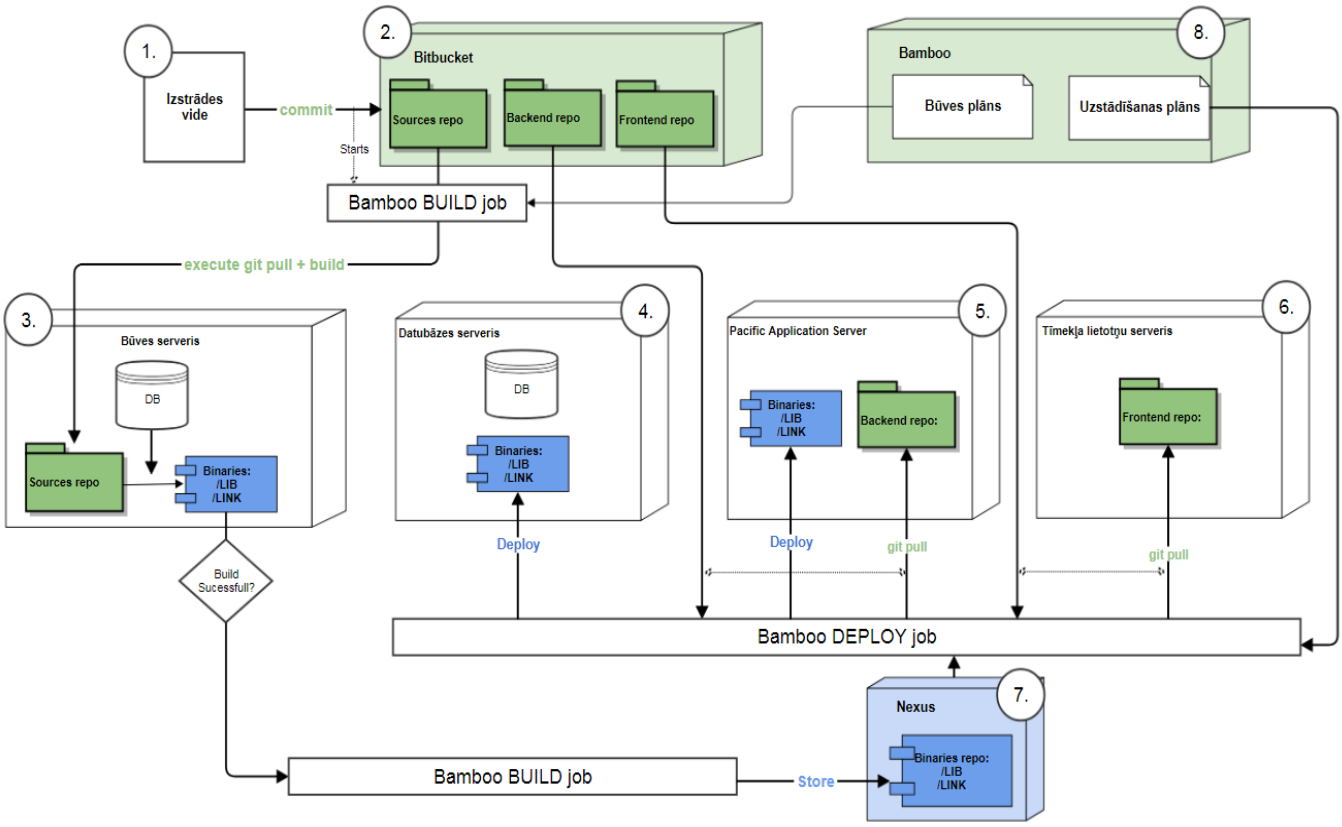
3.1. Integrācijas priekšnoteikumi

Lai varētu sekmīgi ieviest nepārtraukto integrāciju bankas kodola sistēmās, autors ir definējis vairākus priekšnoteikumus:

1. GIT izmantošana – visiem izstrādātāju komandas locekļiem versiju kontrolei ir jāizmanto GIT, autors ir izveidojis apmācības materiālus.
2. Branching tree definēšana – Lai varētu efektīvi izmantot izstrādātāju resursus, kā arī vieglāk kontrolēt izstrādes versijas, autors ir definējis koda versiju zarošanās koku, kas atbilst GitFlow definētajam.
3. JIRA izmantošana – visi izstrādes uzdevumi tiek reģistrēti JIRA un izstrādes komanda regulāri atjauno uzdevumu statusu.
4. Vienībtestu rakstīšana – izstrādātājiem papildus programmas kodam ir jāpapildina arī vienībtestu komplekss .
5. Integrācijas testu esamība – testētājiem ir jāizstrādā integrācijas testu komplekss.

3.2. Izvietošanas diagramma

Šajā apakšnodaļā ir detalizēti apskatīta nepārtrauktās integrācijas kanāla komponenšu izvietošanas diagramma. Diagramma redzama *3.1. attēlā.*



3.1. att. **Nepārtrauktās integrācijas kanāla komponenšu izvietošanas diagramma.**

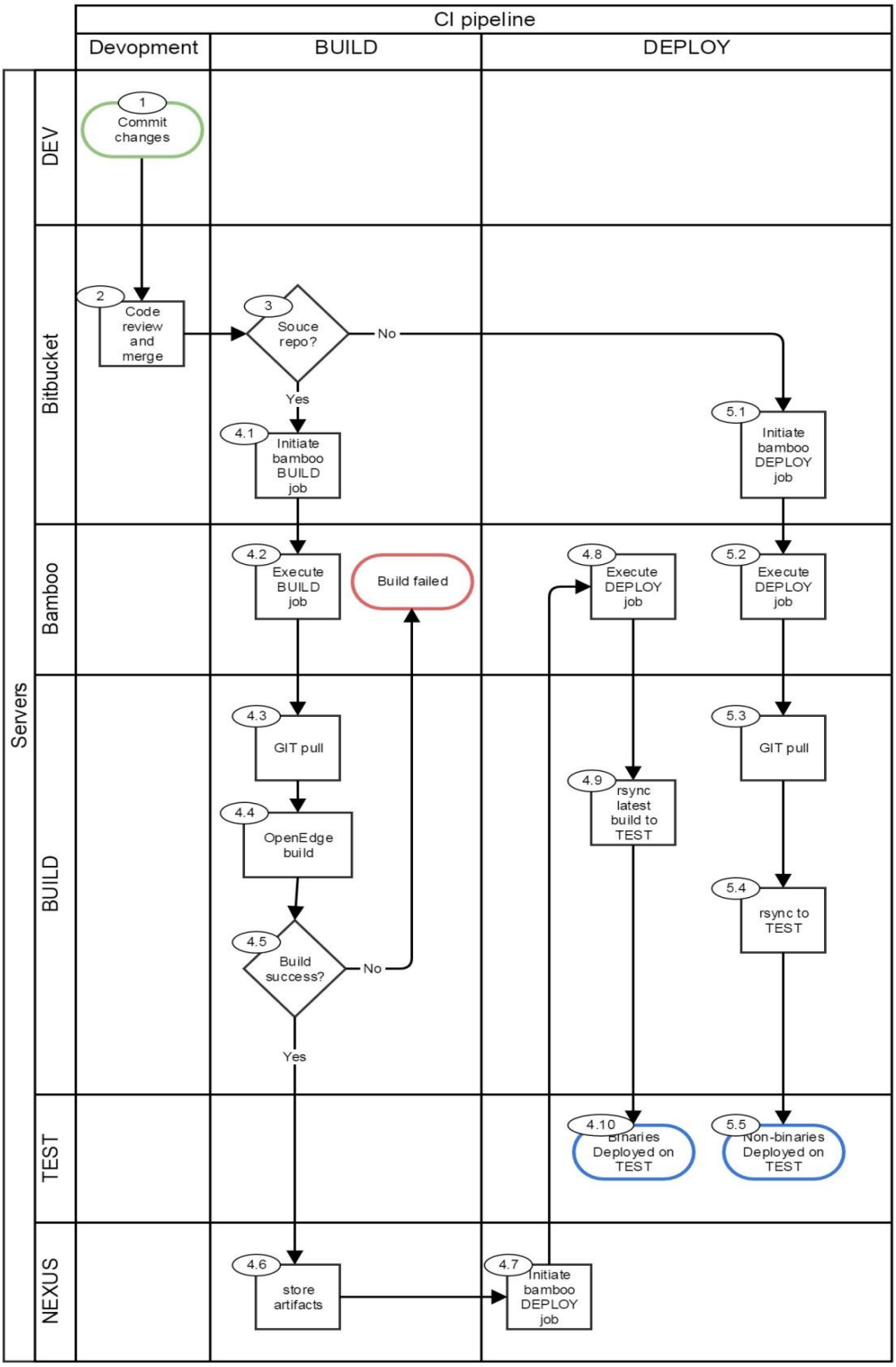
Šī diagramma detalizēti attēlo komponenšu novietojumu un kontekstu. Tālāk tiks apskatīta katra no komponentēm:

1. Izstrādes vide – izstrādātāju datori, kuros uzstādīts Open Edge developer studio un git klients, kā arī attiecīgās komponentes repozitorijs.
2. Bitbucket serveris – Platon sistēmai ir atvēlēti 3 git repozitoriji:
    - Sources repo – Open Edge avota koda repozitorijs
    - Backend repo – tīmekļa lietotnes back-end failu repozitorijs
    - Frontend repo – tīmekļa lietotnes front-end failu repozitorijs
3. Būves serveris – serveris uz kura tiek veikti automātiskie būvējumi, tā sastāvdaļas:
    - avota koda repozitorijs, jo tikai šis repozitorijs glabā kodu, kuram nepieciešama kompilācija
    - aktuālo datubāžu struktūras (tukšas datubāzes) – nepieciešamas kompilācijai
    - bināro failu glabātava – no šejienes binārie faili- būvējums tiek nodots glabāšanai Nexus artefaktu repozitorijā.

