RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts
Lietišķo datorzinātņu katedra

Kārlis Rozenbergs

Docker tehnoloģijas analīze un realizācija programmatūras izstrādes procesa automatizācijai

Bakalaura darbs

Darba vadītājs
Docents, Dr.Sc.Ing.
Artūrs Bartusevičs

Rīga, 2017

Anotācija

Bakalaura darba tēma ir “Docker tehnoloģijas analīze un realizācija programmatūras izstrādes procesa automatizācijai”.

Darba analītiskajā daļā ir apskatītas virtuālo mašīnu tehnoloģijas un to iedalījums tipos – uz hipervizoriem bāzēta un uz konteineriem bāzēta virtualizācija. Ir veikts abu tipu salīdzinājums, novērtējot ieguvumus un trūkumus.

Darbā ir veikta sīkāka Docker tehnoloģijas izpēte. Ir apskatīta tās izveides vēsture un attīstība, kā arī uzbūves arhitektūra. Ir izpētīti ar Docker konteineriem saistītie drošības riski, kā arī analizēts veiktspējas pētījums, kurā tos salīdzina ar uz hipevizoriem balstītu virtualizāciju.

Darba praktiskajā daļā ir veikta Docker virtualizācijas prototipa izstrāde. Ir apskatīti programmatūras testēšanas vides izveides, testa būvējumu uzstādīšanas, testa vides atjaunošanas procesi, kā arī veikta šo procesu automatizācija. Ir aprakstīti izstrādātā prototipa ieguvumi, rezultāti un tālākie attīstības virzieni.

Darba rezultātā ir sasniegti visi izvirzītie mērķi un dots reāls risinājuma piemērs programmatūras ražošanas procesa izpildes ātrumu uzlabošanai.

Bakalaura darbā ir 51 lappuse, 14 attēli, 7 tabulas, 60 nosaukumu informācijas avoti.

Abstract

The title of the Bachelor Thesis is “Analysis and Implementation of Docker Technology for Automation of Software Development Process”.

Virtualization technology and types of virtual machines – hypervisor and conteiner based - are reviewed in the analytical part of the Bachelor Thesis. Both types are compared in context of performance and feature availability.

In Thesis deeper exploration in Docker technology has been made. History, development and architecture of Docker is described. Then security risks and performance analysis of Docker conteiners is reviewed.

In practical part of the Bachelor Thesis Docker virtualization prototype is created. Processes of creating test environment, deployment of test builds, restoring of test environment are described. Implementation of automatization is made for those processes. Finally pros and cons of prototype are reviewed and perspectives of future are given.

All goals has been completed for Thesis and real life solution to improve performance of software development process is given.

The Bachelor Thesis consists of 51 pages, 14 illustrations, 7 tables and 60 sources of Information.

Saturs

Terminu un saīsinājumu vārdnīca 7
Ievads 8
1. Virtualizācija 10
1.1. Hipervizori 10
1.1.1. WMware Workstation 12
1.1.2. Oracle VM VirtualBox 12
1.1.3. Microsoft Hyper-V 13
1.2. Konteineri 13
1.2.1. Chroot Jails 14
1.2.2. Namespaces 15
1.2.3. Cgroups 15
1.2.4. Salīdzinājums ar VM 16
2. DOCKER 18
2.1. Vēsture 18
2.2. Atbalsta tehnoloģijas 20
2.3. Arhitektūra 21
2.3.1. Klienta-servera modelis 21
2.3.2. Tīkla porti 22
2.3.3. Komandrindu rīks 23
2.3.4. API 23
2.3.5. Konteineru tīklošana 23
2.4. Veiktspēja 24
2.5. Drošība 25
2.6. Docker un citi DevOps rīki programmatūras izstrādes procesu automatizācijas kontekstā 26
3. DOCKER virtualizācijas prototips 28
3.1. Problēmvides raksturojums 28
3.2. Uzdevuma nostādne 30
3.3. Risinājuma izstrāde 31
3.3.1. Produkcijas vides izveide un testa datu sagatavošana 31
3.3.2. Testa vides izveide 35
3.3.3. Piegādes uzlikšana testa vidē 36

3.3.4. Automatizācijas izstrāde 37
3.4. Rezultāti 41
Nobeigums 45
Pateicības 47
Literatūra 48

TERMINU UN SAĪSINĀJUMU VĀRDNĪCA

Saīsinājums	**Angļu valodas termins**	**Latviešu valodas termins ([Termini])**
API	Application programming interface	Lietojumprogrammu saskarne
BFS	Boot Filesystem	Palaišanas failu sistēma
FTP	File Transfer Protocol	Failu pārsūtīšanas protokols
GB	Gigabyte	Gigabaits
HTTP	Hypertext Transfer Protocol	Hiperteksta pārsūtīšanas protokols
IT	Information Technology	Informācijas tehnoloģija
LXC	Linux Containers	Linux konteineri
NIS	Network Information Service	Tīkla informācijas serviss
OS	Operating System	Operētājsistēma
RootFS	Root Filesystem	Saknes failu sistēma
SDK	Software Development Kit	Programmatūras izstrādes komplekts
SSH	Secure Shell	Drošais čaulas protokols
TCP	Transmission Control Protocol	Pārraides vadības protokols;
VDI	Virtual Disk Image	Virtuālais diska attēls
VHD	Virtual Hard Disk	Virtuālais cietais disks
VM	Virtual Machine	Virtuālā mašīna
VMDK	Virtual Machine Disk	Virtuālās mašīnas disks
VMM	Virtual Machine Monitor	Virtuālās mašīnas monitors

IEVADS

Jebkāda veida datortehnikas izmantošana sastāv no noteiktu uzdevumu izpildes uz fiziskiem resursiem, kā procesoru, atmiņu, videokarti, tīkla iekārtām, vai tā būtu vienkārša matemātiska darbība, vai arī sarežģīta lietojumprogramma. Laika gaitā datortehnikas fizisko resursu jaudas ir strauji augušas, tāpēc bieži lietojumprogrammas spēj izmantot tikai nelielu daļu no tām pieejamajiem resursiem. Šī iemesla dēļ fiziskie resursi tika virtualizēti dodot iespēju vairākām lietojumprogrammām vienlaikus lietot tikai nelielu daļu no vieniem un tiem pašiem fiziskajiem resursiem uz vienas un tās pašas fiziskās iekārtas. Šo procesu, kad tiek simulēti fiziskas datorkomponentes resursi, tad arī sauc par virtualizāciju [Singh, 2004].

Virtualizācijas pielietojums informācijas tehnoloģiju (IT) sistēmās kļūst aizvien plašāks sakarā ar ieguvumiem ko tā spēj sniegt. Šos ieguvumus saskata arī uzņēmumi, kuri ikdienā savus darbiniekus nodrošina ar jebkādiem ar IT saistītiem pakalpojumiem. Kā galvenās serveru virtualizācijas priekšrocības var minēt:

- zemākas IT infrastruktūras uzturēšanas izmaksas;
- atvieglota rezerves kopiju izveide un atkopšana;
- mazāks vai pilnīgi likvidēts sistēmu atslēgumu laiks;
- iespējas ātri iedalīt resursus jaunām lietojumprogrammām un sistēmām;
- iespēja automatizēt IT sistēmu ieviešanas un uzturēšanas procesus.

Aptaujas liecina, ka 2016. gadā jau 76 procenti uzņēmumu izmanto serveru virtualizācijas tehnoloģijas, savukārt 9 procenti plāno tās lietot [Spiceworks, 2016].

Virtualizācijas tehnoloģiju sniegtās iespējas plaši izmanto ar programmatūras izstrādi saistīti uzņēmumi. Klasisks modelis programmatūru izstrādē ir izveidot un uzturēt vairākas virtualizācijas vides dažādiem izstrādes procesa posmiem: izstrādes, testa, kvalitātes nodrošināšanas, pirms-produkcijas un produkcijas vides. Lai nodrošinātu kvalitatīvas programmatūras piegādes un izvairītos no klasiskas "strādā pie mums; tā jūsu IT cilvēku problēma tagad" problēmas ir jāiegulda daudz laika un resursus, lai visas virtuālās vides spētu uzturēt pēc iespējas līdzīgākas produkcijas videi. Tāpat izstrādātājiem tiek pieprasīts aizvien ātrāk piegādāt jaunas programmatūras funkcijas, nodrošinot netraucētu sistēmas darbību.

Programmatūras izstrādātāju vidū milzīgu popularitāti ir iemantojuši Docker konteineri. Datalog veiktajā pētījuma konstatēts, ka 2016. gada griezumā Docker adoptācija kompānijās pieaugusi par 40% [Datalog, 2017]. Šī tehnoloģija risina augstāk minētās problēmas, atvieglo un ļauj automatizēt dažādus ar programmatūras izstrādi saistītus procesus.

Docker ir atvērtā koda virtualizācijas dzinis, kas automatizē jebkādas lietojumprogrammas iepakošanu vieglā, portatīvā un pašpietiekamā konteinerī, tā nosūtīšanu un uzstādīšanu [Docker, 2017]

Piemēram, lai uzģenerētu virtuālo mašīnu (VM) ir nepieciešamas 10 minūtes. Docker gadījumā jauna konteinera ģenerēšana prasa vien pāris sekundes, kas ir milzīgs ieguvums nepārtrauktās piegādes un testēšanas modeļos [Anderson, 2015].

Par bakalaura darba mērķi tika izvirzīts izpētīt Docker virtualizācijas tehnoloģiju iespējas programmatūras ražošanas procesa automatizācijai un izstrādāt prototipu, kas automatizē noteiktu programmatūras ražošanas procesa daļu.

Mērķa sasniegšanai tika noteikti sekojoši uzdevumi:

1. Apskatīt industrijā plašāk lietotos virtualizācijas risinājumus;
2. Izpētīt Docker konteinerus un to izmantošanas iespējas programmatūras ražošanas procesu automatizācijai;
3. Izmantojot Docker tehnoloģiju, izstrādāt prototipu programmatūras ražošanas procesu soļu automatizācijai;
4. Notestēt prototipa darbību, novērtēt ieguvumus un nepilnības, noteikt attīstības virzienu.

Saskaņā ar izvirzīto mērķi bakalaura darbs sastāv no 3 galvenajām nodaļām un secinājumiem.

Darba pirmajā nodaļā apskatītas 2 galvenās virtualizācijas pieejas. Vispirms tiek dots ieskats par virtuālo mašīnu virtualizāciju, aprakstītas populārākās VM tehnoloģijas - WMware, VirtualBox un Hyper-V. Tālāk tiek aplūkota uz konteineriem bāzēta virtualizācija, minot tās priekšrocības un trūkumus salīdzinājumā ar VM.

Otrajā nodaļā ir plaši izklāstīts par Docker tehnoloģiju – vēsturi, arhitektūru, veiktspēju un drošību. Tā kā Docker tehnoloģija iekļaujas DevOps kultūrā, ir dots neliels ieskats DevOps kustībā, tās principos un filozofijā, kā arī tiek apskatīti mūsdienās populāri DevOps rīki, kas mijiedarbībā ar Docker konteineriem, tiek izmantoti ražošanas procesu automatizācijai.

Trešājā nodaļā ir aprakstīta Docker virtualizācijas prototipa izstrāde. Ir apskatīti programmatūras testēšanas vides izveides, testa būvējumu uzstādīšanas, testa vides atjaunošanas procesi, kā arī veikta šo procesu automatizācija. Ir aprakstīti izstrādātā prototipa ieguvumi un tālākie attīstības virzieni.

1. VIRTUALIZĀCIJA

1.1. Hipervizori

Virtualizācija ir ietvars vai metodoloģija, kas izmantojot tādus konceptus vai tehnoloģijas, kā datoraparatūras un programmatūras nodalīšana, laika dalīšana, daļēja vai pilna mašīnas simulācija, emulācija un daudzas citas, sadala datortehnikas fiziskos resursus vairākās izpildes vidēs [darba autora tulkojums, Singh, 2004].

Virtuālā mašīna ir fiziskas mašīnas vai izpildes vides realizācija programmatūrā, kas dod iespēju uzturēt vairākas virtualizācijas vides uz vienas fiziskas mašīnas. Visas virtualizācijas vides ir izolētas viena no otras, kā arī no reālās mašīnas, sniedzot lietotājam sajūtu, ka viņš patiešām lieto un tieši piekļūst reālajai mašīnai. Industrijā visplašāk virtuālās mašīnas tiek pielietotas šādiem mērķiem:

- Serveru konsolidācija, kas nozīmē daudzu serveru izvietošana uz mazāka skaita fiziskām mašīnām, instalējot tos uz VM. Šī serveru izvietošana metode ļauj pamatīgi ietaupīt fizisko serveru iegādes un uzturēšanas izmaksas;
- Kļūdu un ielaušanās risku izslēgšana. Piemēram, VM dod iespēju izolētā vidē lietot nedrošas lietojumprogrammas, palaist aizdomīgus izpildes kodus, nebaidoties ietekmēt fiziskās mašīnas. Šis pielietojums ir noderīgs, kad nepieciešams veidot drošas datošanas vides gala lietotājiem;
- Sistēmu migrācija. VM dod iespēju viegli migrēt no vienas fiziskās mašīnas uz citu, samazinot iespēju rasties programmatūras un dzelžu saderības kļūdām;
- Atkļūdošana un testēšana. VM ļauj izolētā vidē veikt visdažādākos programmatūras testus un atkļūdošanu. Pēc nepieciešamības var izmantot VM momentuzņēmumus un atgriezties uz sistēmas sākumstāvokli un veikt testa procesus pa jaunam;
- Augstas pieejamības lietojumsistēmas. Lietojot augstas slodzes tehnoloģijas kā kļūmpārleces klasteri (failover cluster), var nodrošināt, ka uz virtuālajām mašīnām esošie serveri bez pārtraukumiem būs pieejami lietotājiem;
- Dinamiska resursu pārraudzīšana. VM atbalsta manuālu vai automātisku resursu iedalīšanu, ja tām palielinās noslodze, trūkst resursu;
- Virtuālā aparatūra. VM mašīnas var piedāvāt virtualizētu aparatūru, kādu fiziski varbūt nebūtu iespējams iegādāties, piemēram, dažādi virtuālie Ethernet adapteri, maršrutētāji.

