LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

BRĪVU MĀKOŅRESURSU PIEEJAMĪBA DATU ANALĪZEI

BAKALAURA DARBS

Autors: Ralfs Cimermanis
Studenta apliecības Nr.: rc17006
Darba vadītājs: prof., Dr.sc.comp. Jānis Zuters

RĪGA 2021

ANOTĀCIJA

Mākoņresursi ir ļoti izplatīti un paliek ar vien populārāki un uztur ļoti lielu daļu internetā pieejamo resursu, kā rezultāta tiek veidotas lielas infrastruktūras šīs informācijas nodrošināšanai. Liela daļa resursu no šīm infrastruktūrām netiek pilnībā izmantoti. Tas liecina, ka ir daudz brīvi pieejamu resursu, kurus var potenciāli izmantot datu analīzei. Tādēļ bakalaura darba izstrādes gaitā tiks meklēts un izstrādāts risinājums, ar kuru palīdzību veicot liela apjoma datu apstrādi paralēli citiem procesiem, tie neietekmē galveno procesu veiktspēju. Risinājums tiek balstīts uz to, ka brīvu mākoņresursu pieejamais daudzums ir aptuveni 70% un tie ir potenciāli izmantojami datu analīzei, veidojot risinājumu uz VMware vSphere bāzes, kopā ar Kubernetes un Docker konteineriem, veiksmīgi izmantojot līdz 17% CPU jaudas nepārsniedzot 5% CPU gaidīšanas laika.

Atslēgvārdi: Mākoņresursi, VMware vSphere, Docker, Kubernetes, brīvie mākoņu resursi, datu analīze

ABSTRACT

Availability of free cloud resources for data analysis

Cloud resources are very common and they are becoming more and more popular, and they maintain a very large share of the resources available on the Internet, resulting in the creation of large infrastructures that can provide and store the information found on Internet. Much of the resources from these infrastructures are underused. This suggests that there are many freely available resources that can potentially be used for data analysis. Therfore, during the development of the bachelor's thesis, a solution will be sought and developed, that will help with launching large-scale data processing processes in parallel with other processes. Solution is based on that, that available unused cloud resourses are around 70% and that they are potentially usable for data analytics, and it's made so that data analytics do not affect the performance of the main key processes. This solution was created based on VMware vSphere, together with Kubernetes and Docker containers, to successfully use 20% additional CPU resources without reaching CPU readiness above 5%.

Keywords: Cloud Resources, VMware vSphere, Docker, Kubernetes, Free Cloud Resources, Data Analysis.

SATURS

VĀRDNĪCA

1.tabula

Vārdnīcas termini un skaidrojumi

Jēdziens	**Skaidrojums**
Programmatūra kā pakalpojums (SaaS)	"Software as a Service" ir programmatūras izplatīšanas modelis, kurā trešās puses pakalpojumu sniedzēs mitina lietojumprogrammas un padara tās pieejamas klientiem internetā. [9, 10]
Platforma kā pakalpojums (PaaS)	"Platforma s a Service" ir mākoņdatošanas modelis, kurā trešās puses pakalpojumu sniedzējs piegādā aparatūras un programmatūras rīkus lietotājiem internetā. Parasti šie rīki ir nepieciešami lietojumprogrammu izstrādei. [9, 10]
Integrācija kā pakalpojums (IaaS)	"Integration as a Service" ir mākoņa bāzes piegādes modelis, kas cenšas savienot vietējos datus ar datiem, kas atrodas mākoņa lietojumprogrammās. [9, 10]
NIST	ASV Nacionālo standartu un tehnoloģiju institūts
Ops	Operācijas sekundē
Kodola / CPU gatavība	Gatavība ir laiks, kad virtuālā mašīna bija gatava, taču nevarēja ieplānot darboties ar fizisko procesoru.
API (Lietojumprogrammas saskarne)	Saskarne caur kuru programmas var komunicēt ar citām programmatūrām.
VM	Virtuālā mašīna

IEVADS

Mākoņpakalpojumi ir izplatīta informāciju tehnoloģiju komponente, ar kuras palīdzību tiek uzturēta liela daļa internetā pieejamo resursu. Viens no iemesliem ir šo resursu izdevīgā cena uzņēmumiem pār privātiem datu centriem, kā arī to elastība jaunu sistēmu uzstādīšanā.