4. Datubāzes serveris, uz tā tiek uzstādīts aktuālais veiksmīgais būvējums. Binārie bibliotēkas faili.
5. Pacific aplication server – tīmekļa lietotnes back-end lietotņu serveris uz kura ir ievietoti gan back-end tīmekļa lietotnes faili gan Open Edge programmu biblioteka.
6. Tīmekļa lietotņu serveris – front-end serveris uz kura iz izvietoti tikai front-end tīmekļa lietotnes faili.
7. Nexus serveris – glabā būvējuma artifaktus
8. Bamboo serveris – glabā un izpilda būves un uzstādīšanas plānus

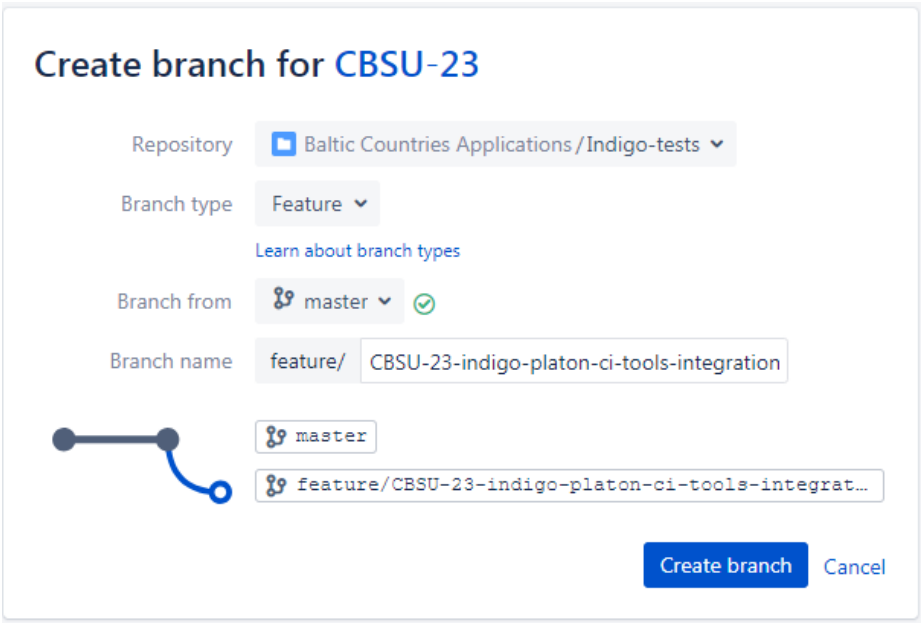
3.3. Procesu shēma un apraksts

Darba gaitā autors ir izveidojis un piedāvājis ieviest konkrētu darbību kopumu, secību, ko veiks nepārtrauktās integrācijas kanāls. Šajā apakšnodaļā tiks aprakstīta nepārtrauktās integrācijas procesu shēma, un aprakstīti tās soļi. Shēma redzama *3.2. att.*



3.2. att. **Nepārtrauktās integrācijas procesu shēma**

Sākot izstrādi tiek izveidots daba git repozitorija zars, šis zars var tikt izveidots tieši no JIRA lietotnes. Kā tas redzams *3.3. att.*



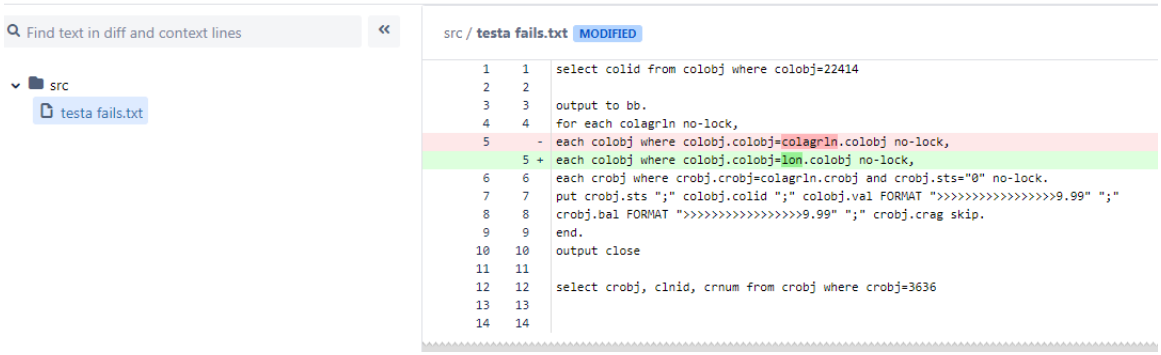
3.3. att. **Funkcionalitātes zara izveide.**

Tālāk detalizēti aprakstīti attēlā redzamie soļi:

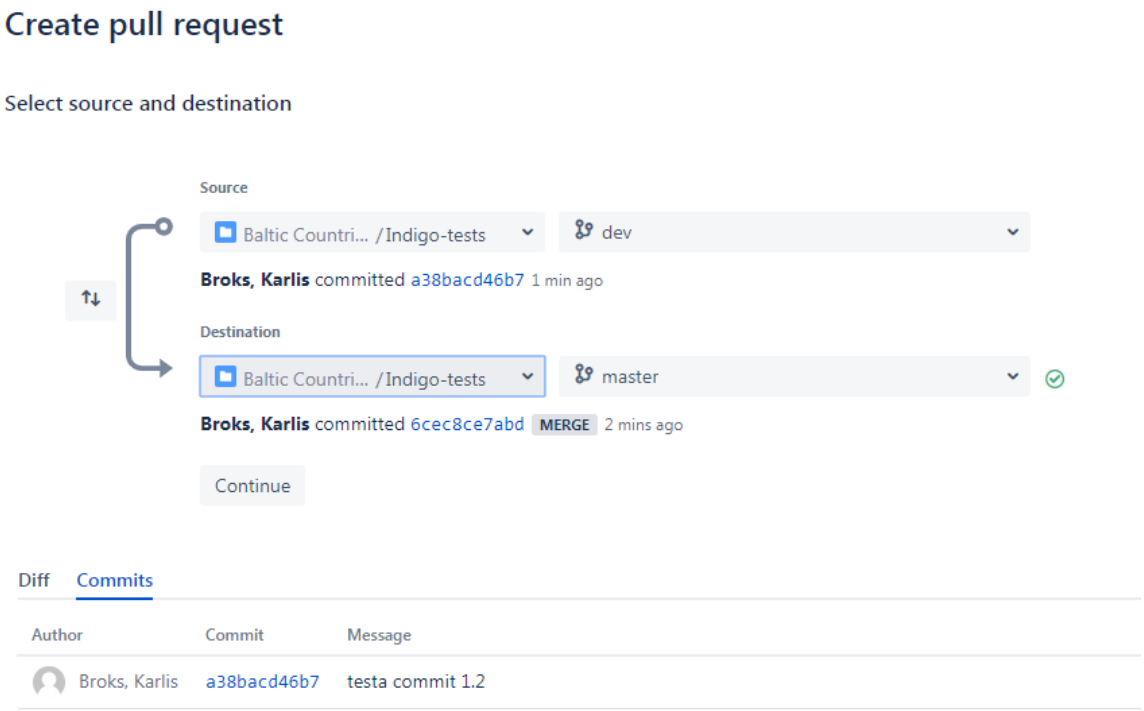
1. Izstrādātājs – git push – koda versijas nodošana uz bitbucket serveri(dev zars). (sk. *3.4.att.*)

3.4. att. **Koda versijas nodošana uz bitbucket izmantojot GIT terminālī**

2. Koda pārskatīšana un apvienošana – kad izstrādes vadītājs pārskata kodu, viņš pieņem lēmumu izstrādātāja kodu no dev zara apvienot ar master zaru, vai arī bugfx zaru apvienot ar master zaru. (sk. *3.5., 3.6. att.*)