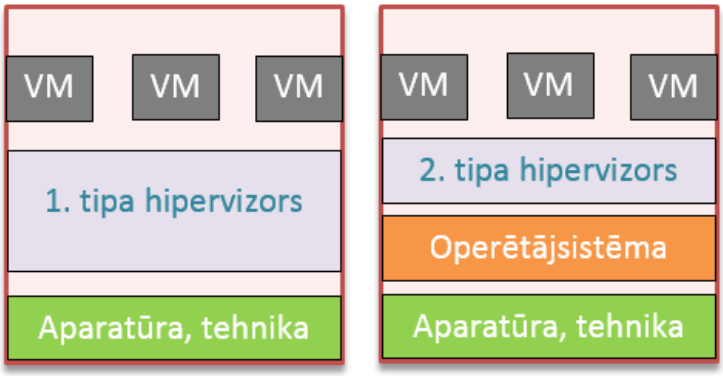
Lai varētu apjaust kādas vēl iespējas sniedz virtuālās mašīnas, ir nepieciešams apskatīt, kā šī tehnoloģija ir uzbūvēta. VM pamatā ir virtuālo mašīnu pārraugs jeb hipervizors, kas, abstrakti sakot, ir programmatūras slānis, kas atrodas zem virtuālajam mašīnām un virs fiziskās mašīnas. Šī slāņa uzdevums ir pārraudzīt resursu un atmiņas iedalīšanu virtuālajām mašīnām, nodrošinot, ka tās vienu otru netraucēs. Hipervizora darbības jēgu var definēt ar trīs raksturiezīmēm:

- Nodrošināt tādu vidi, kas ir identiska fiziskajai videi;
- Nodrošināt šo vidi ar pēc iespējas zemāku veiktspējas ietekmi;
- Saglabāt pilnīgu kontroli pār sistēmas resursiem [Popek, Goldberg, 1974, 412].

Hipervizorus var iedalīt divos tipos. Pirmā tipa hipervizors darbojas pa tiešo ar aparatūru, kamēr otrā tipa hipervizors darbojas kā lietojumprogramma uz kādas operētājsistēmas (OS) (Attēls 1.1).

Pirmā tipa, ko mēdz dēvēt arī par tīra-metāla (bare-metal) hipervizoru, pa tiešo komunicē ar fiziskajiem resursiem jeb aparatūru, tāpēc tas ir daudz efektīvāks par otrā tipa hipervizoru. Tāpat tas tiek uzskatīts par daudz drošāku, jo VM jeb viessistēmai (guest OS) nav iespējams to ietekmēt. Viessistēma var bojāt pati sevi, bet tas neietekmēs pārējo VM un hipervizora darbību. Kā populārākos pirmā tipa hipervizorus var minēt VMware ESX, Microsoft Hyper-V un Citrix XenServer.

Otrā tipa hipervizors jeb OS līmeņa virtualizācija, ir lietojumprogramma, kas darbojas kādā no OS. Viena no tā priekšrocībām ir plašais aparatūras atbalsts, jo tas balstās uz kādu no populārajam OS. Bieži šī tipa hipervizoru uzstādīšana vienkārša un ātra, jo aparatūras konfigurāciju jau ir paveikusi saimnieksistēma (host OS). Šī modeļa lielākais mīnuss ir veiktspēja, jo hipervizora resursu komunikācijai ar aparatūru starpā ir OS. Kā otru mīnusu var minēt, kas tas ir mazāk drošs, jo ir daudz lielāka iespēja, ka kaut kas var nojukt sakarā saimnieksistēmas darbību. Piemēram, atjauninājumu izraisīts sistēmas restarts restartēs arī visas VM [Portnoy, 2012, 21]. Kā populārākos šī tipa hipervizora piemērus var minēt VMware Workstation, Microsoft Virtual PC un Oracle VM VirtualBox.



Attēls 1.1. Salīdzinājums starp 1. un 2. tipa hipervizoriem.

Lai saprastu, kā atšķiras abu hipervizoru tipu sniegtās iespējas, apskatīsim trīs populārus virtualizācijas risinājumus.

1.1.1. WMware Workstation

1999. gadā VMware kompānija izlaida jaunu virtualizācijas produktu ar nosaukumu WMware Workstation. Programmatūra būtībā ir otrā tipa hipervizors, kas darbojas uz Windows un Linux operētājsistēmām. Tā lietotājiem dod iespēju uzstādīt un vienā laikā lietot vairākas VM uz vienu darbstaciju. Hipervizors atbalsta vairāk kā 200 operētājsistēmu veidu uzstādīšanai viessistēmās.

Virtualizācijas rīks nodrošina saimnieksistēmas tīkla adapteru, fizisko diskdziņu un USB ierīču apvienošanu (bridging) ar VM. Tas spēj simulēt diskdziņus, piemontēt ISO diska attēlus. Kā viena no noderīgākajām funkcijām var minēt iespēju jebkurā brīdī saglabāt VM momentuzņēmumus, uz kuriem pēc nepieciešamības var atgriezties.

Programmatūras izstrādātāji šo risinājumu var izmantot, ja nepieciešams izstrādāt un testēt lietojumsistēmu uz vairākām OS, kā arī DevOps un spējās izstrādes (agile) darbplūsmām [VMware, 2017].

1.1.2. Oracle VM VirtualBox

Kā lielisku bezmaksas un atvērtā koda alternatīvu iepriekšminētajam produktam, 2007. gadā kompānija Innotek izlaida virtualizācijas programmatūru Virtualbox, kura izstrāde pašlaik ir Oracle pārvaldībā.

Arī šis produkts ir otrā tipa hipervizors. VirtualBox ir pieejams uz Windows, Linux, Mac OS X un Solaris operētājsistēmām.

VirtualBox salīdzinot ar VMware Workstation papildus piedāvā tādas funkcijas kā:

- Komandrindas atbalsts mijiedarbojoties ar VM,
- VM videoieraksts,
- VM diska attēlu šifrēšana.

Risinājums piedāvā emulēt cietā diska attēlus trīs formātos – VDI, kas ir VirtualBox formāts, VMDK, kas ir VMware produktu formāts, un VHK, kas ir MS Windows formāts. Tas nozīmē, ka ar šo rīku ir iespējams izmantot diska attēlus, kas iepriekš ir izveidoti citos virtualizācijas risinājumos [VirtualBox, 2017].

1.1.3. Microsoft Hyper-V

Microsoft Hyper-V tehnoloģija jeb agrāk zināms kā Windows Server Virtualization tika izlaists 2008. gadā izmantojot Windows atjauninājumu (Windows Update) servisu.

Šis virtualizācijas risinājums pamatā ir pirmā tipa hipervizors, bet to ir iespējams arī uzstādīt, kā otrā tipa hipervizoru uz Windows Server OS. Tā lielākais ieguvums ir veiktspēja. Ar Hyper-V ir iespējams virtualizēt Windows un nelielu skaitu Linux operētājsistēmas.

Līdzīgi, kā otrā tipa hipervizori, arī Microsoft risinājums dod iespēju veidot VM momentuzņēmumus. Tas piedāvā tādas noderīgas funkcijas kā dzīvā migrācija (live migration) - dod iespēju pārvietot VM no vienas fiziskas mašīnas uz citu neietekmējot VM pieejamību lietotājiem, dinamiskā atmiņa (Dynamic Memory) – palīdz pārvaldīt resursus un uzlabo uzticamību VM restartēšanai [Hyper-V, 2016].

Šī tehnoloģija pamatā ir izstrādāta priekš serveru virtualizācijas, tāpēc programmatūras ražošanas kontekstā var noderēt, piemēram, Microsoft SharePoint risinājumu izstrādei.

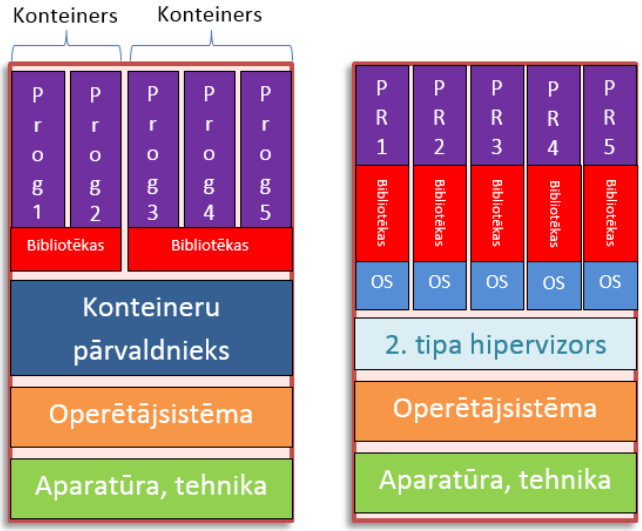
1.2. Konteineri

Uz hipervizoriem bāzētas virtualizācijas pamatā ir plānā kodola (thin kernel) slānis, kas VM dotās instrukcijas nodot apakšējam aparatūras slānim. Katrai VM ir nepieciešama pilnīga OS, ieskaitot kodolu, kas nozīmē, ka tās ir gandrīz pilnīgi izolētas viena no otras.

Uz konteineriem bāzētā virtualizācijā visas virtuālās vides lieto vienu un to pašu saimnieksistēmas kodolu un tai nav nepieciešamība pēc hipervizora. Šis virtualizācijas koncepts tiek izmantots, ja ir nepieciešama optimāla efektivitāte, bet OS izvēle nav tik svarīga [Container, 2015].

Konteineri salīdzinājumā ar hipervizoriem nodrošina augstāku veiktspēju, jo nav nepieciešami papildus resursi katras pilnas OS darbināšanai. Konteineri piedāvā virtualizācijas platformu, kur katrs konteineris var lietot savu OS, bet tiem būs jādalās ar vienu kodolu (Attēls

1.2). Katra vienība darbojas izolētā un drošā vidē. Konteineri ir mazāki un ātrāki, tāpēc to izveide, migrēšana vai jebkāda cita darbība ir "zibenīga". Mazais izmērs arī nodrošina iespēju uz vienas OS izvietot daudz vairāk konteineru kā VM.



Attēls 1.2. Salīdzinājums starp konteineru un hipervizora bāzētām virtualizācijām.

Tādi konteinerizācijas rīki, kā LXC, Docker, FreeBSD jails, OpenVZ un Solaris Containers ir uzbūvēti pamatā izmantojot trīs Linux kodola apakšsistēmas – Chroot, Namespaces un Cgroups.

1.2.1. Chroot Jails

Linux OS attīstoties radās ideja par procesu izolāciju no saimnieksistēmas failu sistēmas. 1979. gadā ideja tika realizēta izveidojot jaunu Linux komandu (sistēmas vaicājumu) un utilītprogrammu – chroot, kas atšifrējot nozīmē "change root". Šis vaicājums dod iespēju definēt jaunu saknes katalogu (root directory) kādam procesam un tā apakšprocesiem nodrošinot, ka tie nevar piekļūt failiem saknes kataloga ārpusē. Tikmēr lietojumprogrammas no ārpuses bez ierobežojumiem var piekļūt definētajam katalogam [Chroot, 2017].

Šo mākslīgi izveidot saknes katalogu dēvē par "chroot jail" jeb saknes kataloga cietumu. Ir svarīgi, ka nevienam no "cietumā" esošajiem procesiem nav iespējas iegūt galvenā lietotāja (root) tiesības, jo tas dotu iespēju tiem izlauzties un ietekmēt saimnieksistēmu. Šo darbību, kad process izlaužas no cietuma sauc par "jailbraik" [Chroot Jails, 2016].

Mūsdienās šo tehnoloģiju izmanto, lai izveidot vienkāršas, izolētas vides, lai, piemēram, testētu nezināmas izcelsmes vai nestabilas lietojumprogrammas, kā arī lai ierobežotu lietotājus, kas pieslēgušies serverim caur SSH vai FTP protokoliem.

1.2.2. Namespaces

Tā kā Chroot apakšsistēmai ir samērā daudz mīnusu, kā, piemēram, nepieciešamība dublēt failus un drošības problēmas, Linux procesu izolācija tika uzlabota izveidojot jaunas apakšsistēmas un sistēmas vaicājumus. Namespace ir Linux kodola mehānisms, kas izolē procesus vienu no otra.

"Namespace tehnoloģijas mērķis ir abstrakti "ievīstīt" kādu no globāliem sistēmas resursiem tā, ka katram procesam ir iedalīta sava izolēta vienība no attiecīgā resursa [darba autora tulkojums, Kerrisk, 2013]".

Mūsdienās Linux atbalsta sešus Namespace veidus:

1. PID – izolē PID procesus;
2. USER – izolē lietotājus un grupas;
3. MOUNT - izolē montēšanas punktus;
4. NETWORK – izolē tīkla adapterus, portus;
5. IPC – izolē SystemV IPC un POSIX ziņojumus;
6. UTS – izolē saimnieksistēmas un tīkla informācijas servisa (NIS) domēnus.

Modernas, uz konteineriem bāzētas virtualizācijas tehnoloģijas, piemēram, Docker un LXC izmanto visus 6 veidus brīdī, kad tiek startēta kāda lietojumprogramma [Yemelianov, 2017].

1.2.3. Cgroups

Procesu izolācijas mehānismiem konteineru virtualizācijā ir ļoti būtiska loma, bet ar to vien nepietiek. Ja izolētā vidē startējam lietojumprogrammu, mums jābūt drošiem, ka tai ir iedalīti pietiekami daudz resursu vai arī ka tai nav iedoti par daudz resursu, kā rezultātā būtu traucēta saimnieksistēmas darbība. Šo uzdevumu veic Linux kodola apakšsistēma ar nosaukumu Cgroups jeb kontroles grupas.

2006. gadā divi kompānijas Google darbinieki uzsāka darbu pie Cgroups mehānisma, kuru 2008. gadā oficiāli pievienoja Linux kodolam. Mehānisms sistēmas administratoriem dod iespēju kontrolēt sistēmas resursus, kurus izmanto procesu grupas. Grupas var būt sakopotas mežam līdzīgā struktūrā, kur koku saknes ir katras sistēmas pamata grupas [Menage, 2004].

Cgroups mehānisms sastāv no divām sastāvdaļām – kodola un apakšsistēmas. Apakšsistēmas ir kontrolieri, kas atbild par individuāliem sistēmas resursu tipiem. Kā visbiežāk lietotās apakšsistēmas var minēt:

- CPU – procesiem dod pieeju procesora resursiem;

- BLKIO – kontrolē pieeju ievadizvades ierīcēm, ierobežojot rakstīšanas un lasīšanas ātrumus;
- CPUSET – norāda individuālus procesora kodolus, kurus attiecīgā procesu grupa drīkst izmantot;
- MEMORY – pārvalda atmiņas iedali procesu grupām;
- DEVICES – dod vai ierobežo pieeju ierīcēm [Yemelianov, Cgroups, 2017].

Vienkāršojot Cgroups mehānismu var teikt, ka specifiski procesi tiek iedalīti grupās, kuras tālāk tiek "iespiestas" kādā no 12 apakšsistēmām.

1.2.4. Salīdzinājums ar VM

VM virtualizācijā katrai VM tiek izveidota jauna OS vienība. Tās lielākā priekšrocība ir iespēja izveidot pilnīgi atšķirīgu, ar citu tipa OS, viessistēmu no saimnieksistēmas, bet tam ir arī vairāki trūkumi.