Uzņēmumiem un privātpersonām pārejot pie mākoņpakalpojumu sniedzējiem ir augstas ekspektācijas par to kvalitāti un nepārtrauktu pakalpojuma sniegšanu. Lai šīs lietas pakalpojumu sniedzēji varētu nodrošināt, tiek būvētās lielas infrastruktūras, ar pietekami daudz resursiem, lai apmierinātu gan lielu uzņēmumu vajadzības gan privātpersonu mazākas vajadzības.

Dēļ tā, ka mākoņu infrastruktūras tiek būvētas ar domu par nākotni, tās vidēji patērē tikai 25-30% pieejamās procesēšanas jaudas, kā rezultātā rodas jautājums, kā šos brīvos resursu varētu izmantot dažādām datu analīzes aplikācijām neietekmējot citas sistēmas.

Darba izveide tiek balstīta uz kursa darba "Brīvu mākoņresursu pieejamība datu analīzei" [18] veikto izpēti par VMware vSphere virtualizācijas risinājumu pielietošanu datu analīzei, kā arī to, ka brīvu mākoņresursu pieejamais daudzums ir aptuveni 70% un tie potenciāli ir izmantojami datu analīzei. Bāzes risinājums tiks papildināts ar papildus līmeņu loģiku izmantojot Kubernetes, lai sīkāk menedžētu brīvos resursus, un risinājums tiek fokusēts uz privātu mākoņu izvietošanas modeli, kuru sistēmas pamatā ir VMware risinājumi.

Darba Mērķis: Veiksmīgi izstrādāt VMware un Kubernetes risinājumu datu analīzes veikšanai, nepārsniedzot absolūto CPU gaidīšanas laiku par 5%.

Darba uzdevumi:

- Iepazīties ar virtualizāciju konteineros.
- Izpētīt un pielietot Kubernetes.
- Izpētīt un pielietot Docker konteinerus papildinot Kubernetes.
- Pielietot VMware vSphere mākoņu hipervizoru.
- Izvērtēt un analizēt iegūtos datus, izdarīt secinājumus.

Darba struktūra: Darbs sastāv no ievada, anotācijas latviešu un angļu valodā, 4 nodaļām, 30 apakšnodaļām, izmantotās literatūras saraksta, secinājumiem un pielikumiem. Darbā ir 12 attēli un 9 tabulas.

1. VIRTUALIZĀCIJA

Virtualizācija ir tehnoloģija, kas ļauj no vienas fiziskas aparatūras sistēmas izveidot vairākas simulētas vides vai īpašus resursus. Programmatūra, ko sauc par hipervizoru, tieši savienojas ar aparatūru un ļauj sadalīt vienu sistēmu atsevišķās, atšķirīgās un drošās vidēs, kas pazīstamas kā virtuālās mašīnas (VM). Šīs VM paļaujas uz hipervizora spēju atdalīt iekārtas resursus no aparatūras un atbilstoši tos izplatīt.[2, 4]