3.5. att. **Koda pārskatīšana bitbucket**

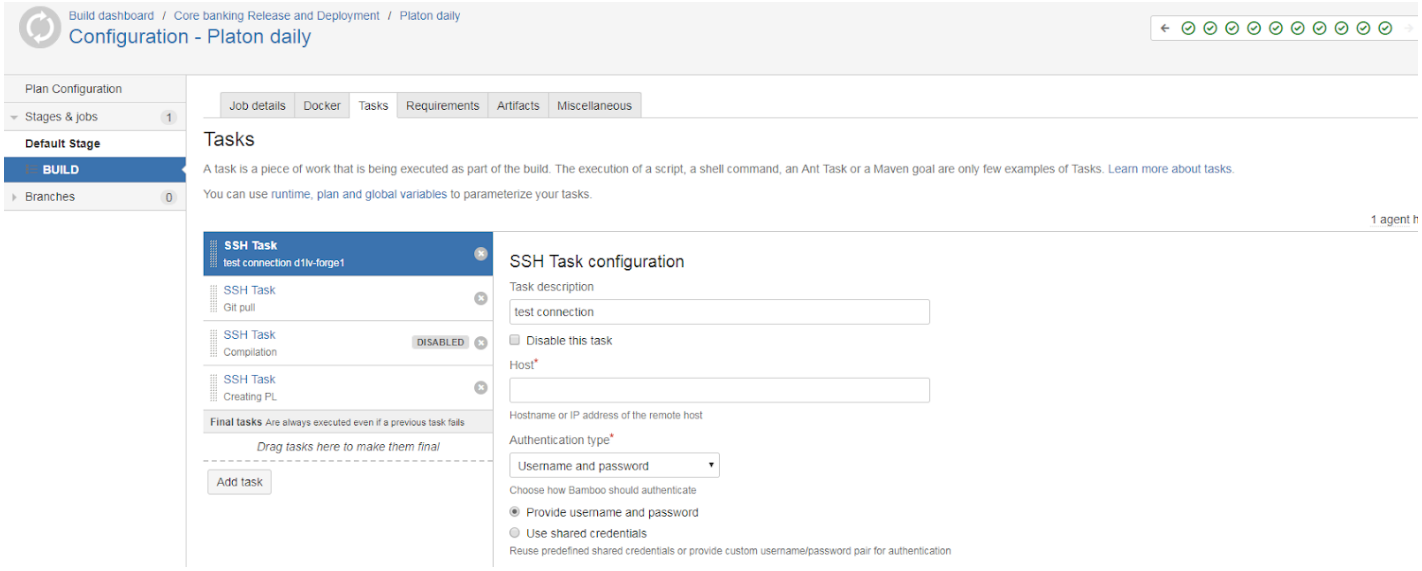


3.6. att. **Koda apvienošana (merge) bitbucket**

3. Vai izmaiņas ir veiktas programmas avota koda repozitorijā vai kādā citā?

4.1. Ja izmaiņas tiek veiktas avota koda repozitorijā, bitbucket iedarbija bamboo BUILD darbu.

4.2. Bamboo serveris izpilda BUILD darbu, izpilda skriptu uz būves servera. (sk. *3.7. att.*)

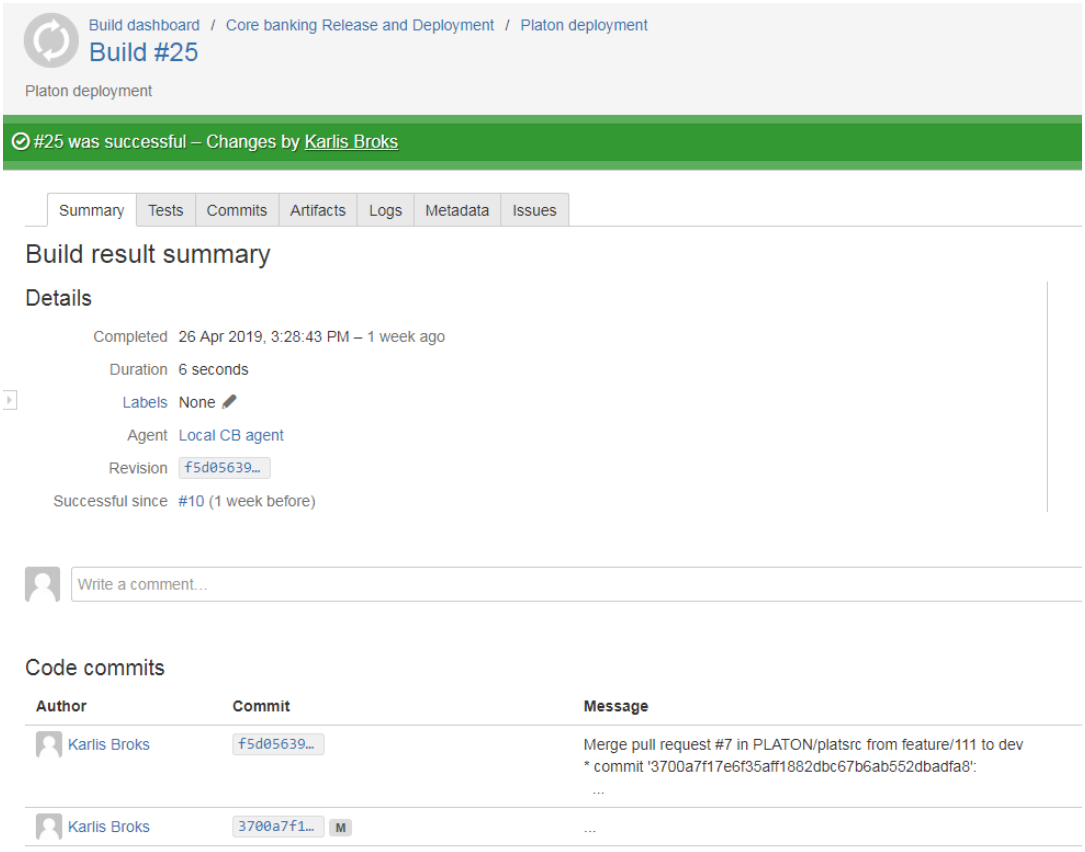


3.7. att. **Koda apvienošana (merge) bitbucket**

4.3. Skripts izpilda git pull komandu un uz būves servera tiek iegūta avota koda jaunā versija

4.4. Tiek izpildīta OpenEdge būves programma, tiek veikti automātiski vienībtesti, kompilēta bibliotēka, izmantojot nepieciešamās datubāzu struktūras.

4.5. Vai būvējums sekmīgs? Vai programmas veiksmīgi nokompilējušās? Vai testi veiksmīgi? (sk. *3.8. att.*)

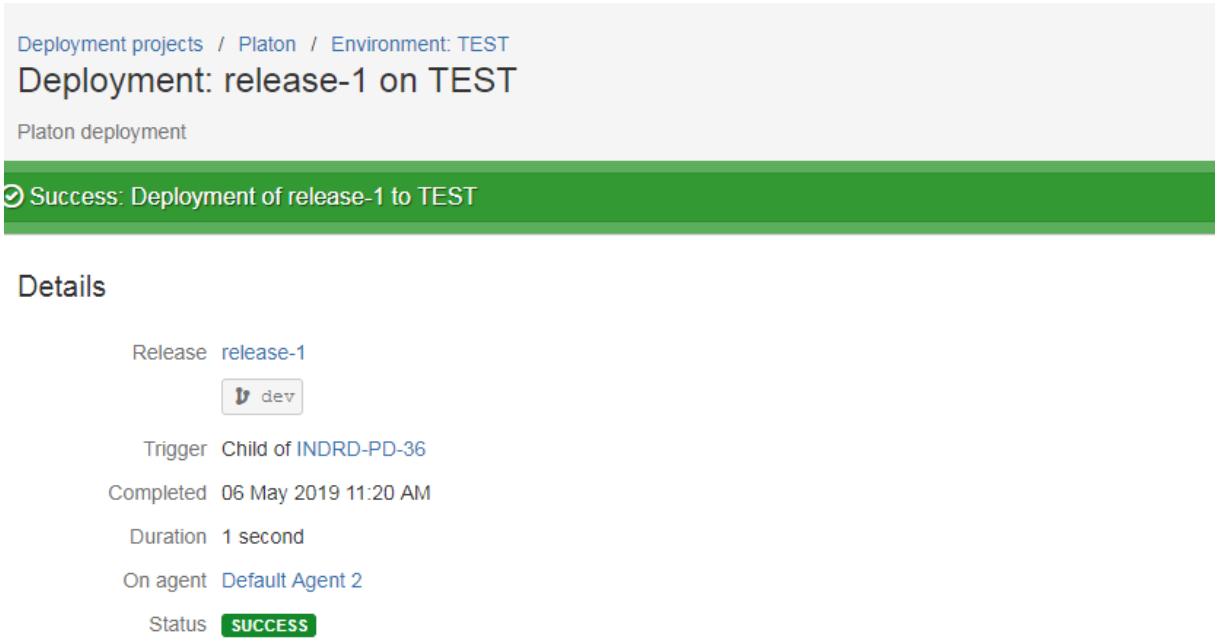


3.8. att. **Bamboo būvējuma kopsavilkums**

4.6. Būvējums tiek novietots NEXUS artefaktu glabātavā

4.7. NEXUS ierosina bamboo DEPLOY darbu

4.8. Bamboo izpilda DEPLOY darbu. (sk. *3.9. att.*)



3.9. att. **Bamboo DEPLOY darba kopsavilkums**

4.9. Būvējums tiek uzstādīts uz testa vides serveriem "rsync" komandu.

4.10. Binārās programmas uzstādītas testa vidē

5.1. Ja izmaiņas tiek veiktas citos repozitorijos (web faili, statiskie faili, konfigurāciju faili) repozitorijā, bitbucket iedarbija bamboo DEPLOY darbu.