Kā vienu no trūkumiem var minēt fizisko resursu virstēriņu (overhead), kas, piemēram, rodas, ja kādai no VM ir jāgaida uz resursiem, ko attiecīgajā brīdī izmanto cita VM. Gaidīšanas laiks nozīmē papildus fizisko resursu patēriņu [Minas L., Ellison B., 2009]. Vislielākais virstēriņš rodas virtualizētu disku un tīkla adapteru dēļ, kas ir arī konstatēts Peter Senna Tschudin [Tschudin, 2012] veiktā pētījuma rezultātos. Rezultātos parādīta procentuāla vērtība, kas atspoguļo resursu virstēriņu. Piemēram, ja uz vienas fiziskas mašīnas darbojas tikai viena Linux VM un tai ir pieeja 100% procesora resursiem, tad reāli VM saņem 94,67% procesora jaudas (Tabula 1.1).

Tabula 1.1

Resursu virstēriņš uz Microsoft Hyper-V tehnoloģiju izveidotās VM [darba autora tulkojums, Tschudin, 2012].

VM operētājsistēma	Linux	Windows
CPU procesi	5,43%	3,10%
Diska ievadizvades procesi	24,07%	20,96%
Tīkla ievadizvades procesi	16,50%	24,34%

Resursu virstēriņš parādās arī uz konteineriem bāzētā virtualizācijā, bet tas ir pavisam minimāls, jo konteineri izmanto saimnieksistēmas resursus tieši. Minimālo virstēriņu izraisa iepriekš aprakstītā kontroles grupa MEMORY, jo veic ļoti sīku atmiņas izmantošanas uzskaiti

[Petazzoni, 2013]. Tāpat ar tīkla izmantošanu saistīti procesi izraisa minimālu virstēriņu, ja konteineram ir izveidots virtualizēts iekšējais tīkls [Wachowicz, 2016].

Otrs VM trūkums ir to lielais izmērs sakarā ar iekļauto pilno OS. Piemēram, Docker konteineris ar Ubuntu 16.04 OS aizņem tikai 120,8 MB diska vietas, bet tā pati OS uzstādīta VM aizņems vismaz 3 GB diska vietas.

Tiešs abu virtualizācijas tehnoloģiju salīdzinājums ir atspoguļots tabulā Tabula 1.2.

Tabula 1.2

Virtuālo mašīnu un konteineru tiešs salīdzinājums

[darba autora tulkojums, Pethuru u.c., 2016].

Virtuālās mašīnas	**Konteineri**
Aparatūras līmeņa virtualizācija	Operētājsistēmas līmeņa virtualizācija
Smagnējas	Viegli
Limitēta veiktspēja	Saimnieksistēmai identiska veiktspēja
Pilnīgi izolētas, drošākas	Izolācija tikai procesu līmenī, mazāk droši
Laikietilpīga nodrošināšana un uzturēšana	Reāla laika nodrošināšana un mērogojamība

Veiktspējas kontekstā abas tehnoloģijas salīdzinātas IBM pētījumā [Felter, 2014], kurā tika pētīta procesora un atmiņas caurlaide, kā arī diskvietas un tīkla veiktspējas. Rezultātos redzams, ka Docker startēšanās laiks ir par 50 reizēm ātrāks kā VM. Pētījumā nobeigumā secināts, ka konteineru veiktspēja ir vienāda vai labāka ar VM gandrīz visos gadījumos.

Tā kā konteineru virtualizācija daudzos aspektos ir pārāka par uz hipervizoriem balstītu virtualizāciju, tad būtu vērts tālāk apskatīt pašlaik milzīgu popularitāti ieguvušo Docker tehnoloģiju.

2. DOCKER

Docker tehnoloģiju vislabāk raksturo kompānijas dibinātāja Solomon Hykes teiktais: "Tas viss sākas ar kaut ko vienkāršu un mazsvarīgu. Konteineris ir tikai parasta kastīte, bet ar šo kastīti, jūs varat kopā sapakot tik ļoti daudz programmatūras produktus, platformas un sistēmas, ka katra spēj netraucēti darīt sev paredzētās darbības. Beigās šie konteineri ir visur, un tu tos vari pārvietot jebkur.

Katrs Docker konteineris ir "šūna" planētas izmēra organismā, ko saucam par internetu. Fiziskas mašīnas, kabeļi, maršrutētāji un cieti diski – šie ir tikai kuģi, kas nodrošina, lai šūnas varētu glabāt, izskaitļot un apmainīties ar ziņojumiem" [darba autora tulkojums, Hykes, 2013].

Docker ir konteineru virtualizācijas platforma, kas dod iespēju viegli sapakot un izplatīt lietojumsistēmu vai programmu kopā ar visu tai nepieciešamo atbalsta programmatūru. Šī tehnoloģija atvieglo programmkoda transportēšanu uz kādu no programmatūras izstrādes vidēm. [Docker, 2017].

Industrijā Docker tehnoloģija tiek pielietota sekojošajās sfērās:

- Augstas veiktspējas programmatūras lielizmēra izvietošana (hyper scale deployment);
- Mikroservisu (micro-services) arhitektūras nodrošināšana;
- Platforma-kā-serviss (PaaS) infrastruktūras izveide;
- Dalīšanās ar izstrādes vidēm programmatūras izstrādes procesos;
- Testēšanas vides;
- Pētījumu validācija – pētījumu pārsūtīšana to pārbaudei ietverot nepieciešamo kodu un rīkus [Ashraf, 2016].

2.1. Vēsture

2013. gada 15. martā Python izstrādātāju konferencē, kas norisinājās Kalifornijā, kompānijas dotCloud dibinātājs Solomon Hykes pirmo reizi pasauli iepazīstināja ar Docker tehnoloģiju, kā arī paziņoja, ka tā būs pieejama visiem kā atvērtā koda programmatūra. Paziņojums sagādāja lielu pārsteigumu, jo par tehnoloģiju iepriekš zināja tikai dotCloud darbinieki un 40 cilvēki ārpus kompānijas, kuriem bija dota iespēja "paspēlēties" [Matthias, Kane, 2015].

Pāris nedēļu laikā pēc konferences šai virtualizācijas tehnoloģijai tika pievērsta milzīga preses uzmanība, jo pirms Docker parādīšanās nebija pieejami risinājumi, kas garantētu

programmatūras portativitāti. 2013. augustā tika izlaista interaktīva pamācība, kuru mēneša laikā izmēģināja 10 000 izstrādātāju [Martin, 2015]. Tajā pašā gadā "tehnoloģiju giganti" Google, Baidu, Yandex, Ebay un Cloudfare integrēja Docker savos pakalpojumos [Golub, 2014].

2014. gadu Docker uzsāka ar 15 miljonu dolāru finansējumu, kas deva kompānijai iespēju pastiprināti investēt Docker atvērtā koda projektā un tā atbalsta pasākumos – dokumentācijas, treniņi, konferences. Liela daļa naudas līdzekļu tika novirzīti platformas kopienas paplašināšanai, kas deva iespēju uzlabot Docker Hub reģistru un, piemēram, integrēt GitHub izejas koda sistēmu [Golub, 2014].

2014. gada martā tika izslaista Docker 0.9 versija, kurā tika nomainīta pamata izpildes vide no LXC uz Docker pašu izstrādāto Libcontainer, kas nodrošināja papildus neatkarību no OS un stabilitāti. Libcontainer dzinis ir izstrādāts izmantojot Google izveidoto programmēšanas valodu Go [Solomon, 2014].

Kā milzīgs pienesums konteinerizācijas tehnoloģijas attīstībai bija 2014. gada 15. oktobra paziņojums no Microsoft par Docker platformas iekļaušanu Windows ekosistēmā. Galvenās sadarbības komponentes bija:

- Docker Engine atbalsts Windows Server OS un Windows Server attēlu failu (image files) iekļaušana Docker Hub reģistrā;
- Docker Hub reģistra iekļaušana Microsoft Azure platformā;
- Docker lietojumprogrammu saskarnes (API) integrācija Azure platformā;
- Docker dziņa Windows Server versijas izstrāde notiks Docker atvērtā koda projekta aizgādībā [Microsoft 2014].

Otrs "lielais grūdiens" Docker popularitātei 2014. gadā bija tā oficiāla iekļaušana Ubuntu 14.04 LTS operētājsistēmā, kas nozīmēja, ka katrs Ubuntu serveris atradās 3 komandrindu attālumā no iespējas izveidot un palaist virtualizācijas konteinerus [Barbier, 2014].

Sakarā ar lielo Docker popularitāti, pieauga arī pieprasījums pēc laba konteineru pārvaldības rīka. 2014. gada jūnijā tika izlaists konteineru "orķestrēšanas" (orchestration) rīks Kubernetes, kas pēc Slant vērtējuma [Slant, 2017] joprojām ir labākais rīks šajā kategorijā. 2015. gada 3. novembrī Docker izlaida paši savu pārvaldības rīku Docker Swarm, kas, piemēram, deva iespēju veidot un uzturēt Docker kļūmpārlēces klāsterus (failover cluster) [Luzzardi, 2015].

Kopienas paplašināšanas nolūkos 2014. gadā norisinājās pirmā DockerCon konference, kas paredzēta "nākošās paaudzes" programmatūras, kas veidota ar konteineru palīdzību,

izstrādātājiem. Pasākumā liels uzsvars bija likts uz Docker izmantošanas piemēriem lielajās tehnoloģiju kompānijās kā Yelp, Groupon, Chef un Google. Pašlaik konferences tiek organizētas katru gadu dažādās pasaules malās un pulcē lielu dalībnieku skaitu [DockerCon].

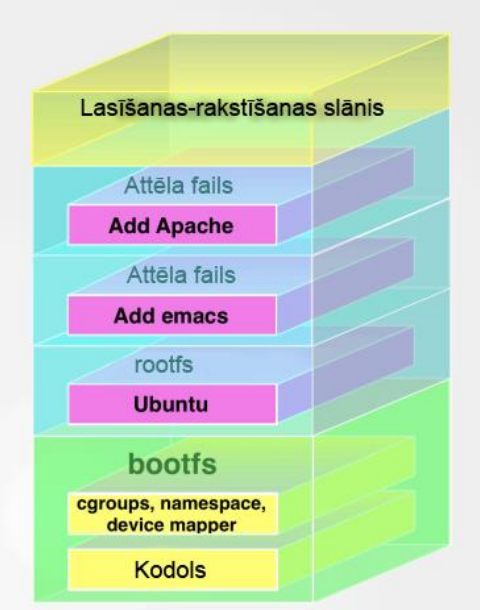
2016. gada DockerCon konferencē Docker vadītājs Ben Golub prezentēja statistikas datus, kas dod labu priekštatu par to, cik šī tehnoloģija ir kļuvusi populāra:

- 460 tūkstoši "konteinerzētu" lietojumprogrammu, 3100% pieaugums 2 gadu laikā;
- 4 miljardi reižu konteineri ir bijuši novilkti no publiskajiem reģistriem;
- 95 tūkstoši ar Docker saistīti projekti, 1300% pieaugums 2 gadu laikā;
- 125 tūkstoši Docker Meetup grupas dalībnieku visā pasaulē [Arijs, 2016].

2.2. Atbalsta tehnoloģijas

Tā kā Docker ir konteineru virtualizācijas rīks, tad tas izmanto tās pašas resursu izolācijas Linux kodola apakšprogrammas, kas aprakstītas 1.2 apakšnodaļā – Cgroups un Namespaces. Papildus šīm divām tehnoloģijām tiek izmantots UnionFS serviss.

Kad tiek startēts konteineris, Docker izmanto mehānismu, ko sauc par UnionFS pievienošanu (mount), kas tiek reprezentēta kā loģiskā failu sistēma, kas apvieno dažādas failu sistēmas un katalogus Attēls 2.1. Attēla fails (image) sastāv no vairākām failu sistēmu kārtām, kas atrodas viena virs otras. Apakšējais slānis ir palaišanas failu sistēma (BFS), kas līdzinās tipiskai Linux OS BFS. Virs tās atrodas saknes failu sistēma (RootFS), kas var būt viena vai vairākas OS. Pārējie slāņi var saturēt dažādas programmatūras komponentes. Pašā augšpusē tiek veidots lasīšanas-rakstīšanas slānis, kurš saglabā visas konteinerī veiktās izmaiņas [Ashraf, 2016].



Attēls 2.1. Union failu sistēmas uzbūve [darba autora tulkojums, Ashraf, 2016].

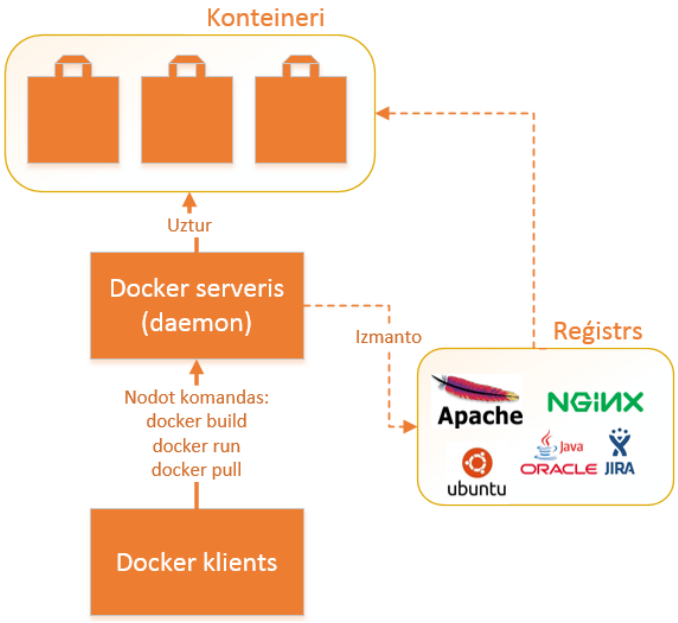
2.3. Arhitektūra

2.3.1. Klienta-servera modelis

Docker ir ļoti "spējīga" tehnoloģija, kas nozīmē arī augstu tās sarežģītības pakāpi. Tā balstās uz klienta-servera arhitektūru jeb Docker klients komunicē ar Docker dēmonu (daemon), kurš izveido, palaiž un izplata konteinerus. Abi procesi izmanto vienu un to pašu izpildāmo failu (executable). Nozīme ir tam, kā tiek izsaukta "docker" komandrinda. Bez šiem diviem procesiem vēl ir reģistra komponente, kur glabājas attēlu faili un to metadati [Pethuru 2016].

Dēmons klausās API pieprasījumus un pārvalda konteinerus, attēla failus, tīkla infrastruktūru un sējumus (volumes). Lai varētu kontrolēt Docker servisus, tas spēj arī komunicēt ar citiem serveriem.

Klients ir galvenais rīks, kuru izmantojot, lietotājs var komunicēt ar vienu vai vairākiem serveriem. Komandas tiek nodotas dēmonam izmantojot API Attēls 2.2 [Docker Architecture].



Attēls 2.2. Docker klienta-servera modelis [Docker Architecture].