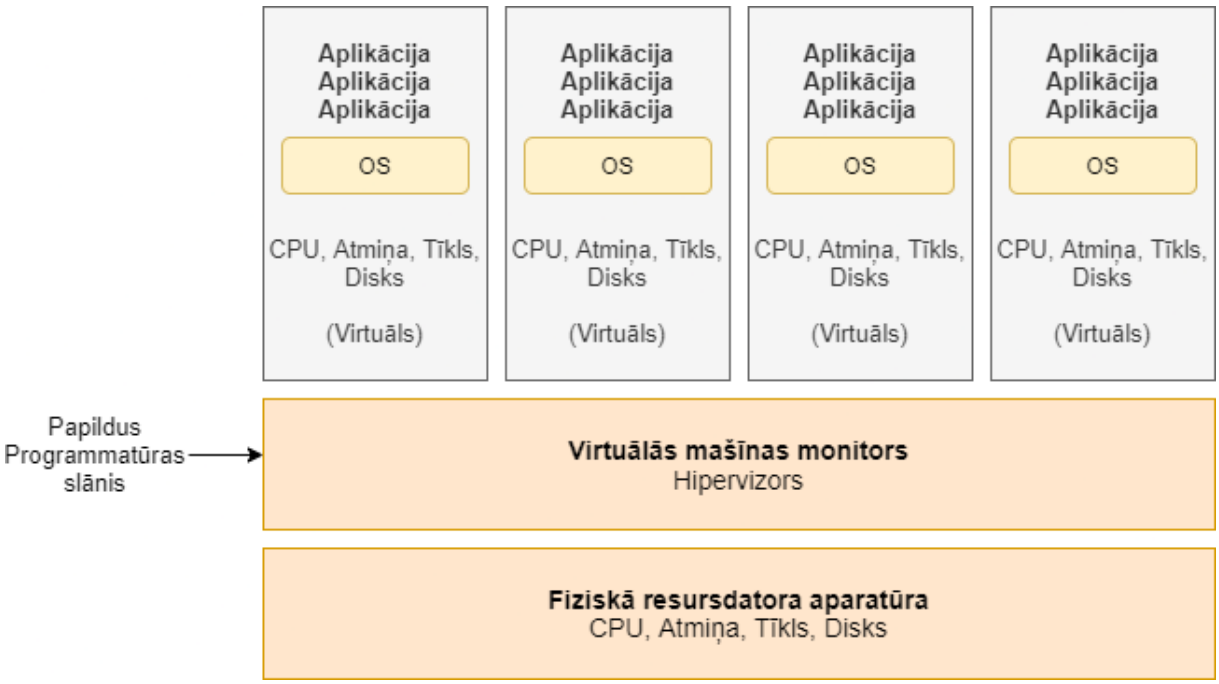
Ir Daudzi iemesli, kāpēc izmantot virtualizāciju. Galddatora lietotājiem virtualizācija visbiežāk tiek izmantota iespēja palaist lietojumprogrammas, kas paredzētas citai operētājsistēmai, nemainot datorus vai pārstartējot datoru citā sistēmā. Servera administratoriem virtualizācija piedāvā arī iespēju darbināt dažādas operētājsistēmas, bet, iespējams, vēl svarīgāk, tās piedāvā veidu, kā lielu sistēmu segmentēt daudzās mazākās daļās, ļaujot serveri efektīvāk izmantot vairākiem lietotājiem vai lietojumprogrammām ar dažādām vajadzībām. Virtualizācija arī ļauj izolēt VM, saglabājot programmas, kas darbojas virtuālās mašīnas iekšienē, pasargājot no procesiem, kas notiek citā virtuālajā mašīnā tajā pašā resursdatorā.[1, 3]

1.1 Kā strādā virtualizācija?

Kā iepriekš tika minēts, programmatūra, ko sauc par hipervizoru, nošķir fiziskos resursus no virtuālajām vidēm – lietas, kuram šie resursi ir nepieciešami. Hipervizori var atrasties virsū uz citām operētājsistēmām (piemēram, uz galddatora Windows operētāj sistēmas) vai tikt instalēti tieši aparatūrā (piemēram, serverī kā tā operētājsistēma), kā lielākā daļa uzņēmumu virtualizējas. Hipervizori ņem sistēmas fiziskos resursus un sadala tos, lai katra VM tos varētu izmantot.

Resursi Tiek sadalīti pēc nepieciešamības no fiziskās vides uz daudzajām VM. Lietotāji mijiedarbojas ar virtuālo vidi (parasti to sauc par viesu mašīnu vai virtuālo mašīnu) un veic aprēķinus. VM darbojas kā viens datu fails. Tāpat kā jebkuru digitālo failu, to var pārvietot no viena datora uz otru, atvērt jebkurā no tiem un sagaidīt, ka tas darbosies tāpat.

Kad darbojas virtuālā vide un lietotājs vai programma izdod instrukciju, kurai nepieciešami papildu resursi no fiziskās vides, hipervizors nosūta pieprasījumu fiziskajai sistēmai un saglabā kešatmiņā izmaiņas – tas viss notiek gandrīz ar resursdatora pamata ātrumu.[5] Zemāk (skat. 1.1. att.) var redzēt hipervizora sistēmas uzbūvi.



1.1. att. ***Virtualizācijas sistēmas uzbūve***

1.2 Kas ir hipervizors?

Hipervizors ir svarīga programmatūras daļa, kas padara iespējamu virtualizāciju. Tas abstrahē viesu mašīnas un operētājsistēmas, kurā tie darbojas, no faktiskās sistēmas. Tie izveido virtualizācijas slāni, kas atdala procesoru, RAM, diska atmiņu, un citus fiziskos resursus no izveidotajām virtuālajām mašīnām. Datoru, kurā tiek instalēts hipervizors, sauc par resursdatoru, salīdzinot ar viesu virtuālajām mašīnām, kas darbojas virs tām.