5.2. Bamboo serveris izpilda DEPLOY darbu, izpilda programmu uz testa servera.

5.3. Skripts izpilda git pull komandu un uz būves servera tiek iegūta koda jaunā versija.

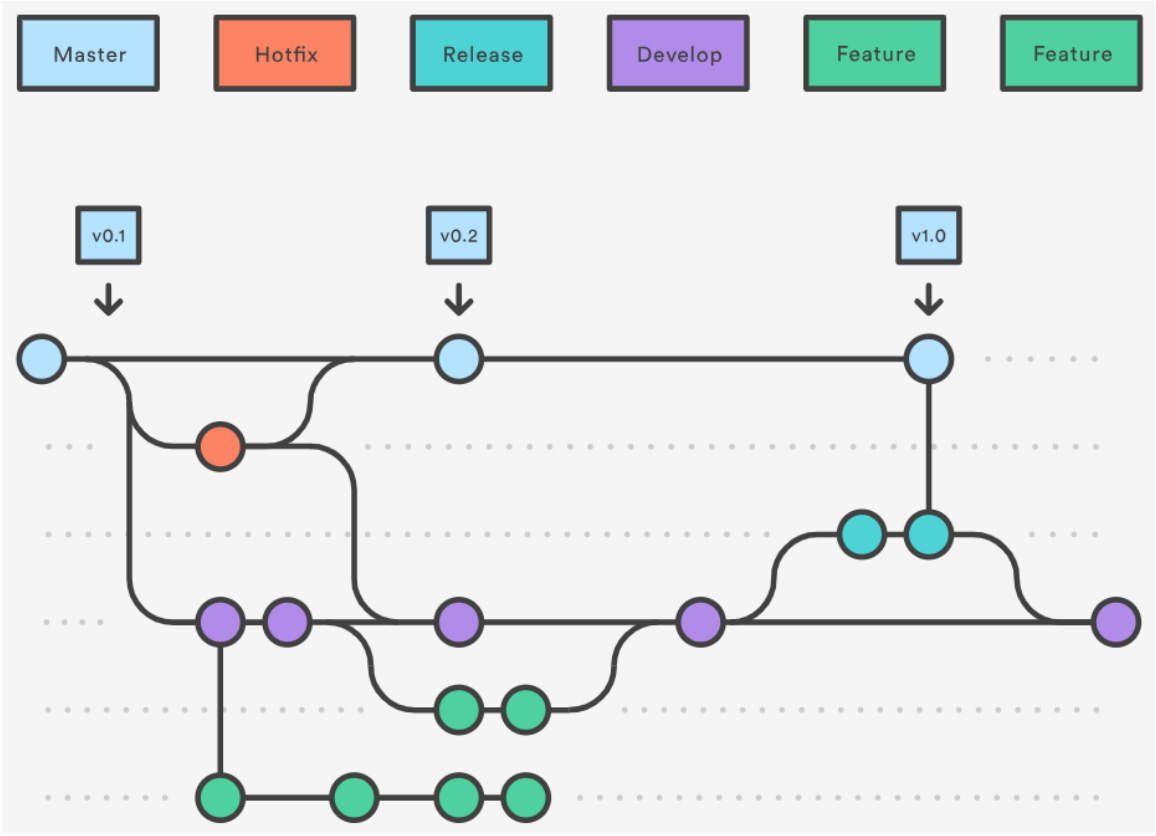
5.4. Nebinārās programmas komponentes tiek novietotas uz testa vides serveriem izmantojot "rsync" komandu.

5.5. Nebinārās programmas komponentes uzstādītas uz testa vides serveriem.

3.4. Ieviešanas soļi

3.4.1. Branching

Lai nodrošinātu efektīvu izstrādes procesu, ir nepieciešams definēt un izmantot attiecīgu versiju zarošanās metodi. Tādējādi tiek nodrošināta viegla versiju atsekošana, kā arī iespējots efektīvs izstrādes komandas paralēls darbs izstrādājot dažādas funkcionalitātes, kas avota koda līmenī iespējams var pārklāties. Bankas kodola sistēmas izstrādei tiks izmantots populāra zarošanās metode – "GitFlow" redzama *3.10. att.* Nākošajās apakšnodaļās ir detalizēti aprakstīti šis zarošanās metodes principi.[17]



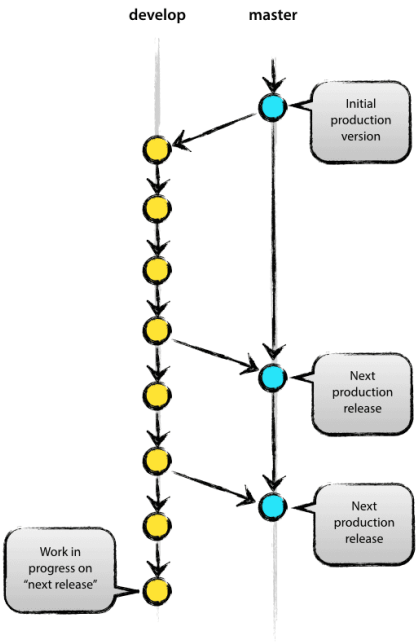
3.10. att. **GIT flow zarošanās metode**[17]

3.4.1.1. Galvenie zari

Centrālajā repozitorijā (origin) pastāvīgi tiek izmantoti 2 zari (sk. *3.11. att.*):

1. master
2. develop

"Master" zars vienmēr attēlo programmatūras produkcijas stāvokli, bet "develop" zars vienmēr attēlo pēdējās izstrādātās izmaiņas nākošajai relīzei. "develop" zars tiek saukts arī par "integrācijas zaru". Kad avota kods "develop" zarā sasniedz stabilu stāvokli un ir gatavs uzstādīšanai, tas tiek apvienots ar "master" zaru un tiek piešķirts relīzes numurs. Tādā veidā tiek izveidota produkcijas relīze. [17]



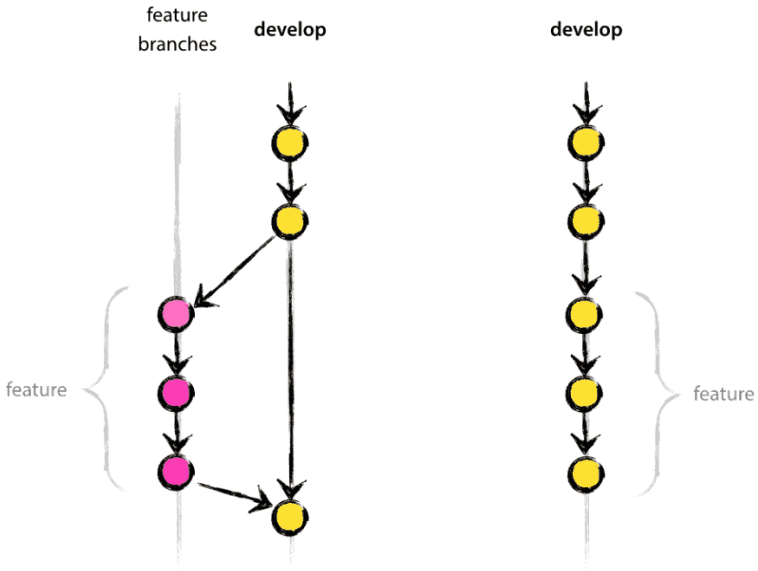
3.11. att. ***Master un develop zari***

3.4.1.2. Atbalsta zari

Izstrādes procesā var pastāvēt 3 veidu atbalsta zari:

1. feature
2. release
3. hotfix

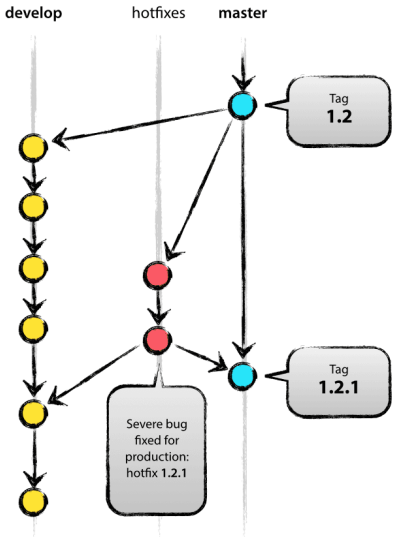
"Feature" zars atzarojas no "develop" zara un tam jābūt apvienotam ar "develop" zaru (sk. *3.12. att.*). Šī veida atbasta zars parasti tiek izmantots lai izstādātu jaunas funkcionlaitātes, nākotnes relīzēm. Šo relīžu uzstādīšanas laiks var arī nebūt noteikts. "feature" zars pastāv tikai līdz brīdim, kad funkcionalitāte tiek piegādāta uz "develop" zara. Parasti "feature" zars pastāv tikai izstrādātāju lokālajos repozitorijos, bet ne uz GIT servera (origin). [17]



3.12. att. ***Feature zars***

"Release" atzarojās no "master" un tiek apvienots ar "develop" un "master" zariem. Tas atbalsta sagatavošanos jauna produkta relīzei. Šī zara izmantošana palīdz veikt nelielus kļūdu labojumus un sagatavot metadatus relīzes izlaišanai.To visu darot "release" zarā "develop" zars tiek atbrīvots lai saņemtu nākošās relžies funkcionalitāti.