Reģistrs ir satura glabāšanas un izplatīšanas sistēma, kurā atrodas Docker attēlu failu dažādas versijas. Reģistru var izmantot, ja nepieciešams:

- Stingrāk kontrolēt, kur glabājas attēla faili;
- Integrēt attēlu glabāšanu un izplatīšanu kādā iekšējas izstrādes darbplūsmā;
- Pilnībā kontrolēt attēla failu izplatību [Docker Registry].

Kā alternatīvu lokāli uzturētam reģistram, var, izmantot publisko, "mākonī" izvietoto Docker Hub, kas papildus piedāvā tādas iespējas, kā automātiski būvējumi (builds), organizāciju konti un pieeja plašam konteineru attēlu klāstam. Reģistrā ir iekļauti sertificēti repozitoriji no lielām tehnoloģiju kompānijām: Oracle, Red Hat, MySQL, Apache u.c. [Docker Hub].

2.3.2. Tīkla porti

Lai klients ar dēmonu spētu komunicēt, ir nepieciešams izmantot tīkla ligzdas (network sockets). Serveri ir iespējams nokonfigurēt klausīties vienu vai vairākas TCP vai Unix tīkla ligzdas, piemēram, vienu šifrētu un vienu nešifrētu TCP portu.

Docker dēmonam pēc noklusējuma ir uzstādīts šifrēts ports - 2375. Nešifrētai komunikācijai ir rezervēts 2376 ports, bet šo konfigurāciju ir viegli mainīt.

Drošības apsvērumu dēļ šifrētai komunikācijai Docker izmanto protokolu, kas atbalsta transporta slāņa drošību sākot ar versiju 1.0 (TLS1.0) un jaunākas [Daemon].

2.3.3. Komandrindu rīks

Galvenais interfeiss, ar ko saskaras Docker tehnoloģijas lietotāji, ir komandrindu rīks "docker", kas pēc noklusējuma ir iekļauts instalācijas pakotnē. Šīs rīks ir uzkodēts Go programmēšanas valodā, kuru var kompilēt un palaist jebkurā no izplatītajām OS.

Ar komandrindu palīdzību, piemēram, var veikt sekojošās darbības:

- "Uzbūvēt" konteinera attēla failus;
- No reģistra uz dēmonu atgādāt (pull) vai no dēmona uz reģistru aizgādāt (push) attēla failus;
- Izgūt žurnālfailus no attālās piekļuves servera;
- Docker serverī startēt konteinerus priekšplāna vai fona režīmā;
- Startēt komandrindas čaulu (shell) darbojošā konteinerī.

Ņemot vērā šī rīka plašo funkcionalitāti, ir ērti izveidot būvēšanas un izvietošanas darbplūsmas [Matthias, Kane, 2015].

2.3.4. API

Komandrindas nav vienīgais veids kā mijiedarboties ar Docker. Plašākas mijiedarbības iespējas piedāvā API, ar kura palīdzību iespējams kontrolēt jebkuru šīs tehnoloģijas aspektu.

Tā kā API tiek izmantots arī klient-servera modeļa komunikācijas nodrošināšanai, tad to, ko spēj klients, ir iespējams paveikt ar API. Lai piekļūtu API, var izmantot hiperteksta pārsūtīšanas protokolu (HTTP) un Python vai Go programmatūras izstrādes komplektus (SDK) [Docker API].

2.3.5. Konteineru tīklošana

Lai gan Docker tehnoloģija pārsvarā balstās uz procesiem, kas pa tiešo tiek startēti uz saimnieksistēmas, tīkla līmeņa procesu darbība ir nedaudz savādāka. Pēc noklusējuma Docker ir iebūvēti trīs tīklu veidi: bridge, null, host. Ja konteinera izveides brīdī netiek norādīts, kādā tīklā tas darbosies, tad tam tiks izmantots tilts (bridge).

Tilta tīkla darbības pamatā ir tas, ka katram konteinerim ir izveidots virtuālais Ethernet interfeiss, kam piesaistīta sava IP adrese. Interfeisi ir savienoti ar Docker tiltu, kas tālāk komunicē ar saimnieksistēmu. Lai no ārpuses varētu piekļūt konteineriem, katra konteinera portus ir iespējams piesaistīt saimnieksistēmas portiem.

Nulles (null) tīklā konteineriem netiek izveidots virtuālais interfeiss, savukārt saimnieka tīklā (host) konteineris izmanto tieši to pašu tīklu, ko saimnieksistēma, un tie savā starpā nav izolēti [Docker Networking].

2.4. Veiktspēja

Neliels tests ar jauna konteinera izveidošanu no eksistējoša attēla faila atklāj to mazo izmēru – 12 baiti. Mazais izmērs saistāms ar to, ka tas satur tikai norādi uz UnionFS glabātu attēla failu un konfigurācijas metadatus. [Docker Containers]. Salīdzinājumam, jaunas VM izveide no attēla faila var prasīt simtiem un vairāk megabaitu no diska vietas.

Konteineru "vieglums" dod iespēju tos izmantot situācijās, kur jaunas VM izveide būtu pārāk smagnēja veicamajam uzdevumam. Piemēram, jaunas un mazas MySQL datubāzes izveidei programmatūras testēšanas nolūkiem būtu muļķīgi veidot un konfigurēt jaunu VM.

Docker vieglums un ātrdarbība ir atspoguļota pētījumā "Evaluation of Docker Containers Based on Hardware Utilization [Preeth u.c., 2015]". Pētījumā ir notestēta Docker tehnoloģijas ātrdarbība salīdzinājumā ar saimnieksistēmu tādos aspektos, kā failu sistēma, CPU, atmiņa un tīkla ievadizvades procesi. Lai gūtu priekšstatu par konteineru virtualizācijas veiktspēju, nepieciešams sīkāk apskatīt zinātniskajā darbā iegūtos datus.

Failu sistēmas veiktspējas testam tika izmantota etalonmērījumu (benchmarking) veikšanas rīks Bonnie++, kas veidots C++ programmēšanas valodas bāzes. Kā redzams Tabula 2.1, izvades temps abās izpildes vidēs ir gandrīz vienāds. Atšķirība ir redzama tikai ievades tempa rezultātos, kas pierāda to, ka saimnieksistēmai ir pieejami vairāk resursu attiecīgajā aspektā.

Tabula 2.1

Bonnie++ testa izpildes rezultāti Docker konteinerī un saimnieksistēmā
[darba autora tulkojums, Preeth u.c., 2015].

Izpildes vide	**Faila izmērs**	**Secīgā izvade**		**Secīgā ievade**	
		Izvades temps	CPU laiks	Ievades temps	CPU laiks
Konteineris	40Mb	507Kb/s	98%	1351Kb/s	97%
Saimnieksistēma	40Mb	536Kb/s	98%	4388Kb/s	96%

Lai noskaidrotu sistēmas utilizācijas datus par CPU, atmiņu, diska vietas izmantošanu un tīklu, pētījuma autori izmantoja Python valodas bibliotēku ar nosaukumu Psutil. Atmiņas utilizācijas rezultāti abās izpildēs vidēs redzami Tabula 2.2. Kā redzams atmiņas lietojums ir gandrīz identisks, kas nozīmē, ka Docker pilnībā izmanto saimnieksistēmas resursus.

Tabula 2.2

Atmiņas lietojums Docker un saimnieksistēmas vidēs izteikts baitos
[darba autora tulkojums, Preeth u.c., 2015].

Izpildes vide	**Kopā**	**Izmantota**	**Brīva**
Konteineris	4018626560	1398411264	2620215296
Saimnieksistēma	4018626560	1395318784	2623307776

CPU un diska utilizācijas rezultāti abās izpildes vidēs ir gandrīz identiski. Būtiskākā atšķirība veiktspējā ir tīkla ievadizvades mērījumos, kas saistāms ar to, ka, visticamāk, pētījuma autori konteineri veidojuši ar noklusējuma tīkla veidu – tiltu (bridge).

Pēc ievāktajiem datiem un to analīzi, pētījuma autori secinājuši, ka Docker virtualizācijas tehnoloģijas veiktspēju var pielīdzināt saimnieksistēmas veiktspējai [Preeth u.c., 2015].

2.5. Drošība

Interneta resursos ir atrodams liels informācijas daudzums saistībā ar Docker drošības problēmām, kā rezultātā var rasties iespaids, ka Docker tehnoloģija nav droša un to nav ieteicams izmantot produkcijā. Lai saprastu, kāpēc šāda informācija "uzpeld", nepieciešams sīkāk apskatīt šo tehnoloģiju no drošības aspekta.

Apsverot Docker drošību ir jāņem vērā trīs galvenās jomas, kur var rasties problēmas:

- Uzrukuma iespēja Docker dēmonam;
- Drošības caurumi konteineru konfigurācijas profilā;
- Kodola drošība un tā atbalsts Namespaces un Cgroups apakšsistēmām.

Konteineru palaišana un uzturēšana ir iespējama pateicoties Docker dēmonam, kura darbībai nepieciešamas administratora (root) tiesības. Nepieciešamība pēc administratora tiesībām ir pāris spēcīgu Docker iespēju sekas, piemēram, portu sasaiste ar saimnieksistēmu un failu sistēmu pievienošana. Lai nerastos drošības problēmas, dēmona kontrole jānodrošina tikai uzticamiem lietotājiem. Ja kontrolei tiek izmantots API, tad jānodrošina šifrēts savienojums starp HTTP un dēmonu.

Runājot par Docker drošību, vērts arī pieminēt 1.2 nodaļā apskatītās Linux kodola apakšsistēmas. Namespaces piedāvā vistiešāko konteineru izolācijas formu, kas nozīmē, ka konteinerī esošie procesi neredz un neietekmē procesus citos konteineros vai saimnieksistēmā.

Cgroups, savukārt, nodrošina resursu uzskaiti un limitēšanu, kas nozīmē, ka vienam konteinerim nav iespējams apturēt visas sistēmas darbību iztērējot kādu no resursiem [Docker Security].

2.6. Docker un citi DevOps rīki programmatūras izstrādes procesu automatizācijas kontekstā

Docker plašā funkcionalitāte paver lielas tā izmantošanas iespējas tehnoloģiju industrijā. Kā izmantošanas piemērus var minēt:

- Atvieglota konfigurācija – dod iespēju sapakot vidi un tās konfigurāciju "kodā" formātā;
- Koda konveijera (pipeline) pārvaldība – nemainīga vide visos programmatūras izstrādes posmos, atvieglojot izstrādi un izvietošanu;
- Izstrādātāju produktivitātes uzlabošana – nodrošina pēc iespējas identiskāku izstrādes vidi produkcijai, kā arī ātra izstrādes vides izveide un uzturēšana;
- Atkļūdošana - konteineru versiju kontrole atvieglo programmatūras atkļūdošanas procesus [Suleman].

Augstāk minētie piemēri dod iespēju izmantot Docker tehnoloģiju, lai realizētu tādus DevOps kustības principus, kā infrastruktūra kā kods un automatizēta un momentāna izvietošana.

DevOps ir termins, kas apzīmē metodoloģijas kopumu, kas laika gaitā ir pārvērties par kustību un strauji izplatījies tehnoloģiju kopienā. To var definēt ar divām saistītām idejām.

Pirmkārt, DevOps ir piekopta prakse, kad IT infrastruktūras (IT operations) un izstrādes (developers) inženieri kopā ir iesaistīti visos produkta izstrādes cikla procesos. Tiek aizvietots ierastais modelis, kurā atsevišķi ir komandas koda rakstīšanai, testēšanai, izvietošanai un, visbeidzot, uzturēšanai [Mueller, 2016].

Otrkārt, DevOps var raksturot, kā programmatūras izstrādātāju izmantoto metodoloģiju un principu izmantošana IT infrastruktūras inženieru pienākumu izpildei. DevOps pieejas sevī ietver biznesu, klientus un pakalpojuma sniedzējus, programmatūras izstrādi un arhitektūru [Whickett, 2016].

Komandas, kas piekopj šo filozofiju, fokusējas uz izstrādes vižu standartizēšanu un programmatūras piegādes procesu automatizāciju, lai uzlabotu piegāžu kvalitāti, pārredzamību,

efektivitāti un drošību. DevOps metodoloģija izstrādātājiem piedāvā lielāku kontroli un saprašanu par produkcijas vidi. Šī kustība balstās uz trīs pamata principiem:

- Infrastruktūra kā kods;
- Nepārtrauktā piegāde;
- Sadarbības kultūra.

Infrastruktūra kā kods ir priekšnosacījums tādām DevOps pamatidejām, kā nepārtrauktā integrācija, versiju kontrole, automatizēta testēšana un koda pārskatīšana. Visas šīs metodes tiek pielietotas, lai izslēgtu cilvēku pieļautās kļūdas un ieviestu automatizāciju programmatūras izstrādes procesos. Automatizācijas ieviešanas iespējas un izmantojamo tehnoloģiju iedalījums redzams Tabula 2.3 [Kehrli, 2017].

Tabula 2.3

Infrastruktūras procesi, kuros iespējams ieviest automatizāciju izmantojot DevOps rīkus [darba autora tulkojums, Kehrli, 2017].

Infrastruktūras process	**Apakšprocesi**	**Rīki**
Programmatūras, servisu izvietošana un orķestrēšana	• Programmatūras koda izvietošana un atsaukšana • Resursu konfigurācija • Programmatūras un servisu startēšana	• Čaulas (shell), Python, Bash skripti • Ansible • Capistrano
Sistēmas instalācija, konfigurācija	• Servisu konfigurācija - žūrnāla (logs) faili, porti, lietotāju grupas utt. • Pievienošana uzraudzīšanai • JVM un programmatūru serveru uzstādīšana • Starpprogrammatūra (middleware)	• Chef • Puppet • Vagrant • Red Hat Satelite • CFEngine
Mašīnas uzstādīšana	• Virtualizācija • Pašapkalpošanās (self-service) vides • OS instalācija	• OpenStack • Docker • VMWare Vcloud • OpenNebula

3. DOCKER VIRTUALIZĀCIJAS PROTOTIPS

3.1. Problēmvides raksturojums

Docker virtualizācijas prototips tiks izstrādāts pieteikumu apstrādes sistēmai.

"Pieteikumu apstrāde, bieži saukta par kļūdu izsekošanu, ir programmatūras izstrādes izsekošanas process. Kļūdu izsekošana šajā gadījumā ir maldīgs, pārāk šaurs termins šim procesam. Pieteikumu apstrāde, savukārt, ir pietiekami plašs termins, lai raksturotu visus uzdevumus, kas saistīti ar programmatūras izstrādi [Henderson, 2006]".

Pieteikumu apstrādes sistēma uzņēmumos, kas saistīti ar tehnoloģiju izstrādi, ir tik pat ikdienišķs darbarīks, kā e-pasta klienta programma. Kā vienu no populārākajiem šāda veida rīkiem var minēt uzņēmuma Atlassian produktu Jira [Lindert-Buława , 2016].