Hipervizori atdarina pieejamos resursus, lai viesu virtuālās mašīnas tos varētu izmantot. Neatkarīgi no operētājsistēmas, kas atrodas uz virtuālās mašīnas, tā domās, ka tās rīcībā ir reāla fiziskā aparatūra. No virtuālo mašīnu viedokļa nav atšķirības starp fizisko un virtualizēto vidi. Virtuālās mašīnas nezina, ka hipervizors tās ir izveidojis virtuālā vidē, vai arī viņi dala pieejamo skaitļošanas jaudu. Virtuālās mašīnas darbojas vienlaikus ar aparatūru, kas tās darbina, tāpēc tie ir pilnībā atkarīgi no aparatūras stabilās darbības.

Hipervizori tradicionāli ir sadalīti divās klasēs: Pirmā tipa jeb “Bare metal”, kas viesu virtuālās mašīnas vada tieši uz sistēmas aparatūras, būtībā rīkojoties kā operētājsistēma. Otrā

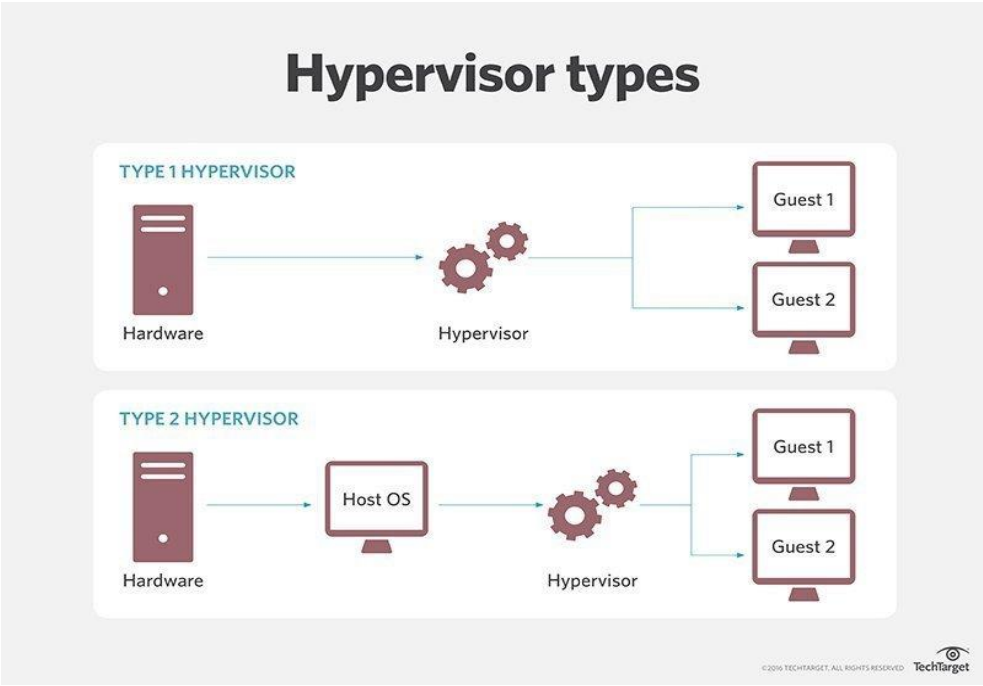
tipa hipervizori izturas vairāk kā tradicionālās lietojumprogrammas, kuras var sākt un apturēt kā parastu programmu.[7]

1.2.1 Pirmā tipa hipervizors

Pirmā tipa hipervizors ir programmatūras slānis, kuru instalē tieši uz fiziskā servera un tā pamatā esošās aparatūras, kā to, piemēram, darītu ar Windows instalāciju. Starp serveri un hipervizoru nav programmatūras vai operētājsistēmas, tāpēc nosaukums ir "bare-metal" hipervizors. Pirmā tipa hipervizors izceļas ar to, ka tam ir tieša piekļuve aparatūrai, tā rezultātā tas nodrošina labu veiktspēju un stabilitāti, jo tas nedarbojās uz kādas citas operētājsistēmas. Parasti šis hipervizora tips tiek izmantots uzņēmumu vidēs.[7]