"Hotfix" atzarojās no "master" un tiek apvienots ar "develop" un "master" zariem. (sk. 3.*13.att.*) "hotfix" zari ir līdzīgi "release" zariem, bet tie tiek izmantoti, lai sagatavotu neplānotu produkcijas relīzi. Tie tiek izveidoti, lai labotu kritiskas kļūdas (bug) produkcijā. Izmantojot "hotfix" zaru, darbs no citu izstrādātāju puses "develop" zarā var turpināties. [17]



3.13. att. **Hotfix zars**

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

3.4.2. Būvēšanas servera izveide

Sekmīgai būves procesa izveidei tika pieņemts lēmums izveidot atsevišķu būves serveri. Tika izveidots virtuālas būves serveris ar Red Hat Enterprise Linux operātājsistēmu. Servera parametri redzami *3.1. tabulā.*

3.1. tabula

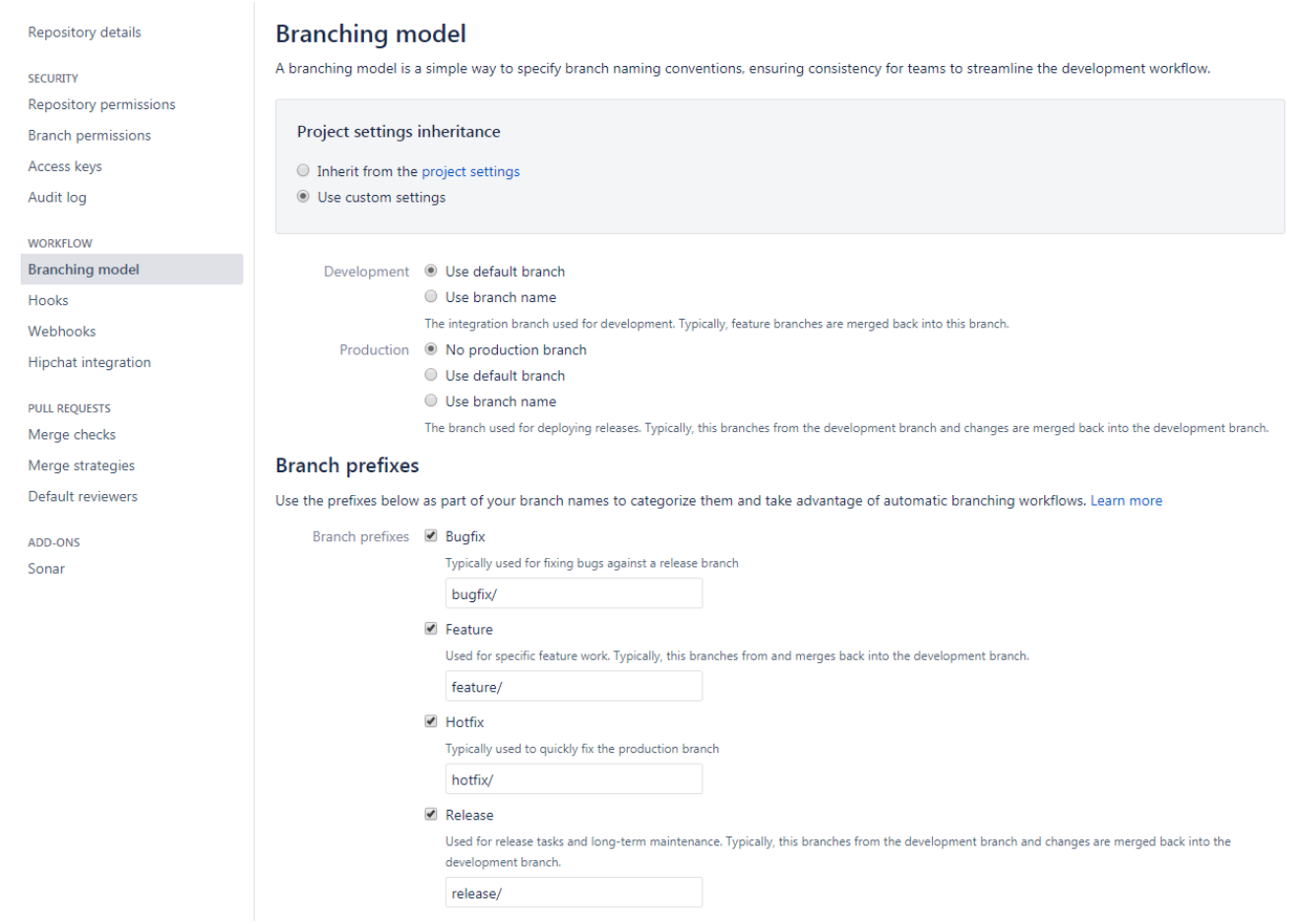
Būves servera parametri

Operātājsistēma	Red Hat Enterprise Linux operātājsistēmu
CPU	Intel Xeon E5-2699 v3 @ 2.30GHz
RAM	32GB
HDD	200GB

Tā kā šis serveris atrodas bankas iekšējā tīklā un tam nav un nebūs piekļuve ārējam tīklam, radās nepieciešamība pēc ērta pakotņu uzstādīšanas mehānisma. Tādēļ uz NEXUS repozitoriju menedžera tika nokonfigurēts yum proxy repozitorijs. Ar tā palīdzību ir iespējams uzstādīt pakotnes, kas atrodas RHEL oficiālajā repozitorijā. Tas pavēra iespēju uzstādīt GIT, OpenEdge un citas lietotnes un to atkarības. Tā kā no tīkla infrastruktūras viedokļa gan bitbucket, gan bamboo, gan nexus atrodās ārpus bankas tīkla – ārējā pakalojumu sniedzēja mākonī, bija nepieciešams izveidot VPN ar bankas iekšējo tīklu. Tālāk bija nepieciešams konfigurēt banakas ugunsmūri, lai pastāvētu savienojums starp būves serveri un mākonī izvietotajiem serveriem.

3.4.3. Bitbucket konfigurācija

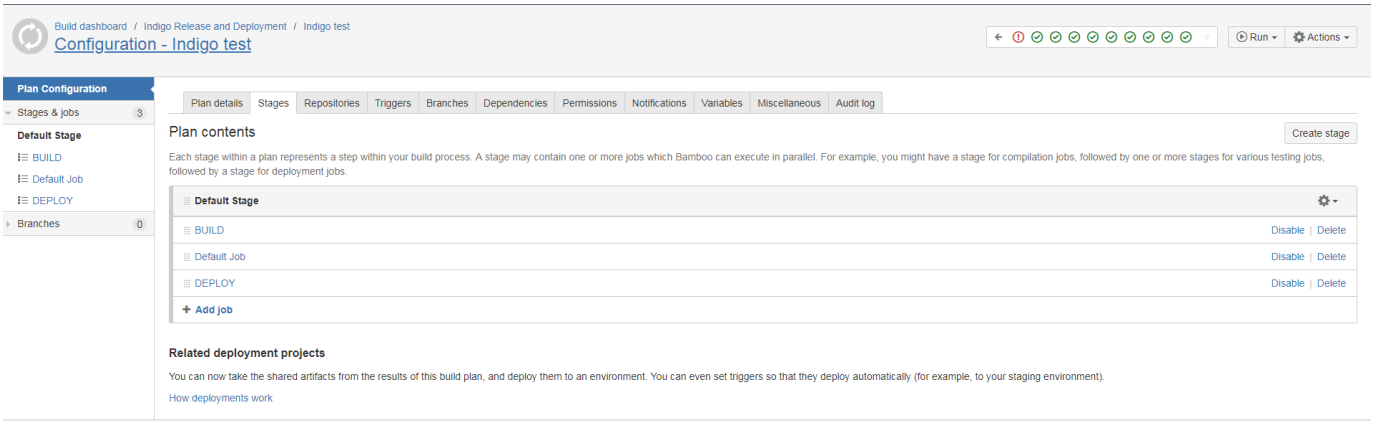
Tiek izveidots bitbucket projekts, pie kura piekļuves tiesības ir tikai sistēmas izstrādātājiem. Piekļuves tiesības tiek pārvaldītas izmantojot Azure AD. Zem attiecīgā projekta tiek izveidoti nepieciešamie repozitoriji. Tiek konfigurēti zarošanās uzstādījumi (sk. *3.14.att.*), izvēlēts attiecīgais zarošanās koks un citi uzstādījumi. Tiek izveidoti arī piekļuves marķieri (access tokens), lai varētu izstrādātāji varētu ērti piekļūt repozitorijiem no savu darbstaciju termināliem.