Jira ir pieteikumu apstrādes sistēma, kas piedāvā pieteikumu apstrādi, projektu pārvaldību, kā arī darba plūsmu vadību. Mūsdienās šo sistēmu izmanto jau 75 000 kompāniju, lai uzlabotu programmatūras izstrādi. Sistēma ir pieejama 3 modifikācijās:

- Jira Software – projektu un pieteikumu pārvaldība;
- Jira Service Desk – klientu apkalpošanas serviss;
- Jira Core – biznesa vadība [Jira].

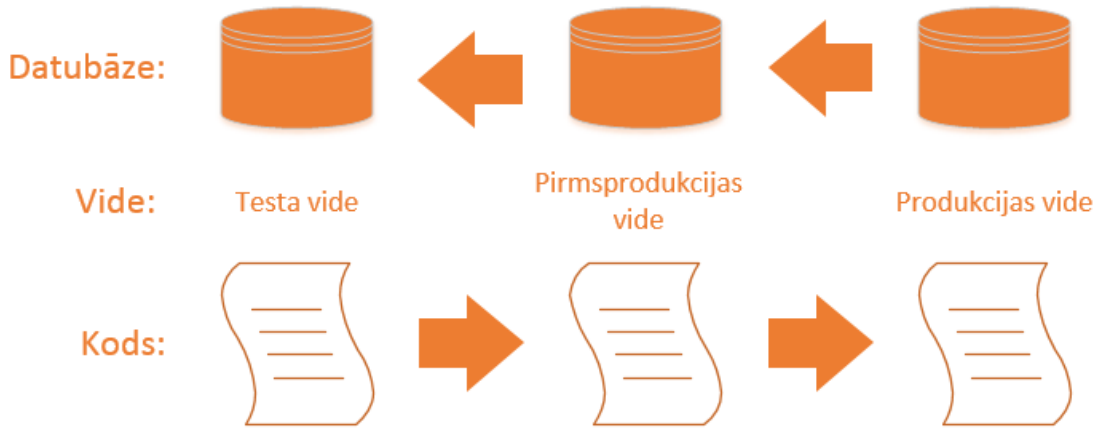
Tā kā Jira sistēmu mēdz lietot kompānijas ar darbinieku skaitu virs 1000, kā, piemēram, CITRIX, Tinder, Twitter, Audi, Latvenergo u.c. [Jira Customers], tad Atlassian pret sava produkta izstrādi un uzturēšanu izturas nopietni, regulāri izlaižot sistēmas atjauninājumus. Lai nodrošinātu kvalitatīvu sistēmas darbību un varētu izmantot papildus funkcionalitāti, ko piedāvā jaunākas Jira versijas, ir nepieciešams veikt sistēmas migrāciju uz jaunākām versijām.

Lielās kompānijās ar milzīgām Jira instancēm, kas satur daudz modifikāciju, spraudņus un integrācijas ar citām sistēmām, migrācijas darbi mēdz pāraugt pus gadu ilgos projektos, iesaistot lielu skaitu IT infrastruktūras un izstrādāju inženierus. Migrācijas projekts sastāv no tādiem pašiem etapiem, kā jebkurš cits programmatūras izstrādes projekts – prasību specifikācija, izstrāde, testēšana un ieviešana. Izstrādes, testēšana un ieviešanas etapiem attiecīgi ir nepieciešams nodrošināt vides, kur veikt izstrādi un testēšanu.

Izstrādes vide nodrošina izstrādātājus ar datortehnikas un programmatūras rīkiem jaunas produkta versijas jeb izstrādes būvējuma (developement build) izveidei. Izstrādes vide satur rīkus, kas nodrošina kodēšanu, vienību testus, regrestestus, konfigurāciju pārvaldību, saderības pārbaudes un versiju kontroli. Kad izstrādātājs ir beidzis darbu pie izstrādes būvējuma, tas tiek pārvietots uz testa vidi un kļūst par testa būvējumu (test build) [Cline, 2015, 108].

Testa vide nodrošina testētājus ar datortehnikas un programmatūras rīkiem testa scenāriju izpildei. Videi jānodrošina visu nepieciešamo automatizētai testēšanai: integrāciju testēšana, regrestestēšana un akceptestēšana, sistēmas veiktspējas testēšana, slodzes testēšana. Tapāt vide var saturēt defektu uzraudzīšanas un izziņošanas rīkus. Kad testa būvējums ir notestēts, tas tiek pārvietots uz pirmsprodukcijas (staging) vai produkcijas vidi (Attēls 3.1) [Testing Guidelines].

Lai nodrošinātu kvalitatīvu testa būvējumu jeb piegāžu testēšanu, videi jābūt pēc iespējas līdzīgai produkcijas videi. Būtu maz jēgas no piegāžu testiem, kas atšķiras no tā, kā tā tiks lietota produkcijā [Cline, 2015, 108]. Testa vides izveide un uzturēšana pēc iespējas līdzīgāka produkcijai prasa lielu cilvēkresursu ieguldījumu.



Attēls 3.1. Programmatūras izstrādes darbplūsma.

Tā kā daudz uzņēmumu savu programmatūras sistēmu uzturēšanai izmanto virtualizācijas tehnoloģijas, tad, piemēram, testēšanas vides izveide un sagatavošana Jira sistēmas migrācijas darbiem sastāv no sekojošiem posmiem:

1) Divu VM izveide;
2) Servera, uz kā atradīsies Jira sistēma, konfigurācija;
3) Datubāzes servera konfigurācija;
4) Datubāzes sistēmas uzstādīšana (Oracle 11g);
5) Jira sistēmas uzstādīšana;
6) Datu eksports no produkcijas vides;
7) Datu imports testa vidē.

Pēc testa vides izveides var uzsākt piegāžu testēšanu, kas sastāv no sekojošiem etapiem:

1) Piegādes failu saņemšana FTP serverī vai izmantojot kādu no versiju kontroles sistēmas repozitorijiem [G2 Crowd];
2) Piegādes uzstādīšana;
3) Jira sistēmas restartēšana, ja neieciešams;
4) Piegādes testēšana;
5) Problemātiskas piegādes gadījumā vides atgriešana iepriekšējā stāvoklī.

Visi augstāk minētie procesi testa vides izveidei un piegāžu testēšanai aizņem daudz iesaistīto cilvēku laiku, kuru būtu iespējams samazināt ieviešot automatizāciju. Piemēram, VM sagatavošana nozīmē izveidot jaunu VM, izmantojot iepriekš sagatavotu veidni (template), piešķirt serverim nosaukumu, piesaistīt domēna vārdu, uzstādīt nepieciešamo atkarības programmatūru (dependency software), veikt Jira vai Oracle 11g sistēmu instalāciju.

Attiecībā uz piegāžu testēšanas soļiem, ir nepieciešams daudz manuāla darba, lai veiktu tik vienkāršu darbību, kā jauna Jira spraudņa uzstādīšanu testēšana vajadzībām. Piemēram, IT inženieris saņem informāciju, ka uz FTP servera ir izvietots jauns testa būvējums. Darbinieks pieslēdzas FTP serverim, pārkopē saņemtos failus uz savu OS, pieslēdzas Jira sistēmai, caur administratora sadaļu augšupielādē spraudni, veic testēšanu.

Ja testa būvējums ir bijis nekvalitatīvs un sabojājis visas sistēmas darbību, tad testa vidi nepieciešams veidot pa jaunam atkal lieki patērējot IT infrastruktūras inženieru laiku. Šajā gadījumā noderētu versiju kontroles sistēma, kas ļautu saglabāt testa vides stāvokli pirms piegādes uzstādīšanas un testēšanas uzsākšanas.

3.2. Uzdevuma nostādne

Pielietojot Docker virtualizācijas tehnoloģijas piedāvātās iespējas, var atrisināt iepriekšējā apakšnodaļā aprakstītās problēmas. Procesi, kuru izpildes ātrumus iespējams uzlabot, ir:

1) Jaunas testa vides izveide – 4 stundas;
2) Piegādes uzstādīšana – 15 minūtes;
3) Vides atjaunošana, ja piegādes uzlikšanas rezultātā, tā vairs nav lietojuma – 2 stundas.

Patērētais laiks balstīts uz datiem, kas ņemti no Latvijā strādājoša uzņēmuma, kurā IT infrastruktūras procesi ir sadalīti pa struktūrvienībām. Piemēram, testa vides izveidē ir iesaistīts VMWare virtualizācijas eksperts, Linux OS inženieris, kā arī inženieris, kas veic Jira lietojumsistēmas instalāciju uz sagatavotā servera.

Izstrādājot virtualizācijas prototipu ir sagaidāmi sekojoši rezultāti:

1) Samazināt jaunas testa vides izveides laiku līdz mazākam par 10 minūtēm;
2) Piegādes uzlikšanas laiku samazināt līdz mazākam par 2 minūtēm;
3) Vides atjaunošanas laiku samazināt līdz mazākam par 2 minūtēm;
4) Ieviest automatizāciju iepriekšējos trīs procesos.

3.3. Risinājuma izstrāde

Prototips sastāv no 3 daļām. Vispirms tiek izveidota Jira produkcijas vide un sagatavoti testa dati. Otrajā daļā tiek veidota Jira testa vide, bet trešajā daļā Jira testa vides izveides procesam tiek ieviesta automatizācija.

Prototipa izstrādei ir sagatavota sekojoša 2 serveru ekosistēma, izmantojot VMWare virtualizācijas risinājumu:

1) Servera nosaukums: ftp-server:
   - Operētājsistēma: Centos Linux 7 (1611);
   - Atmiņa: 1024MB;
   - CPU: 4 * 2496Mhz;
   - Uzstādīts FTP serveris, kur atradīsies testa būvējumu piegādes, kā arī Jenkins automatizācijas rīks;
2) Servera nosaukums: jira-test:
   - Operētājsistēma: Centos Linux 7 (1611);
   - Atmiņa: 32GB;
   - CPU: 4 * 2496Mhz;
   - Produkcijas vides datu sagatavošana un Jira testa vides izveide.

3.3.1. Produkcijas vides izveide un testa datu sagatavošana

Lai izstrādātais prototips pēc iespējas vairāk līdzinātos procesiem reālajā pasaulē, tad vispirms nepieciešams izveidot Jira produkcijas vidi ar testa datiem un saglabāt datubāzes datu izrakstu (data dump).

Ar Linux termināļa komandrindām

```
sudo apt-get install docker-ce
sudo systemctl enable docker
```

jira-test serverī tiek uzstādīta Docker CE konteineru platforma un iestatīts, lai Docker dēmons startētos reizē ar operētājsistēmu.

Virtualizācijas prototipa izstrādē tiks izmantoti Docker Hub reģistrā publiski pieejami konteineru attēlu faili. Tā kā Jira programmatūrai tiks izmantota Oracle 11g datubāze (DB), reģistrā ir nepieciešams atrast piemērotu attēla failu. Balstoties uz aprakstu, lietotāju komentāriem un popularitāti tika izvēlēts *alexeiled/docker-oracle-xe-11g* repozitorijs, kas satur Oracle Express 11g R2 datubāzi un Ubuntu 14.04 LTS OS. Tālāk ar komandu

```
docker run --name oracledb -d --shm-size=1g -p 1521:1521 -p 8080:8080
alexeiled/docker-oracle-xe-11g
```

tiek startēts konteineris ar nosaukumu *oracledb*.

Kad Oracle DB ir iestartēta, nepieciešams uzģenerēt un palaist Jira sistēmu. Konteinera izveidei tiek izmantots *blacklabelops/jira* repozitorijs:

```
docker run -d -p 8081:8080 -v jiravolume:/var/atlassian/jira --name jira blacklabelops/jira
```

Pēc abu konteineru palaišanas, nepieciešams pārliecināties, ka tie veiksmīgi darbojas izmantojot Docker komandu *ps* (Attēls 3.2), kā arī atverot pārlūkprogrammā Oracle DB konfigurācijas paneli un Jira sākumlapu.

```
sh-4.2# docker ps
CONTAINER ID        IMAGE                            COMMAND                  CREATED
        STATUS              PORTS                                          NAMES
bcfcc42bd075        blacklabelops/jira               "/sbin/tini -- /us..."   9 seconds ag
o       Up 2 seconds        0.0.0.0:8081->8080/tcp                         jira
970a7d391d1b        alexeiled/docker-oracle-xe-11g   "/bin/sh -c /start.sh"   19 seconds a
go      Up 18 seconds       0.0.0.0:1521->1521/tcp, 0.0.0.0:8080->8080/tcp   oracledb
```

Attēls 3.2. Docker komanda, kas parāda startētos konteinerus.

Lai varētu saslēgt Jira sistēmu ar DB, vispirms nepieciešams pieslēgties Oracle 11g serverim un izveidot jaunu DB lietotāju. Šim procesam tiks izmantots *sflyr/sqlplus* repozitorijs, kurā ietilpst SQLPlus Client lietojumprogramma. Ar sekojošām komandrindām tiek lejupielādēts un palaists konteineris un izveidots savienojums ar Oracle DB, izveidots jauns lietotājs, piešķirtas nepieciešamās tiesības un izveidots katalogs, kas vēlāk tiks izmantots datu importam un eksportam:

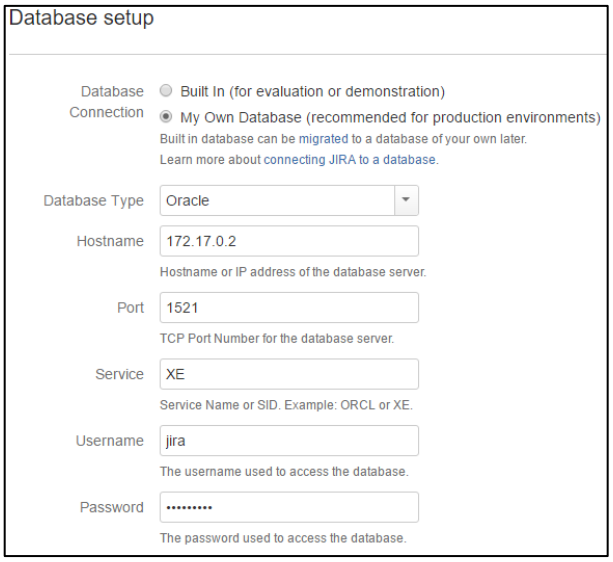
```
docker run -e URL=system/oracle@//172.17.0.2:1521/xe -ti sflyr/sqlplus
create user jira identified by Password1 default tablespace users quota unlimited on users;
grant connect to jira;
```

```
grant create table to jira;
grant create sequence to jira;
grant create trigger to jira;
create directory prod_exp_data AS '/u01/app/oracle/oradata/';
grant read, write on directory prod_exp_data to jira;
exit;
```

Abus izveidotos un nedaudz izmainītos konteinerus nepieciešams saglabāt, kā attēlu failus, vēlākai izmantošanai. Kā failu reģistrs tiks lietots publiski pieejamais Docker Hub, tāpēc nepieciešams veikt pieteikšanos reģistra DB, ievadot pieprasīto informāciju. Ar *commit* komandu tiek veikta attēlu failu saglabāšana.

```
docker login;
docker commit jira drinkits/jira:clean;
docker commit oracledb drinkits/oracle:usercreated;
docker push drinkits/jira:clean;
docker push drinkits/oracle:usercreated;
```

Kad attēlu faili saglabāti, nepieciešams atvērt Jira sistēmu izmantojot interneta pārlūkprogrammu un veikt sākuma parametru iestatīšanu. Kā DB tiek izvēlēta Oracle un aizpildīti visi ar to saistītie lauki (Attēls 3.3). Datubāzes IP adreses laukā tiek ievadīta Docker konteinera iekšējā adrese, kas tika noskaidrota ar *docker inspect* komandrindu.