1.2.2 Otrā tipa hipervizors

Otrā tipa hipervizori ir programmatūras līmeņa hipervizori. Tas nozīmē, ka atšķirībā no pirmā tipa hipervizors VM nav tiešas piekļuves pamatā esošajai aparatūrai. Tādējādi VM ir jānodod aparatūras resursu pieprasījumi resursdatora OS, kas pēc tam VM vārdā piekļūst fiziskajai aparatūrai. Šie hipervizori tiek uzstādīti uz sistēmas esošās operētājsistēmas, tādēļ tie ir ērtāki nekā pirmā tika hipervizori, bet parasti tie nedarbojas tik labi, kā pirmā tipa hipervizori. Otrā tipa hipervizori ir labi, piemēram, ja vēlas pārbaudīt kā kāds hipervizors strādā, tad nav vajadzīgs tam veltīt speciālu mašīnu, tā vietā izmantojot esošu sistēmu. [7, 8]



1.2. att. ***Hipervizora tipi***

1.3 Kas ir virtuālā mašīna

Virtuālā mašīna ir virtuāla vide, kas strādā kā dators iekšā datorā. Šī virtuālā mašīna strādā uz izolētas partīcijas uz resursdatora ar savu CPU jaudu, atmiņu un operētājsistēmu, un citiem resursiem un tie tiek atdalīti no resursdatora. Šis atļauj lietotājiem darbināt lietotnes uz virtuālās mašīnas, kuras normālos apstākļos vajadzētu darbināt uz citas atsevišķas darba stacijas.

Virtuālās mašīnas atļauj biznesiem darbināt operētājsistēmas, kas strādā kā pilnībā nošķirts un izolēts dators lietotnes logā uz resursdatora. Virtuālās mašīnas var tik uzstādītas ar dažādiem jaudas līmeņiem, lai apmierinātu šo mašīnu jaudas vajadzības, lai tās būtu spējīgas darbināt programmatūru, kurai ir vajadzīga cita operētājsistēma, vai arī testētu lietotnes uz drošas izolētas platformas. [6]

1.3.1 Virtuālo mašīnu priekšrocības

- Virtuālās mašīnas var darbināt vairākas operētājsistēmu vides uz viena fiziska datora, taupot fizisko vietu, laiku un menedžmentu.
- Virtuālās mašīnas atbalsta mantotas programmatūras, samazinot migrēšanas izmaksas uz jaunām operētājsistēmām.

1.3.2 Virtuālo mašīnu mīnusi

- Darbinot vairākas virtuālās mašīnas uz vienas fiziskās mašīnas var rezultēt nestabilu performanci, ja infrastruktūras prasības netiek sasniegtas.
- Virtuālās mašīnas ir mazāk jaudīgas un efektīvas, kā pilns fiziskais dators. Lielākā daļa uzņēmumu izmanto kombināciju ar fizisko un virtuālo infrastruktūru, lai balansētu attiecīgos plusus un mīnusus.

1.4 Virtualizācijas tipi

Visas komponentes tradicionālā datu centrā vai IT infrastruktūrā var virtualizēt ar vairākiem specifiskiem virtualizācijas tipiem:

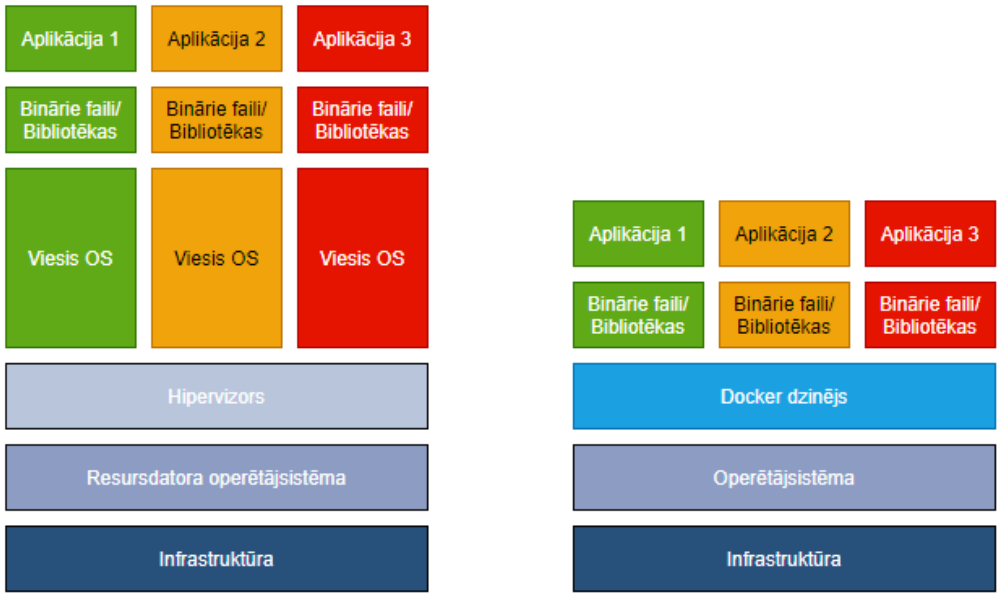
- Aparatūras virtualizācija: Kad virtualizē aparatūru, virtuālās versijas no datora un operētājsistēmas ir izveidotas un konsolidētas vienā primārā, fizikālā serverī. [6]
- Programmatūras virtualizācija: Programmatūras virtualizācijas izveido datorsistēmu ar aparatūru, kas ļauj izveidot vienu vai vairākas viesu operētājsistēmas. Piemēram,

Android operētājsistēma var tikt darbināta uz resursdatora, kas izmanto Windows, pielietojot vienu un to pašu aparatūru, ko izmanto resursdators. Papildus, lietotnes var tikt virtualizētas un nogādātās no servera līdz lietotāja iekārtai, piemēram, kā portatīvā datora vai viedtālruņa. Tas ļauj lietotajiem piekļūt pie centrāli uzturētas lietotnes, strādājot attālināti.[6]

- Krātuves virtualizācijas: Krātuves var tikt virtualizētas konsolidējot vairākas fiziskās krātuves iekārtas, lai tās parādītos kā viena krātuves iekārta. Ieguvumi ir palielināta performance un ātrums, slodzes līdzsvarošana un samazinātas izmaksas. Krātuvju virtualizācijas arī var palīdzēt ar katastrofu seku novēršanu, jo virtuālās krātuves var tikt dublētas un viegli pārvietotas uz citu lokāciju, samazinot dīkstāves.[6]
- Tīkla virtualizācija: Vairāki apakštīkli var tikt izveidoti uz viena fiziskā tīkla kombinējot iekārtas vienā programmatūras balstītā virtuālā tīkla resursā. Tīkla virtualizācija arī sadala pieejamo tīkla jostas platumu vairākos izolētos kanālos, kur katrs var tik piešķirts serveriem vai citām iekārām reālā laikā. Plusi iekļauj palielinātu uzticamību, tīkla ātrumu, drošību un labāku datu monitoringu uz datu izmantošanu. Tīkla virtualizācija var arī būt laba izvēle priekš kompānijām ar augstu apjomu ar lietotājiem, kuriem ir vajadzīga piekļuve visu laiku.[6]
- Darbvirsmas virtualizācija: ir izplatīts virtualizācijas tips, kurš atdala darba virsmas vides no fiziskas iekārtas un glabā darbvirsmu uz attālināta servera, atļaujot lietotājiem piekļūt viņu darbvirsmai no jebkuras vietas uz jebkuras iekārtas. Papildus vieglai piekļuvei, ieguvumi no virtuālajām darbvirsmām ir labāka datu aizsardzība, izmaksu samazināšana uz lietojumprogrammu licencēm un atjauninājumiem, un atvieglotu menedžmentu.[6]