3.14. att. **Bitbucket konfigurācija**

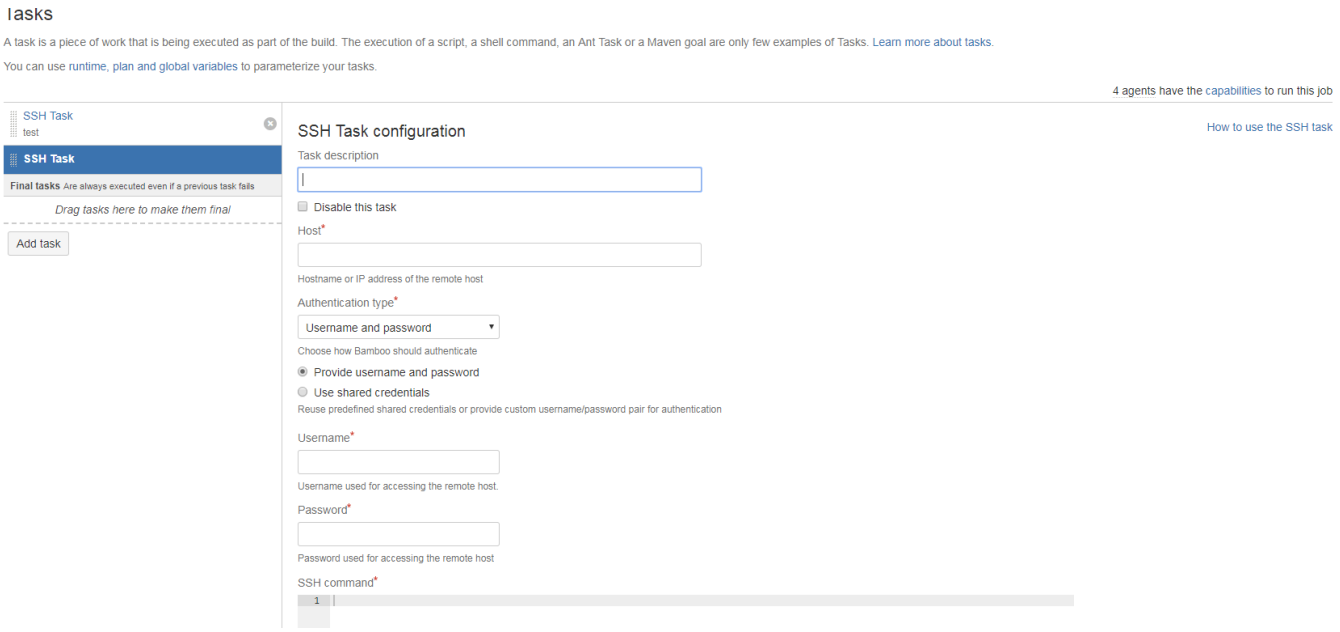
3.4.4. Bamboo konfigurācija

Bamboo tiek izveidots projekts, kas attiecās uz konkrēto bankas sistēmu. Tālāk tiek izveidots attiecīgais plāns kurā tiek izveidoti attiecīgie darbi (jobs)(sk. *3.15.att.*)



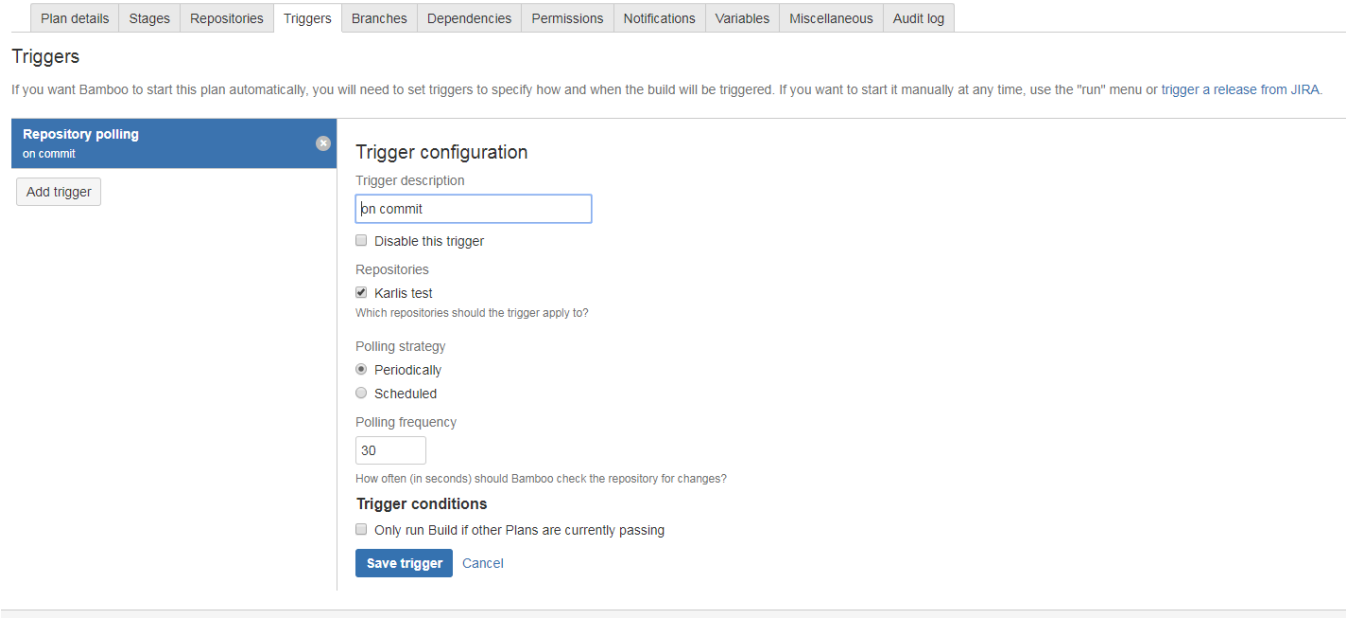
3.15.att. **Bamboo plāna izveide**

Kā sīkākā vienība, darbam tiek izveidoti uzdevumi un BUILD darbam tiek izveidots SSH uzdevums, kas pieslēdzās būves serverim un uz tā izpilda noteiktus skriptus kā redzams *3.16.att.*



3.16.att. **Bamboo darbu izveide**

Tāpat ir svarīgi uzstādīt izpildes nosacījumu (trigger), lai veicot *commit* bitbucket, tiktu izpildīts bamboo darbs.(sk. *3.17.att.*)



3.17.att. **Bamboo izpildes nosacījumu izveide**

3.4.5. Būvēšanas skripti

Pēc plāniem nākotnē būvēšanai ir paredzēts izmantot maven vai ant skriptus, bet pašlaik tiek izmantoti UNIX skripti, kas darbina Open Edge programmas. Tika izveidota speciāla Open Edge programma, kas izmantojot dažādas datubāžu struktūras kombinācijas kompilē kodu. Bibliotēkas būvēšanas process sastāv no 3 daļām:

1. Vides mainīgo definīcija
2. Programma, kas inicializē vides mainīgos un izsauc Open Edge procedure editor
3. Kompilācijas programma, kas kompilē visus Open Edge avota koda failus attiecīgajā direktorijā
4. Kompilācijas log fails – no šī faila var izgūt kompilācijas procesa detaļas, katras programmas kompilācijas statusu, kā arī redzēt kopējo kompilācijas rezultātu.(sk. *3.18.att.*)

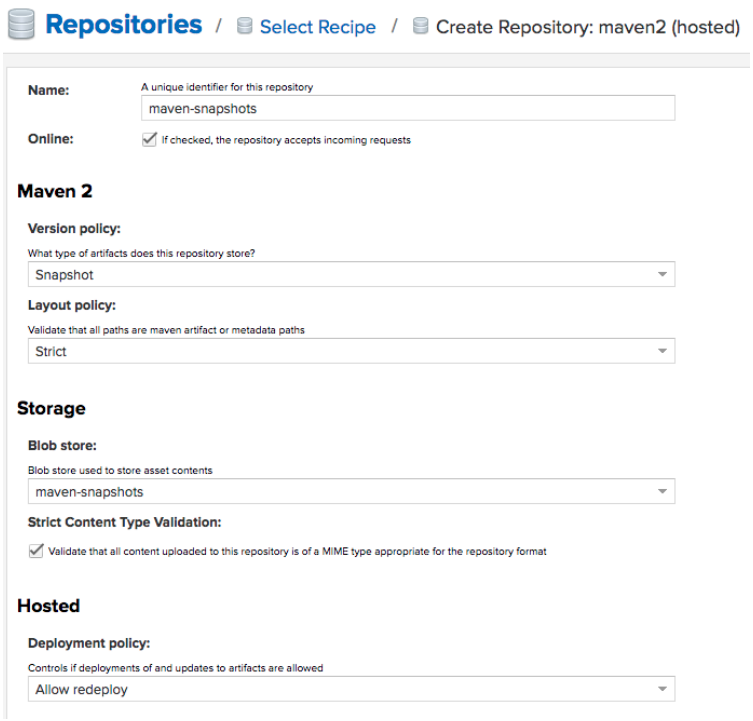
```
[18/04/2019 09:36:09.495+03:00] removing stgetdeal_old2.r AFTER ZPL7
****************************************
[18/04/2019 09:36:09.798+03:00] compiling stgetdeal_old2.p RESULT: SUCCESS IN ZP
L8
****************************************
TOTAL: 4395 FILES
COMPILED: 4395 FILES
FAILED: 0 FILES
```

3.18.att. **Kompilācijas log faila kopsavilkums**

3.4.6. Nexus konfigurācija

Nexus pusē tiek izveidoti maven repozitoriji momentuzņēmumiem (snapshots) un relīzēm. (sk. *3.19.att.*)

Konfigurācijas tiek veiktas arī no klienta puses, lai repozitorijs izmantotu Nexus.