Attēls 3.3. Datubāzes parametru iestatīšana Jira lietojumsistēmai.

Kad Jira sākotnējā konfigurācija pabeigta, var veikt testa datu izveidi. Izmantojot Jira spraudni Jira Data Generator, tika izveidota produkcijas vide ar sekojošiem datiem:

- 20 projekti;
- 32 pieteikumu tipi;
- 32 darbplūsmas;
- 100 000 pieteikumi;
- 699 056 komentāri;
- 3002 lietotāji;
- 107 pielāgotie lauki.

Produkcijas vides datubāzes datu eksportam var izmantot Oracle Data Pump utilītprogrammu. Ar sekojošām komandām tiek atvērta komandrindas čaula *oracldb* konteinerī un eksportēti dati failā *jira_prod.dmp, kā arī* datu izraksta faila un Jira datubāzes konfigurācijas faila pārkopēšana uz saimnieksistēmu.

```
docker exec -i -t oracledb /bin/bash;
expdp jira/Password1@localhost schemas=jira directory=prod_exp_data
dumpfile=jira_prod.dmp logfile=jira_prod.log;
exit;
docker cp oracledb:/u01/app/oracle/oradata/jira_prod.dmp /u01/;
mkdir /u01/dockerjira_home;
docker cp jira:/var/atlassian/jira/dbconfig.xml /u01/dockerjira_home/;
sudo chown -R jira:jira /u01/dockerjira_home;
chmod -R u=rwx,g=rwx,o=rwx /u01/dockerjira_home
```

Jira lietojumsistēma satur sākuma katalogu (home directory), kurā sistēma glabā datus, kas nosaka tās darbību: kešatmiņa, datubāzes savienojuma konfigurācijas fails, pieteikumu pielikumi un spraudņi. Šī kataloga "karkass" tiek izveidots, kad pirmoreiz tiek palaista sistēma un veikta tās konfigurācija. Tā kā šī katalogs atrodas virtualizētā Docker konteinera vidē, tad nepieciešams nodrošināt, lai datubāzes savienojum konfigurācijas fails netiku veidots pa jaunam pie katra jauna konteinera izveides. Lai to nodrošinātu, *dbconfig.xml* failu nepieciešams izvietot kādā no saimnieksistēmas katalogiem. Docker komanda *cp* ļauj brīvi kopēt failus no virtuālās vides jeb konteinera uz saimnieksistēmu un otrādi. Izmantojot šo komandu, no konteinera uz saimnieksistēmu tiek pārkopēts *dbconfig.xml* fails.

3.3.2. Testa vides izveide

Iepriekš veikto darbību rezultātā ir iegūti divu Docker attēlu faili, kas saglabāti reģistrā, kā arī 4,7 GB liels produkcijas vides datubāzes datu izraksts. Šie faili tiks izmantoti jaunas Jira testa vides izveidē, kas saturēs identiskus datus produkcijas videi. Pirms testa vides izveides vispirms nepieciešams atgriezt *jira-test* serveri sākuma stāvoklī un dzēst visu informāciju no Docker: konteinerus, attēlu failus, datu sējumus (volume).

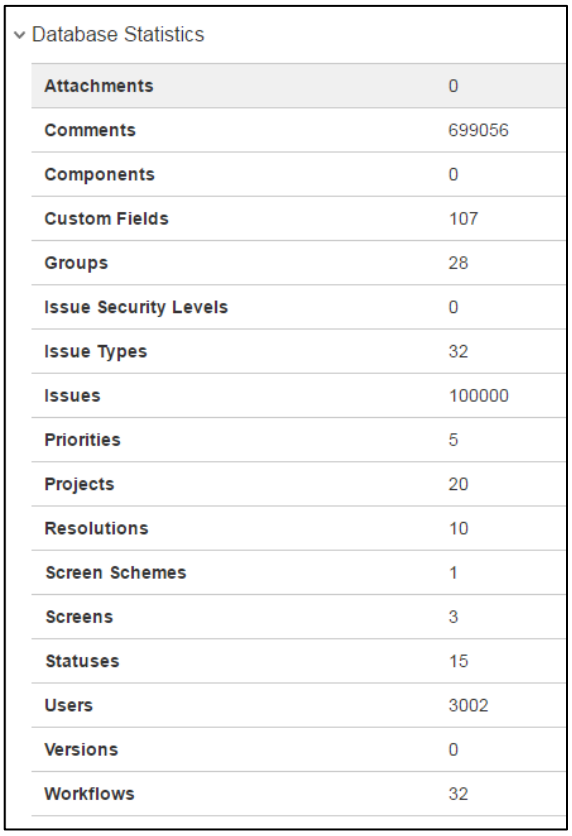
Kad Docker dati izdzēsti, no reģistra nepieciešams lejupielādēt attēlu failu un attiecīgi izveidot *oracledb* konteineru. Pēc konteinera palaišanas var tikt importēti produkcijas vides dati no faila *jira_prod.dmp:*

```
docker run --name oracledb -d --shm-size=1g -p 1521:1521 -p 8080:8080
drinkits/oracle:usercreated;
docker cp /u01/jira_prod.dmp oracledb:/u01/app/oracle/oradata/;
docker exec -i -t oracledb /bin/bash;
chmod 777 /u01/app/oracle/oradata/jira_prod.dmp;
impdp jira/Password1@localhost schemas=jira directory=prod_exp_data
dumpfile=jira_prod.dmp logfile=jira_prod.log;
exit;
```

Tā kā saimnieksistēmā atrodas Jira datubāzes savienojuma konfigurācijas fails, tad *jira* konteinera startēšanas laikā nepieciešams norādīt, ka virtuālās vides attiecīgā konfigurācijas faila vietā tiks izmantots fails no saimnieksistēmas. To iespējams panākt izmantojot Docker sējumu parametru *–v:*

```
docker run -d -p 8081:8080 -v
/u01/dockerjira_home/dbconfig.xml:/var/atlassian/jira/dbconfig.xml --name jira
drinkits/jira:clean;
```

Pēc konteinera palaišanas, Jira administrācijas sadaļā nepieciešams pārbaudīt, vai datu migrācijas no produkcijas vides ir noritējusi veiksmīgi. Kā redzams Attēls 3.4., dati sakrīt ar iepriekš produkcijas vidē sagatavotajiem testa datiem.



Attēls 3.4. Jira datubāzes statistikas dati.

3.3.3. Piegādes uzlikšana testa vidē

Prototipā izstrādē tiek pieņemts, ka testa būvējumu piegādēm tiks izmantots FTP serveris *ftp-server*. Izstrādātāji piegādes, kas arhivētas ZIP formātā, novieto FTP servera katalogā */files*. Kad saņemta informācija par jauna testa būvējumu, kas izvietots FTP serverī, vispirms nepieciešams pārkopēt norādīto arhīva failu uz *jira-test* serveri un izvilkt failus no tā.

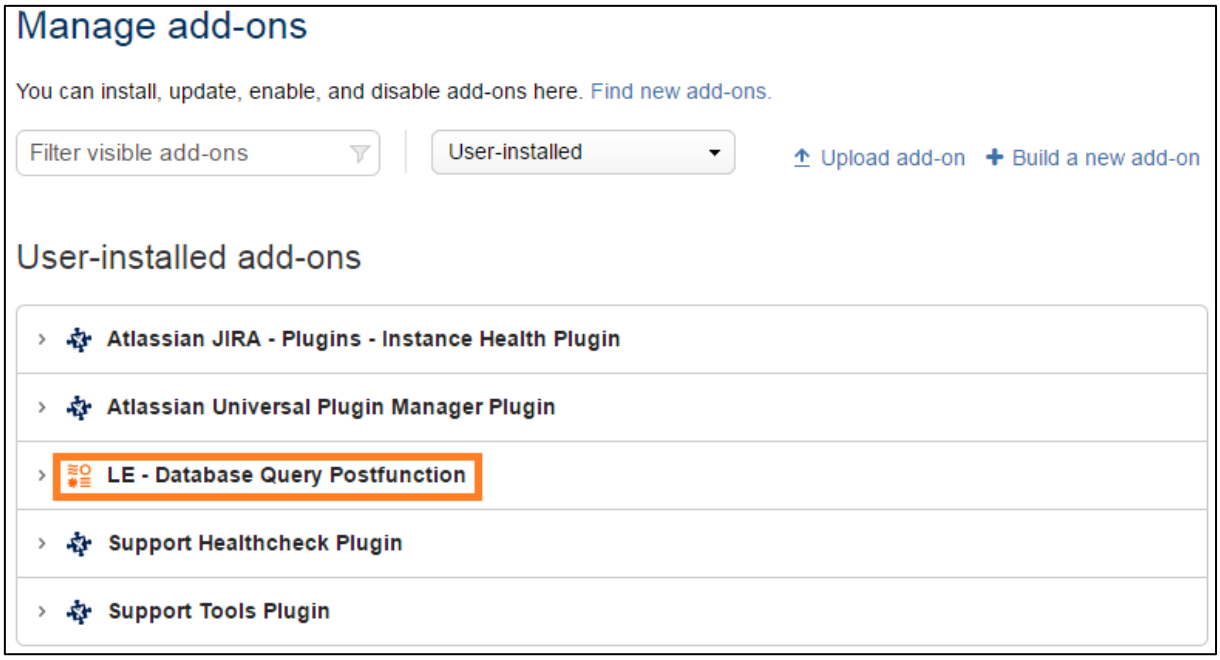
Arhīvs satur sistēmas spraudni ar datnes paplašinājumu *jar*. Prototipa izstrādē tika pieņemts, ka izstrādātājs piegadā 2. tipa Jira spraudņus, kuru instalāciju var veikt, iekopējot *jar* failus sistēmas sākuma katalogā. Pēc faila pārkopēšana nepieciešams veikt sistēmas restartu [Burwinkle, 2015].

Lai veiktu augstāk minētās darbības, nepieciešams terminālī izpildīt sekojošās komandrindas:

```
mkdir /u01/dockerjira_home/piegades;
wget ftp://jira:Password1@ftp-server/files/Pieg.7.005.zip -P
/u01/dockerjira_home/piegades/;
unzip /u01/dockerjira_home/piegades/Pieg.7.005.zip -d /u01/dockerjira_home/piegades/;
```

```
for line in $(find /u01/dockerjira_home/piegades/KVIK.7.005/ -iname '*.jar'); do
    docker cp "$line" jira:/var/atlassian/jira/plugins/installed-plugins
done;
docker restart jira;
```

Pēc sistēmas restarta interneta pārlūkprogrammā nepieciešams atvērt Jira spraudņu pārvaldības sadaļu un pārliecināties, ka instalācija noritējusi veiksmīgi (Attēls 3.5).



Attēls 3.5. Jira spraudņu pārvaldības sadaļa, kurā redzama uzstādītā piegāde.

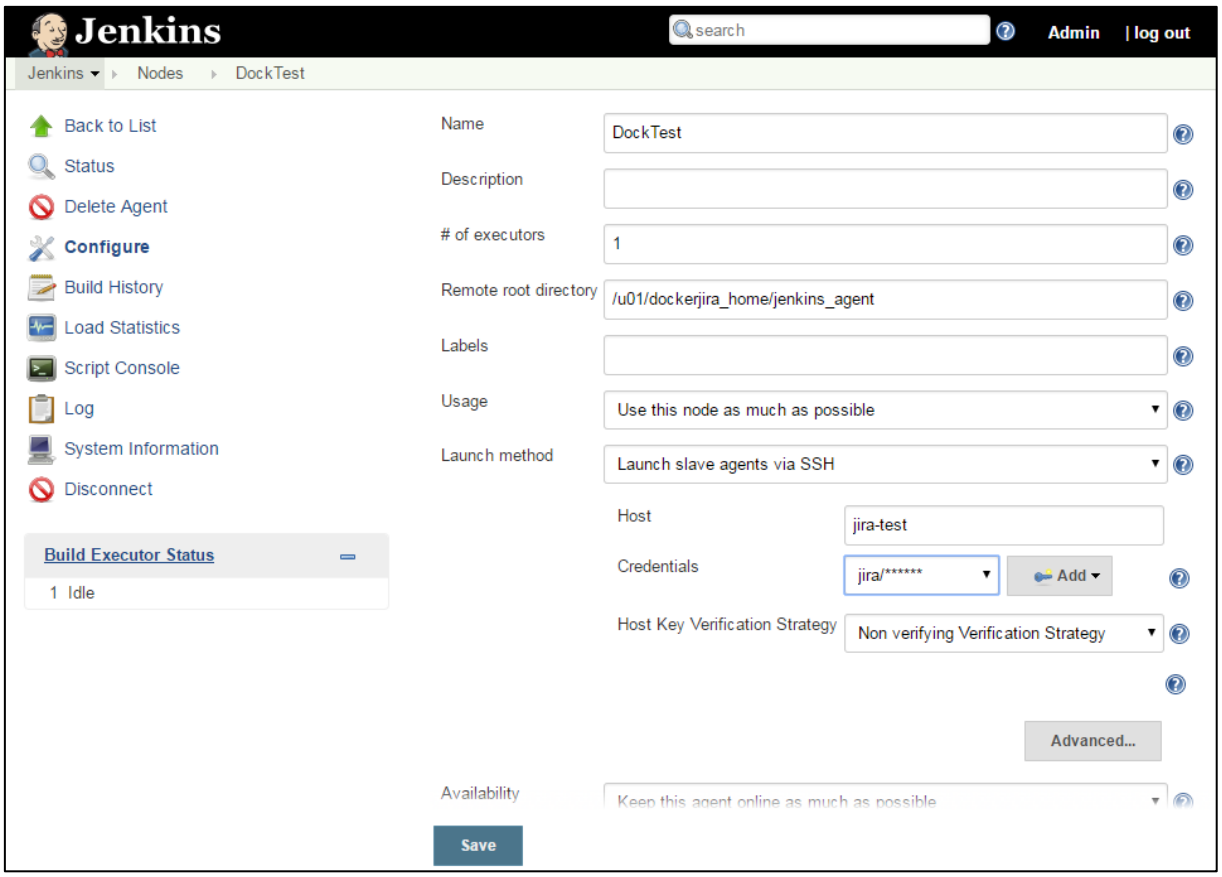
3.3.4. Automatizācijas izstrāde

Kā redzams iepriekšējās prototipa izstrādes daļās, gandrīz visi procesi, kas saistīti ar produkcijas, testa vižu izveidi un piegādes instalāciju, tiek veikti izmantojot tikai komandrindas. Jira lietojumsistēmas konfigurāciju, kura tika veikta izmantojot pārlūkprogrammu, vairs nav nepieciešams atkārtot, jo tā bija nepieciešama tikai testa datu sagatavošanai.