3.19.att. **Nexus momentuzņēmuma repozitorija izveidošana**

3.5. Rezultāti

Šī darba laikā tika īstenota tikai daļa no plānotā nepārtrauktās integrācijas kanāla izveides. Tika veiksmīgi izveidots kompilēšanas un būvēšanas mehānisms kā arī uzstādīšanas mehānisms dažādās vidēs. Izmantojot nepārtrauktās integrācijas kanālu, programmatūras būvējuma sagatavošana un uzstādīšana testa vidē aizņem ilgākais 8 minūtes.

4. REZULTĀTU ANALĪZE

Šajā nodaļā tiks apskatīts kā nepatrauktās integrācijas koncepta izmantošana ietekmē izstrādes un operacionālo resursu izmantošanu, kā arī kā tas uzlabo programmatūras kvalitāti un piegādes ātrumu. Tiks apskatīta pagājušo gadu problēmu pieteikumu statistika kā arī iepriekšējā prakse un prognozēts, kā mainīsies raksturlielumi, kad tiks izmantots nepārtrauktās integrācijas koncepts.

4.1. Piegādes ilgums

Šajā apakšnodaļā tiks salīdzināti piegādes ilgumi izmantojot līdzšinējo programmatūras piegādes mehānismu un izmantojot nepārtrauktās integrācijas kanālu. Tiks apskatīts piegādes laiks uz produkcijas vidi, neņemot vērā lietotāju paņemšanas testu veikšanu, jo šis lielums variē atkarībā no veiktajām izmaiņām/labojumiem.

Piegādes ilgums izmantojot manuālo procesu norādīts ņemot vērā nevis tikai laiku, kas aizņem attiecīgās darbības veikšanai, bet aptuveno darbības izpildes laiku pēc novērotā ikdienas gaitā.

4.1.1. Atsevišķas programmas pievienošana

Tabulā 4.1. attēlots piegādes ilgums atsevišķas programmas pievienošanai būvējumam.

4.1. tabula

Atsevišķas programmas pievienošanai būvējumam

	Manuālais process		**Nepārtrauktā integrācija**	
	Veic	Ilgums	Veic	Ilgums
Kompilēšana un uzstādīšana uz testa vides	manuāli izstrādēs vadītājs	15 min	automātiski	2 min
Kompilēšana produkcijas pakotnei	manuāli izstrādes vadītājs	15 min	automātiski, iniciē izstrādes vadītājs	1 min
Uzstādīšana uz staging vides	manuāli administratori	30 min	automātiski	1 min
Smoke tests	manuāli administratori	30 min	automātiski	5 min
Uzstādīšana uz produkcijas vides	manuāli administratori	30 min	automātiski, iniciē administratori	1 min
Kopā:		2h		10 min

Ņemot vērā novēroto, jaunas programmas versijas uzstādīšana produkcijā aizņem aptuveni 2h izmantojot esošo piegādes mehānismu, šis ir izmaiņas, ko var uzlikt nepārtraucot sistēmas darbību.

Turpretī izmantojot nepārtrauktās integrācijas mehānismu, to ir iespējams izdarīt 12 reizes ātrāk – 10 minūšu laikā turklāt ar minimālu cilvēkresursu piesaisti.

4.1.2. Pilna būvējuma uzstādīšana

Tabulā 4.2. attēlots piegādes ilgums pilnas bibliotēkas būvējuma izveidei un uzstādīšanai produkcijā.

4.2. tabula

Pilna būvējuma uzstādīšana

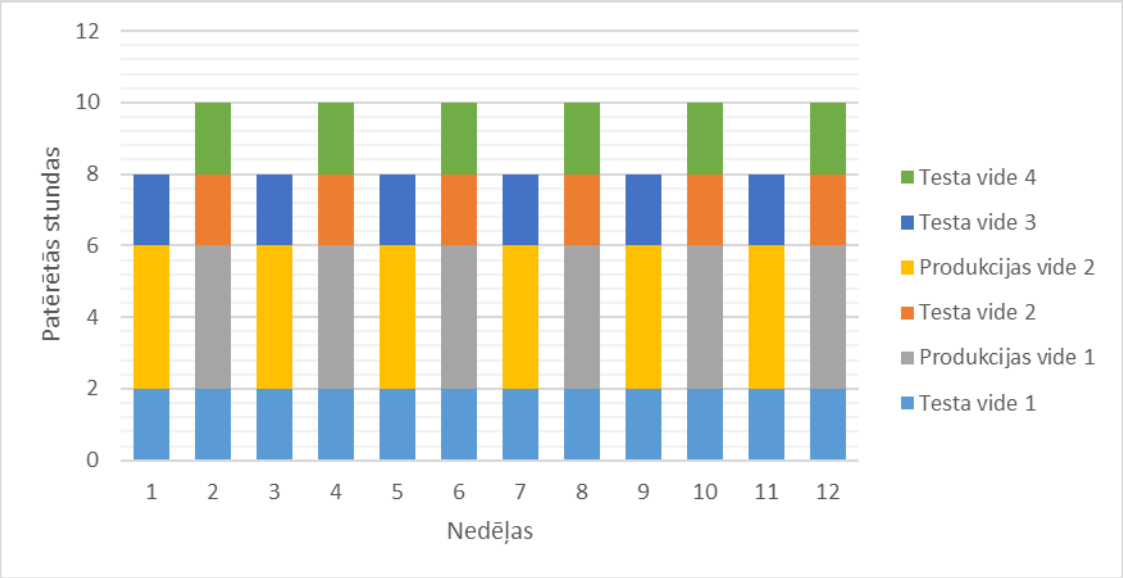
	Manuālais process		**Nepārtrauktā integrācija**	
	Veic	Ilgums	Veic	Ilgums
Kompilēšana un uzstādīšana uz testa vides	manuāli izstrādēs vadītājs	2h	automātiski	12 min
Kompilēšana produkcijas pakotnei	manuāli izstrādes vadītājs	2h	automātiski, iniciē izstrādes vadītājs	12 min
Uzstādīšana uz staging vides	manuāli administratori	30 min	automātiski	1 min
Smoke tests	manuāli administratori	30 min	automātiski	4 min
Uzstādīšana uz produkcijas vides	manuāli administratori	1h	automātiski, iniciē administratori	1 min
Kopā:		6h		30 min

Kā var redzēt no novērotajiem datiem arī veicot pilna būvējuma uzstādīšanu laika ekonomija ir aptuveni 12 reizes. Turklāt ilgāko laiku aizņem programmas kompilēšana. Šo laiku vēl varētu samazināt, ja iespējotu vairāku procesoru izmantošanu kompilēšanas procesā.

Diemžēl ņemot lielo resursu noslogojumu un faktu, ka pakotnes sagatavošanu veic tikai 2 cilvēki, piegādes laiki var izaugt līdz pat 2 dienām. Turpretī izmantojot nepārtrauktās integrācijas kanālu resursu izmantojums ir vien pāris minūtes, kas nepieciešamas lai inicializētu attiecīgo Bamboo darbu.

4.2. Resursu izmantojums

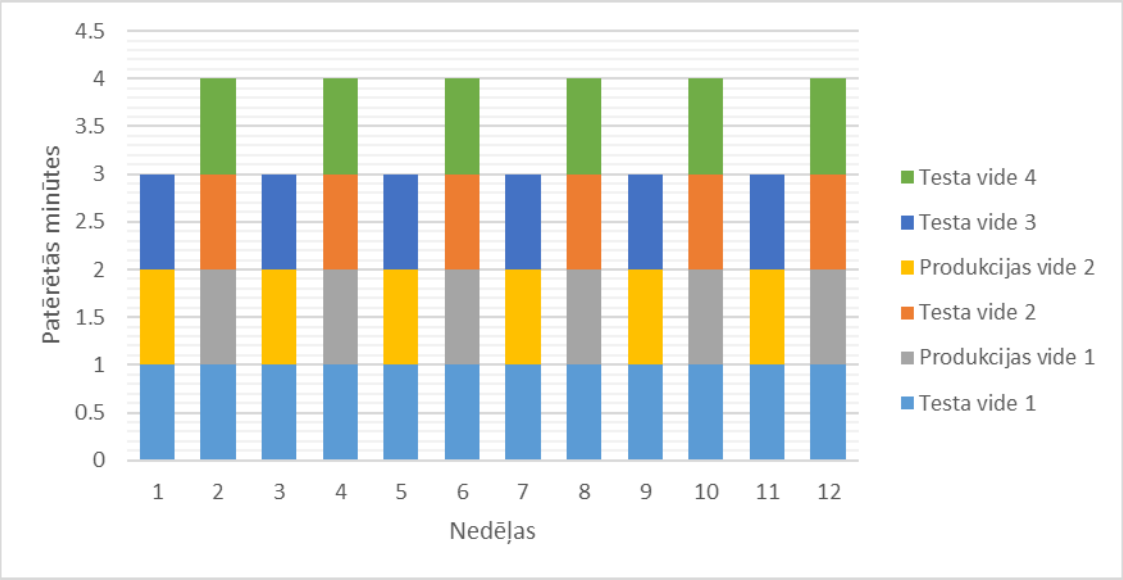
Ņemot vērā relīžu grafiku tika izstrādāts aptuvenais piegāžu grafiks 3 mēnešu griezumā. Izstrāde tiek veikta dažādos paralēlos projektos, tādēļ ir attēlotas 4 testa vides un 2 produkcijas vides. Produkcijas vides ir atšķirīgas, jo šobrīd šī bankas kodolas sitēma tiek izmantota divās valstīs. Testa vidē 1 piegādes tika veiktas reizi nedēļā, jo šajā vidē notiek aktīva izstrāde. Ņemot vērā iepriekšējā apakšnodaļā apskatīto aptuvenais piegādes ilgums uz testa vidi ir 2 stundas un produkcijas piegāde aizņem vēl 4h izmantojot manuālo piegāžu mehānismu. Patērēto cilvēkresursu izmantojumu var redzēt *attēlā 4.1.*



4.1.att. **Patērēto cilvēkstundu lietojums 3 mēnešu griezumā, izmantojot manuālo piegādes mehānismu.**

Kopumā 3 mēnešu griezumā uz šim aktivitātēm ir iztērētas aptuveni 108 stundas. No kurām aptuveni 67% jeb ~72h tika izmantotas izstrādes komandas resursi (izstrādes vadītājs), bet 33% jeb ~36h operacionālie resursi (aplikāciju administratori).

Kā redzams *attēlā 4.2*. Izmantojot nepatrauktās integrācijas kanālu, resursu izmantojums ir līdz 5 minūtēm nedēļā, jo no administratoru, izstrādes vadītāja puses ir nepieciešams tikai apstiprināt, inicializēt automātisko procesu izpildīšanu. Līdz ar to resursu izmantojumu var uzskatīt par neesošu.



4.2.att. **Patērēto cilvēkstundu lietojums 3 mēnešu griezumā, izmantojot nepārtrauktās integrācijas konceptu.**

4.3. Būvējuma kvalitāte

Pēc pieredzes vismaz 20% manuālo būvējumu ir bijuši nederīgi, vai nu tādēļ, ka nav iekļauta kāda aktuālā programma, vai kompilēšanai izmantota nepareizā datubāzes struktūra. Šādi gadījumi nav retums pārsvarā cilvēka faktora, kļūdas dēļ. Tiek zaudēts laiks atkārtoti būvējot bibliotēku.

Nepārtrauktās integrācijas kanāla izbūve nodrošina pienācīgu avota koda versiju kontroli, līdz ar to nodrošinot, ka lietotnes programmas tiks kompilētas, būvētas izmantojot korekto koda versiju no korektā versiju kontroles sistēmas zara. Kļūdas ir viegli atrodamas un iepriekšējās versijas viegli atgriežamas.

Būvējums netiek uzstādīts, ja kāda no komponentēm netiek izpildīta (piemēram vienībtesti). Būvējums tiek veidots pamatojoties uz aktuālajām datubāžu shēmām, kas arī tiek versiju kontrolētas.

SECINĀJUMI

Maģistra darba izvirzītais mērķis ir sasniegts. Darba izstrādes gaitā bankas kodola sistēmās jau daļēji ir ieviests nepārtrauktās izstrādes koncepts un izstrādātas vadlīnijas šī koncepta pilnīgais ieviešanai.

Šī darba gaitā ir izpētīta literatūra par nepārtrauktās integrācijas ieviešanas pieejām, veiksmīgām praksēm. Ir īsumā apskatīti nepārtrauktās integrācijas rīki, kā arī tās pamatprincipi. Detalizēti ir apskatīti esošie programmatūras izstrādes, piegādes procesi un pamatojoties uz specifisko tehnoloģijas pielietojumu kombināciju izvēlēts atbilstošais risinājums izstrādes procesa problēmām izmantojot nepārtrauktās integrācijas rīkus.

Šajā darbā apskatītie rīki nav universāla kombinācija nepārtrauktās integrācijas ieviešanai jebkurā projektā, bet var kalpot par iestrādnēm kādam, kurš vēlas izmantot Atlassian rīku komplektu darbojoties ar nestandarta programmēšanas valodas lietotnēm (it īpaši Open Edge).

Svarīgi saprast, ka nepārtrauktā integrācija ir ne tikai rīku komplekss, bet arī darba stils. Veicot šo darbu ir secināts, ka nepietiek tikai ar rīku ieviešanu, bet arī jāmaina izstrādes komandas, kā arī operacionālās komandas darba pieeja. Tādēļ autors ir sācis darbu pie apmācību kompleksa, kas palīdzētu šīm komandām pielāgot savu darbu nepārtrauktās integrācijas principiem, kā arī apgūtu minētos rīkus.

Veicot rezultātu analīzi autors secināja, ka konkrētajā situācijā izmantojot nepārtrauktās integrācijas kanālu sistēmas piegāžu uzstādīšanas laiku var samazināt līdz pat 12 reizēm. Procesu automatizēšana noved arī pie ievērojama resursu izmantojuma samazināšanas, būtībā likvidējot manuālu darbību nepieciešamību veidojot būvējumu un to uzstādot kādā no vidēm. Tiek nodrošināta arī pienācīga koda versiju atsekošana. Rezultātā piegādes ir ātrākas, lētākas un drošākas.

LITERATŪRAS AVOTI

1. Wikipedia OpenEdge ABL apraksts, 29 Aug 2018. Pieejams: https://en.wikipedia.org/wiki/OpenEdge_Advanced_Business_Language
2. Sistēmu raksturlielumu salīdzinājums, 2018. Pieejams: https://db-engines.com/en/system/OpenEdge%3BPostgreSQL
3. PAS apraksts, 2015. Pieejams: https://documentation.progress.com/output/ua/OpenEdge_latest/pasoe-intro/what-is-pacific-application-server-for-openedge-3f.html
4. Pebble apraksts, 2013. Pieejams: https://github.com/PebbleTemplates/pebble
5. Node.js apraksts, 20 Dec 2018. Pieejams: https://en.wikipedia.org/wiki/Node.js
6. Progress ABLUnit apraksts, 2017. Pieejams: https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/pdsoe/overview-of-ablunit-testing-framework.html
7. Attlassian Jira apraksts, 13 Maijs 2019. Pieejams: https://www.atlassian.com/software/jira
8. Jira Xray apraksts, 13 Maijs 2019. Pieejams: https://marketplace.atlassian.com/apps/1211769/xray-test-management-for-jira?hosting=cloud&tab=overview)
9. Amazon nepārtrauktās integrācijas apraksts, 13. Maijs 2019. Pieejams: https://aws.amazon.com/devops/continuous-integration/
10. Atlassian nepārtrauktās integrācijas, piegādes un uzstādīšanas apraksts, 13. Maijs 2019. Pieejams: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment
11. GIT apraksts, 10 Jan 2019. Pieejams: https://en.wikipedia.org/wiki/Git
12. Bitbucket apraksts, 19 Jan 2019. Pieejams: https://en.wikipedia.org/wiki/Bitbucket
13. Bamboo CI server apraksts, 20 Jūl 2017. Pieejams: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

14. Nexus apraksts, 13 Maijs 2019. Pieejams: https://blog.sonatype.com/2010/04/why-nexus-for-the-non-programmer/
15. G. Jezuns “Pastāvīgā integrācija Java programmatūras izstrādē”, bakalaura darbs, Latvijas Universitāte, 2009.
16. E. Diebelis “Programmatūras paštestēšana”, promocijas darbs datorzinātnes doktora grāda iegūšanai, Latvijas Universitāte, 2012.
17. Gitflow zarošanās metodes apraksts, 13 Maijs 2019. Pieejams: https://datasift.github.io/gitflow/IntroducingGitFlow.html
18. G. Kim, P. Debois, J. (Writer on information technology) Willis, J. Humble, un J. Allspaw, The DevOps handbook : how to create world-class agility, reliability, and security in technology organizations. Portland: IT Revolution Press, 2016.
19. R. Sviklis “DevOps ieviešana IT uzņēmumā”, maģistra darbs, Latvijas Universitāte, 2018.
20. Pressman, Roger S. Software engineering : a practitioner’s approach / Roger S. Pressman. — 7th ed. New York: McGraw-Hill Companies, Inc, 2010.

DOKUMENTĀRĀ LAPA

Maģistra darbs "Nepārtrauktās integrācijas koncepts bankas kodola sistēmās" izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 15.05.2019

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____

(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____

(Vadītāja paraksts un datums)

Darbs iesniegts **maģistratūras sekretariātā** _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: _____

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)