Komandrindu izmantošana infrastruktūras procesu izpildei nozīmē, ka iespējams realizēt DevOps principu – infrastruktūra kā kods un veikt automatizācijas izstrādi iepriekš aprakstītajām darbībām. Automatizācijas izveidei tiks izmantots atvērtā koda automatizācijas rīks Jenkins.

Jenkins ir produktivitātes veicinoša programmatūra, kas atbalsta nepārtrauktās integrācijas un piegādes modeļus. Šo rīku var izmantot, lai izstrādātu un testētu programmatūras izstrādes projektus pielietojot procesu automatizācijas iespējas [Kawaguchi, 2016].

Vispirms Jenkins automatizācijas serveris tiek uzstādīts *ftp-server* serverī. Kad rīks ir instalēts, nepieciešams veikt sākuma konfigurāciju izmantojot interneta pārlūkprogrammu. Sākuma konfigurācija pieprasa izveidot administratora kontu, kā arī uzstādīt spraudņus. Pēc konfigurācijas veikšanas, nepieciešamas pievienot *jira-test* serveri kā Jenkins automatizācijas izpildes mezglu (Attēls 3.6). Izpildes mezgls tiek startēts izmantojot SSH protokolu.

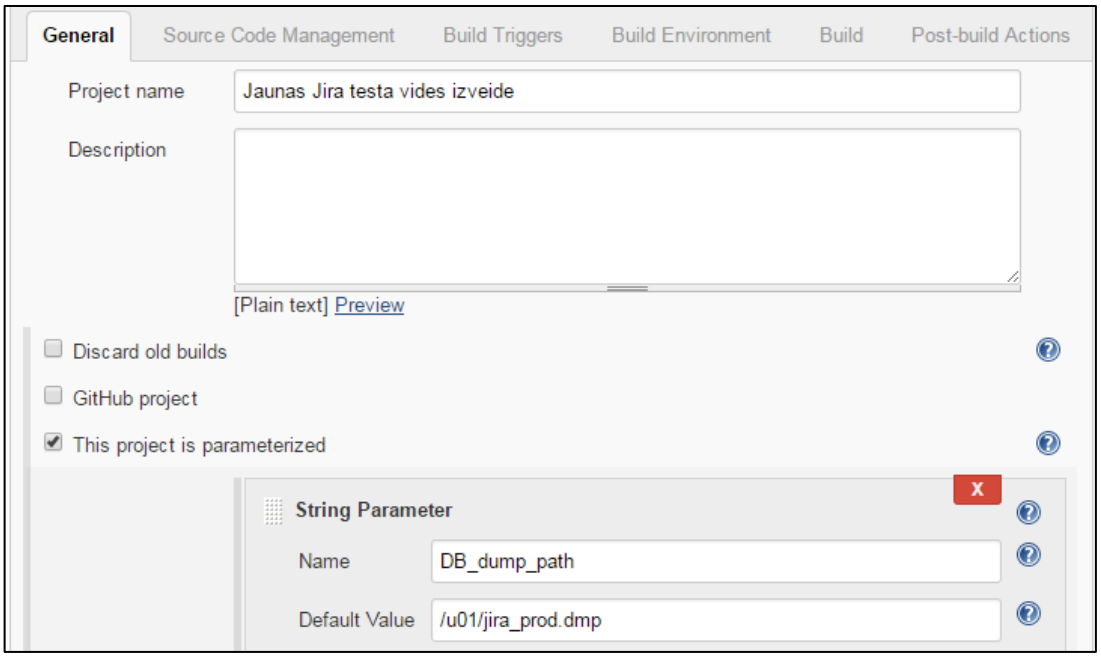


Attēls 3.6. Automatizācijas izpildes mezgla pievienošana Jenkins rīkā.

Pēc veiksmīgas izpildes mezgla pievienošanas, var sākt veidot Jenkins projektus, kas automatizēs sekojošus procesus:

- Jira testa vides izveide;
- Jira testa vides dzēšana;
- Piegādes uzlikšana;
- Testa vides atgriešana iepriekšējā stāvoklī.

Pirmais automatizācijas projekts tiek veidots testa vides sagatavošanai. Tā kā testa vide tiek veidota balstoties uz produkcijas vides datiem, tad projektam nepieciešams norādīt parametru, ar kuru varēs definēt produkcijas vides datubāzes datu izraksta faila atrašanās vietu *jira-test* serverī. Kā noklusētā parametra vērtība tiek ievadīta */u01/jira_prod.dmp (*Attēls 3.7*).*



Attēls 3.7. Projekta izveide un parametra definēšana Jenkins rīkā.

Pēc parametra iestatīšanas tiek norādīts, ka projekta izpildi var veikt tikai jira-test mezglā. Pati svarīgākā konfigurācijas daļa ir Build, kurā nepieciešams norādīt visus automatizācijas soļus. Tā kā iepriekš Jira testa vides izveides procesā tika izmantotas čaulas komandrindas, tad arī šeit katra darbības izpildei tiks lietotas Linux čaulas komandas. Projekta izpilde ir sadalīta divos soļos:

- *oracledb* konteinera izveide un datubāzes datu importēšana;
- *jira* konteinera izveide.

Abi soļi attiecīgi satur sekojošās komandrindas:

```
sudo docker run --name oracledb -d --shm-size=1g -p 1521:1521 -p 8080:8080 drinkits/oracle:usercreated;

sudo docker cp $DB_dump_path oracledb:/u01/app/oracle/oradata/;

sleep 60;

sudo docker exec -i oracledb /bin/bash -c "export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe && export
```

```
PATH=$ORACLE_HOME/bin:$PATH && export ORACLE_SID=XE &&
/u01/app/oracle/product/11.2.0/xe/bin/impdp jira/Password1@localhost schemas=jira
directory=prod_exp_data dumpfile=jira_prod.dmp logfile=jira_prod.log" || true;
```

```
sudo docker run -d -p 8081:8080 -v
/u01/dockerjira_home/dbconfig.xml:/var/atlassian/jira/dbconfig.xml --name jira
drinkits/jira:clean;
```

Kā redzams pirmā soļa komandrindas satur pārtraukuma komandu sleep 60, kas nepieciešams, lai noslēgtos Oracle DB sistēmas palaide pirms tiek veikts datu imports. Skripta beigās ir lietota Jenkins komanda true, kas ļauj ignorēt brīdinājumus, kas saņemti no datu importēšanas procesa.

Kad Jira testa vides izveides projekts sagatavots, nepieciešams sagatavot analogu Jenkins projektu, kas veiks testa vides dzēšanu. Šajā projektā tiek definēts tikai viens solis, kas satur sekojošās komandrindas:

```
sudo docker stop jira || true;
sudo docker rm -v jira || true;
sudo docker stop oracledb || true;
sudo docker rm -v oracledb || true;
```

Trešais Jenkins projekts tiek veidots testa būvējumu instalācijas procesam. Lai šīs procesa automatizācijas lietotājs varētu noradīt, kuru piegādi nepieciešams instalēt, projektā tiek definēts parametrs *Build_name.* Komandrinda tiek definēta tikai viena solim:

```
sudo docker commit jira drinkits/jira:stable;
wget ftp://jira:Password1@ftp-server/files/$Build_name.zip -P
/u01/dockerjira_home/piegades/;
unzip /u01/dockerjira_home/piegades/$Build_name.zip -d
/u01/dockerjira_home/piegades/;
for line in $(find /u01/dockerjira_home/piegades/$Build_name/ -iname '*.jar'); do
    sudo docker cp "$line" jira:/var/atlassian/jira/plugins/installed-plugins;
done;
rm -rf /u01/dockerjira_home/piegades/$Build_name
rm -f /u01/dockerjira_home/piegades/$Build_name.zip
sudo docker restart jira;
```

Komandrindas sākumā tiek veikta konteinera momentuzņēmuma saglabāšana, kas nepieciešama gadījumā, ja piegādes rezultātā ir “sabojāta” Jira testa vide.

Lai būtu iespēja atgriezt testa vidi uz iepriekšējo stāvokli izmantojot iepriekš izveidoto konteinera momentuzņēmumu, nepieciešams izveidot vēl ceturto projektu ar sekojošām komandrindām:

```
sudo docker stop jira || true;
sudo docker rm -v jira || true;
sudo docker run -d -p 8081:8080 -v
/u01/dockerjira_home/dbconfig.xml:/var/atlassian/jira/dbconfig.xml --name jira
drinkits/jira:stable;
```

Jenkins sākumlapā tagad ir redzami visi četri projekti, kas ir piesaistīti jira-test automatizācijas izpildes mezglam (Attēls 3.8).

Attēls 3.8. Jenkins automatizācijas projekti.

3.4. Rezultāti

Prototipa izstrādes rezultātā ir iegūts risinājums, kas iepriekš vairāku inženieru veikto laikietilpīgo darbu ir minimizējis līdz četriem automatizācijas projektiem, kuru startēšana ir "viena klikšķa attālumā". To bija iespējams realizēt pateicoties Docker klienta spējai kontrolēt dēmonu ar komandrindu palīdzību: konteineru izveide, startēšana, failu kopēšana starp izpildes vidēm, kā arī momentuzņēmuma saglabāšana. Lai varētu noskaidrot, vai prototips ir atrisinājis uzdevuma nostādnē definētās problēmas, nepieciešams veikt laika mērījumus. Vispirms tiek attīrīts Docker virtualizācijas rīks no visas informācijas, kas izveidota un saglabāta prototipa izstrādes gaitā.

Pirmajā mērījumā tiks pārbaudīts, cik ilgs laiks nepieciešams, lai izveidotu jaunu Jira testa vidi ar datiem, kas identiski produkcijas videi. Vispirms tiek pārbaudīts, vai eksistē 4,7 GB lielais produkcijas vides datu izraksta fails */u01/jira_prod.dmp*. Pēc pārbaudes, Jenkins rīkā tiek startēts projekts ar nosaukumu *Izveidot Jira testa vidi*. Tā kā datubāzes eksporta faila

nosaukums un atrašanās vietas nav mainīta, tad *DB_dump_path* laukā tiek atstāta noklusētā vērtībā. Pēc *Build* pogas nospiešanas, tiek uzsākta testa vides izveide. Pēc precīzi 10 minūtēm Jenkins programmatūrā tiek saņemts ziņojums par veiksmīgi noslēgušos automatizācijas procesu. Lai pārliecinātos, ka automatizācijas izpilde noritējusi veiksmīgi, tiek veiktas sekojošās darbības:

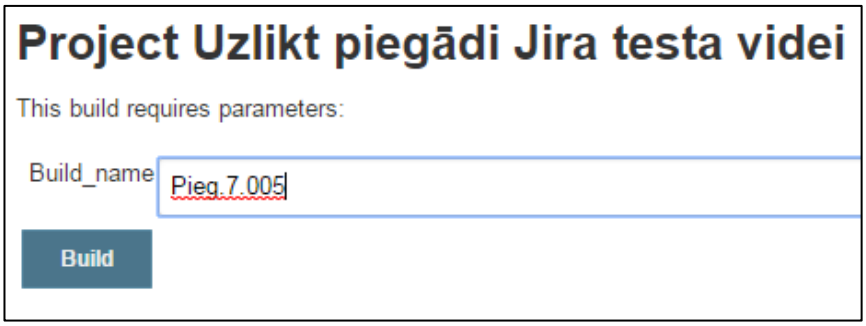
- Pārbaudīts konsoles saturs, kas saņemts no *jira-test* mezgla;
- Interneta pārlūkprogrammā atvērta Jira lietojumsistēma;
- Jira administrācijas sadaļā pārbaudīts, vai importētie dati sakrīt ar datiem kas bija izveidoti produkcijas vidē.

Attēls 3.9. Automatizācijas palaišanas logs Jira testa vides izveidei.

Tā kā pirmajā mērījumā Docker attēlu faili tika lejupielādēti no Docker Hub reģistra, tiek veikts vēl viens mērījums ar lokāli eksistējošiem attēlu failiem. Otrajā mērījumā jaunas vides izveide aizņēma 5 minūtes un 3 sekundes.

Kad iegūti pirmie mērījumi, var palaist testa vides dzēšanas projektu *Dzēst Jira testa vidi*. Automatizācijas process noslēdzās jau pēc 25 sekundēm. Ar Docker komandu *ps* tiek pārbaudīts, vai vide veiksmīgi likvidēta.

Nākošais mērījums tiek veikts *Uzlikt piegādi Jira testa videi* projektam. Projekts tiek startēts, ievadot parametru, kas norāda piegādes nosaukumu (Attēls 3.10). Jenkins rīks jau pēc 14 sekundēm ziņo, ka testa būvējums ir instalēts Jira lietojumsistēmā. Automatizācijas izpildes rezultāts tiek pārbaudīts atverot Jira spraudņu pārvaldības sadaļu un konstatējot spraudņa klātesamību sarakstā.



Attēls 3.10. Automatizācijas palaišanas logs piegādes instalācijai.

Pēdējā mērījumā tiek pārbaudīts, cik ilgā laikā ir iespējams atjaunot testa vidi uz stāvokli, kāds tas bija pirms piegādes uzstādīšanas. Docker virtualizācijas tehnoloģijas ātrdarbība parādījās arī šajā automatizācijas projektā, kura izpilde prasīja 11 sekundes. Rezultāts tiek pārbaudīts konstatējot piegādes neesamību Jira lietojumsistēmā.

Iegūtie mērījumu rezultāti apstiprina uzdevuma nostādnē izvirzīto hipotēzi, ka Docker tehnoloģija uzlabos IT infrastruktūras procesu izpildes ātrumu, kā arī dos iespēju izstrādāt šo procesu automatizāciju. Prognozētie un reāli iegūtie procesu izpildes laiki atspoguļoti Tabula 3.1.

Tabula 3.1

Izstrādātā prototipa prognozētie un reālie izpildes laiki.

Process	**Prognozētais laiks**	**Reālais laiks**
Testa vides izveide	10 minūtes	• 10 minūtes[1] • 5 minūtes un 3 sekundes[2]
Piegādes uzstādīšana testa videi	2 minūtes	14 sekundes
Vides atjaunošana stāvoklī, kas fiksēts pirms piegādes uzstādīšanas	2 minūtes	11 sekundes

[1] Izmantojot Docker Hub reģistru.
[2] Izmantojot lokālos attēlu failus.

Salīdzinot iegūtos rezultātos ar IT inženieru patērēto laiku IT infrastruktūras procesu izpildei, uzlabojumi ir milzīgi. Piemēram, testa vides izveides laiks, izmantojot Docker

tehnoloģiju, ir par vairāk kā 24 reizēm mazāks, bet piegādes uzstādīšanas laiks - 64 reizes mazāks (Tabula 3.2).

Tabula 3.2

IT infrastruktūras procesu izpildes ātrumi.

Process	**IT inženieru darbs**	**Docker un procesu automatizācija**
Testa vides izveide	4 stundas	• 10 minūtes[1] • 5 minūtes un 3 sekundes[2]
Piegādes uzstādīšana testa videi	15 minūtes	14 sekundes
Vides atjaunošana stāvoklī, kas fiksēts pirms piegādes uzstādīšanas	2 stundas	11 sekundes

[1] Izmantojot Docker Hub reģistru.
[2] Izmantojot lokālos attēlu failus.

Kā otru lielāku prototipa ieguvumu var minēt izstrādāto automatizāciju, kas pilnībā izslēdz cilvēku iesaisti IT infrastruktūras procesu izpildei. Tas nozīmē, ka nav nepieciešams datubāžu speciālists Oracle DB uzstādīšanai, IT inženieri – VM mašīnu sagatavošanai un Jira lietojumsistēmas instalācijai.

Prototipa izstrādes gaitā tika konstatēts tikai viens Docker tehnoloģijas mīnuss, kas ir administratora tiesību nepieciešamība, lai būtu iespējams kontrolēt Docker dēmonu. To gan šī prototipa izstrādes kontekstā nevar pilnībā uzskatīt par tehnoloģijas problēmu, jo arī manuāli veicot aprakstītos procesus, būtu nepieciešamas administratora tiesības.

Tā kā prototipa izstrādē galvenais uzsvars tika likts uz ātrdarbību, tad netika pievērsta uzmanība drošības riskiem, kas saistīti ar publiski pieejamu attēlu failu izmantošanu konteineru izveidē. Tos varētu novērst veidojot attēla failus pašrocīgi.

NOBEIGUMS

Darba mērķis bija izpētīt Docker virtualizācijas tehnoloģiju iespējas programmatūras ražošanas procesa automatizācijai un izstrādāt prototipu, kas automatizē noteiktu programmatūras ražošanas procesa daļu. Lai to panāktu, bija nepieciešams apskatīt IT industrijā izmantotas virtualizācijas tehnoloģijas un padziļināti izpētīt Docker konteinerus. Darbā gaitā tika izstrādāts Docker virtualizācijas prototips programmatūras ražošanas procesa automatizācijai. Darba rezultātā tika notestēta prototipa darbība, kā arī novērtēti tā ieguvumi un nepilnības.

Izstrādātā Docker virtualizācijas un IT infrastruktūras procesu automatizācijas prototipa rezultāti tika salīdzināti ar šo pašu procesu izpildi reālajā dzīvē. Kā divus lielākos ieguvums var minēt izpildes ātrumu un cilvēka darba minimizēšanu līdz "vienas pogas piespiešanai.

Pēc Docker tehnoloģijas analīzes un veiktā pētījuma rezultātiem autors var secināt:

1. Docker tehnoloģijas veiktspēja ļauj krietni uzlabot programmatūras izstrādes procesa izpildes laiku;
2. Docker tehnoloģija ļauj ērti vadīties pēc DevOps kustības principiem: *infrastruktūras kā kods, nulles laika izvietošana, automatizēt visu;*
3. Docker komandrindas rīks nodrošina vienkāršu automatizācijas izstrādi;
4. Docker komandrindas rīks ir ērts un ātri apgūstams;
5. Sakarā ar lielo Docker tehnoloģijas popularitāti, ir pieejams milzīgs informācijas resursu, repozitoriju, attēlu failu un reģistru daudzums.

Lai izstrādāto prototipu varētu pielietot jau reālos programmatūras izstrādes projektos, nākotnē būtu nepieciešams uzlabot sekojošo:

1. Docker konteineru pielāgošanai izmantot *Dockerfile* instrukciju faila sniegtās iespējas [Dockerfile];
2. Samazināt Docker attēlu failu izmēru, izvācot visu nevajadzīgo no tiem;
3. Tā kā testa vide balstās uz diviem konteineriem, tad būtu lietderīgi izmantot *Docker Compose* rīka iespējas [Docker Compose];
4. Izpētīt Docker API izmantošanas iespējas prototipa izstrādes procesam;
5. Testa būvējumu saņemšanai izmantot versiju kontroles sistēmu;
6. Automatizācijas izstrādei Jenkins rīkā izmantot spraudņus, kas paredzēti manipulācijām ar Docker tehnoloģiju;
7. Automatizācijas rīkā paredzēt, likvidēt vai piedāvāt apkārtceļus kļūdām, kas var rasties procesu izpildes brīžos;

8. Automatizācijas rīkā nodrošināt iespēju veidot neierobežotu skaitu testa vižu.

PATEICĪBAS

Sievai, Kristīnei Rozenbergai, par milzīgo izturību, atbalstu un motivāciju darba izstrādes gaitā.

Rolandam Levicam par atbalstu un sapratni.

LITERATŪRA

[Anderson, 2015] Anderson C. *Docker*. IEEE Software. 2015, vol.32., pp.102. 0740-7459. Pieejams: doi: 10.1109/MS.2015.62

[Arijs, 2016] Arijs P. (2016). *Docker usage statistics: Increased adoption by enterprises and for production use* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://www.coscale.com/blog/docker-usage-statistics-increased-adoption-by-enterprises-and-for-production-use

[Ashraf, 2016] Ashraf W. (2016). *The Docker EcoSystem* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://washraf.gitbooks.io/the-docker-ecosystem

[Barbier, 2014] Barbier J. (2014). *Docker in Ubuntu, Ubuntu in Docker* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://blog.docker.com/2014/04/docker-in-ubuntu-ubuntu-in-docker/

[Burwinkle, 2015] Burwinkle C. (2015). Installing add-ons [tiešsaiste]. [skatīts 25.05.2017.]. Pieejams: https://confluence.atlassian.com/display/UPM/Installing+add-ons

[Chroot, 2017] *Containerization Mechanisms: Namespaces* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://blog.selectel.com/containerization-mechanisms-namespaces/

[Chroot Jails, 2016] *Configuring and Using Chroot Jails* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://docs.oracle.com/cd/E37670_01/E36387/html/ol_cj_sec.html

[Cline, 2015, 108] Cline A. (2015). *Agile Development in the Real World*. Apress. 297 p.

[Container, 2015] *Container vs. VM: What's the difference?* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: http://searchservervirtualization.techtarget.com/answer/Containers-vs-VMs-Whats-the-difference

[Daemon] *dockerd* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/reference/commandline/dockerd/

[Datalog, 2017] *Docker adoption* [tiešsaiste]. [skatīts 11.05.2017.]. Pieejams: https://www.datadoghq.com/docker-adoption/

[Docker, 2017] *What is Docker* [tiešsaiste]. [skatīts 11.05.2017.]. Pieejams: https://www.docker.com/what-docker

[DockerCon] *DockerCon* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://www.docker.com/events/dockercon

[Dockerfile] *Dockerfile reference* [tiešsaiste]. [skatīts 26.05.2017.]. Pieejams: https://docs.docker.com/engine/reference/builder/

[Docker Architecture] *Docker overview* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/docker-overview/

[Docker API] *Docker Engine API and SDKs* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/api/

[Docker Compose] *Docker Compose* [tiešsaiste]. [skatīts 26.05.2017.]. Pieejams: https://docs.docker.com/compose/

[Docker Containers] *Understand images, containers, and storage drivers* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#copying-makes-containers-efficient

[Docker Hub] *Overview of Docker Hub* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://docs.docker.com/docker-hub/

[Docker Networking] *Docker container networking* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/userguide/networking/

[Docker Registry] *Docker Registry* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://docs.docker.com/registry/

[Docker Security] *Docker security* [tiešsaiste]. [skatīts 18.05.2017.]. Pieejams: https://docs.docker.com/engine/security/security/

[Felter, 2014] Felter W i.a. (2014). *An Updated Performance Comparison of Virtual Machines and Linux Containers. IEEE Software*. 2014. Pieejams: doi: 10.1109/ISPASS.2015.7095802

[Golub, 2014] Golub B. (2014). Docker closes $15 M series B funding [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://blog.docker.com/2014/01/docker-closes-15-m-series-b-funding/

[G2 Crowd] *Best Version Control Systems* [tiešsaiste]. [skatīts 25.05.2017.]. Pieejams: https://www.g2crowd.com/categories/version-control-systems

[Henderson, 2016] Henderson C (2006*). Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications*. Sebastool, CA: O'Reilly Media.

[Hykes, 2013] Metz C. (2013). *The Man Who Would Build a Computer the Size of the Entire Internet* [tiešsaiste]. [skatīts 10.05.2017.]. Pieejams: https://www.wired.com/2013/09/docker/

[Hyper-V, 2017] *Virtualization in Windows Server 2016* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://docs.microsoft.com/en-us/windows-server/virtualization/virtualization

[Johnston, 2014] Johnston S. (2014). *Docker 1.3: signed images, process injection, security options, Mac shared directories* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/

[Jira] *JIRA Software - Issue & Project Tracking for Software Teams* [tiešsaiste]. [skatīts 22.05.2017.]. Pieejams: https://www.atlassian.com/software/jira

[Jira Customers] *Jira Customers* [tiešsaiste]. [skatīts 22.05.2017.]. Pieejams: https://www.atlassian.com/customers

[Kawaguchi, 2016] Kawaguchi K. (2016). Meet Jenkins [tiešsaiste]. [skatīts 25.05.2017.]. Pieejams: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

[Kehrli, 2017] Kehrli J. (2017). *DevOps explained* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://www.niceideas.ch/roller2/badtrash/entry/devops-explained

[Kerrisk, 2013] Kerrisk M. (2013). *Namespaces in operation, part 1: namespaces overview* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://lwn.net/Articles/531114/

[Lindert-Buława, 2016] Lindert-Buława K. (2016). *JIRA Software – #1 among the software development tools for Agile teams* [tiešsaiste]. [skatīts 22.05.2017.]. Pieejams: https://www.intenso-group.com/en/jira-software-1-among-the-software-development-tools-for-agile-teams/

[Luzzardi, 2015] Luzzardi A. (2015). *Announcing Swarm 1.0: Production-ready clustering at any scale* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://blog.docker.com/2015/11/swarm-1-0/

[Martin, 2015] Martin N. (2015). A brief history of Docker Containers' overnight success [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success

[Matthias, Kane, 2015] Matthias K, Kane S.P. (2015). *Docker: Up & Running: Shipping Reliable Containers in Production*. USA: O'Reilly Media, Inc.

[Menage, 2004] Menage P. (2004). *CGROUPS* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[Microsoft 2014] Microsoft News Center (2014). *Docker and Microsoft partner to bring container applications across platforms* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://news.microsoft.com/2014/10/15/DockerPR/

[Minas L., Ellison B., 2009] Minas L., Ellison B. (2009). *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. USA: Intel Press. 330 p.

[Mueller, 2016] Mueller E. (2016). *What Is DevOps?* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://theagileadmin.com/what-is-devops/

[Petazzoni, 2013] Petazzoni J. (2013). *Gathering LXC Docker containers metrics.* [tiešsaiste]. [skatīts 09.05.2017.]. Pieejams: https://blog.docker.com/2013/10/gathering-lxc-docker-containers-metrics/

[Pethuru u.c., 2016] Pethuru R. i.a. (2016). *Docker: Creating Structured Containers.* Packt Publishing Ltd.

[Portnoy, 2012, 21] Portnoy M. (2012). *Virtualization Essentials.* John Wiley & Sons. 286 p.

[Preeth u.c., 2015] Preeth i.a. (2015). *Evaluation of Docker Containers Based on Hardware Utilization. ICCC*. 2015. Pieejams: doi: 10.1109/ICCC.2015.7432984

[Solomon, 2014] Solomon H. (2014). *Docker 0.9: introducing execution drivers and libcontainer* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

[Singh, 2004] *An Introduction to Virtualization* [tiešsaiste]. [skatīts 06.05.2017.]. Pieejams: http://www.kernelthread.com/publications/virtualization/

[Slant, 2017] *What are the best Docker orchestration tools?* [tiešsaiste]. [skatīts 17.05.2017.]. Pieejams: https://www.slant.co/topics/3929/~docker-orchestration-tools

[Spiceworks, 2016] *2016 Latest Trends in IT & Technology Adoption* [tiešsaiste]. [skatīts 14.04.2017.]. Pieejams: https://www.spiceworks.com/marketing/resources/reports/2016-state-of-it/

[Suleman] *8 Proven Real-World Ways to Use Docker* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker

[Termini] *Akadēmiskā terminu datubāze – AkadTerm* [tiešsaiste]. [skatīts 14.04.2017.]. Pieejams: http://termini.lza.lv

[Testing Guidelines] *Pre-Production Testing: Best Practices & Guidance* [tiešsaiste]. [skatīts 22.05.2017.]. Pieejams: http://www.rgoarchitects.com/Files/TestingGuidelines.pdf

[Tschudin, 2012] Tschudin P.S. (2012). *Performance Overhead and Comparative Performance of 4 Virtualization Solutions.* [tiešsaiste]. [skatīts 09.05.2017.]. Pieejams: http://petersenna.com/files/peters-top4-virtualization-benchmark-1.29.pdf

[VirtualBox, 2017] *VirtualBox User Manual* [tiešsaiste]. [skatīts 06.05.2017.]. Pieejams: https://www.virtualbox.org/manual/

[VMware, 2017] *Workstation for Windows* [tiešsaiste]. [skatīts 06.05.2017.]. Pieejams: http://www.vmware.com/products/workstation.html

[Wachowicz, 2016] Wachowicz P. (2016). *Containerization vs. Virtualization – Image size and Overhead* [tiešsaiste]. [skatīts 09.05.2017.]. Pieejams: http://www.brightcomputing.com/blog/containerization-vs.-virtualization-heres-our-blog-smackdown

[Whickett, 2016] Whickett J. (2016). *What Is DevOps?* [tiešsaiste]. [skatīts 19.05.2017.]. Pieejams: https://www.lynda.com/Operating-Systems-tutorials/What-DevOps/508618/555077-4.html

[Yemelianov, 2017] Yemelianov A. (2017). *Containerization Mechanisms: Namespaces* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://blog.selectel.com/containerization-mechanisms-namespaces/

[Yemelianov, Cgroups, 2017] Yemelianov A. (2017). *Containerization Mechanisms: Cgroups* [tiešsaiste]. [skatīts 07.05.2017.]. Pieejams: https://blog.selectel.com/containerization-mechanisms-cgroups/