1.5 Konteineru virtualizācija

Bez tradicionālās virtualizācijas, kur tiek uzstādītas virtuālas mašīnas uz hipervizora, ir arī veids kā uzstādīt lietotnes ar konteineru palīdzību. Šie konteineri, tāpat kā virtuālās mašīnas uz hipervizora ir izolētas viena no otras, lai tās nevajadzīgi netraucētu viena otras darbībai vai uzturēšanai. Bet starpība starp šīm idejām ir tāda, ka virtuālās mašīnas ir apjomīgas – katrai no tām ir nepieciešama sava operētājsistēma, tāpēc arī to lielums parasti ir salīdzinoši liels, un katru no tām var būt darbs uzturēt un uzlabot. Konteineri savukārt izolē lietojumprogrammu izpildes vides vienu no otras, bet koplieto pamatā esošos resursus. Kā rezultātā tiek patērēti daudz mazāk resursu nekā virtuālajās mašīnās, kā arī konteineri var tikt uzstādīti gandrīz nekavējoties. Konteineri arī nodrošina ļoti efektīvu un detalizētu mehānismu kādu programmatūras komponentu apvienošanai vienā lietojumprogrammu un pakalpojumu klāstā, kā arī mehānismus programmatūras komponentu atjaunināšanai un uzturēšanai. [13, 14] Attēlā (skat. 1.3. att.) ir vizuālizēts potenciāls resursu ieguvums, jo nav vajadzīgs izmantot apjomīgas operētājsistēmas kādu lietotņu uzstādīšanai un darbināšanai.



1.3. att. ***Salīdzinājums starp virtualizāciju ar hipervizoru un Docker konteinerizāciju***

1.6 Kas ir mākonis

Kopš 1970. gadu vidus skaitļošana ir notikusi vairākos posmos – no lieldatoriem līdz minidatoriem, līdz personālajiem datoriem, līdz tīkla skaitļošanai, klienta-servera skaitļošanai un distributētai skaitļošanai. Tagad, nonākot pilnā lokā, skaitļošana pāriet uz mākoņiem, uz attālinātiem skaitļošanas resursiem, kuri ir sasniedzami izmantojot internetu.

Atkarībā no tā, kā skatās uz mākoņdatošanu, to var aprakstīt dažādi. Ir vairākas definīcijas, taču ASV Nacionālais standartu un tehnoloģiju institūts (NIST) piedāvā klasisku

definīciju, kas ietver mākoņdatošānas galvenos elementus un īpašības. [9] Definīcija ir sekojoša:

Mākoņdatošana ir modelis, kas nodrošina visur esošu, ērtu, pēc pieprasījuma tīkla piekļuvi koplietojamam, konfigurējamam, skaitļošanas resursu kopumam (piemēram, tīkli, serveri, krātuves, lietojumprogrammas un pakalpojumi), kurus var ātri nodrošināt un atbrīvot ar minimālu pārvaldības piepūli vai pakalpojumu sniedzēja mijiedarbību. Šis mākoņa modelis sastāv no pieciem būtiskiem raksturlielumiem, trim pakalpojuma modeļiem un četriem izvietojuma modeļiem.[11]

1.6.1 Galvenie mākoņa raksturlielumi

Kā tiek minēts 2.1 NIST definīcijā mākoņdatošanai ir pieci būtiski raksturlielumi, kas palīdz atšķirt mākoņdatošanu no citām tradicionālām skaitļošanas formām. Šie raksturlielumi tiek uzskaitīti tabulā (skat. 1.1. tabulu).

Mākoņa raksturlielums	Apraksts
Pašapkalpošanās pēc pieprasījuma (on-demand self-service)	Patērētājs var vienpusēji automātiski nodrošināt nepieciešamās skaitļošanas iespējas, piemēram, servera laiku un tīkla krātuvi, neprasot cilvēku mijiedarbību ar katru pakalpojumu sniedzēju.[10, 11]
Plaša piekļuve tīklam	Iespējas ir pieejamas tīklā, un tām var piekļūt, izmantojot standarta mehānismus, kas veicina neviendabīgi plānu vai biezu klientu platformu (piemēram, mobilo tālruņu, planšetdatoru, klēpjdatoru un darbstaciju) izmantošanu.[10, 11]
Resursu apvienošana	Pakalpojuma sniedzēja skaitļošanas resursi tiek apvienoti, lai apkalpotu vairākus patērētājus, izmantojot vairāku īrnieku modeli, ar dažādiem fiziskiem un virtuāliem resursiem, kas tiek dinamiski piešķirti un pārdalīti atbilstoši patērētāju pieprasījumam. Atrašanās vietas neatkarība ir tāda, ka klientam parasti nav kontroles vai zināšanu par precīzu nodrošināto resursu atrašanos vietu, taču viņš var norādīt atrašanās vietu augstākā abstrakcijas līmenī (piemēram, valstī vai datu

	centrā). Resursu piemēri ir krātuve, procesora jauda, atmiņa un tīkla joslas platums.[10, 11]
Ātra elastība un mērogojamība	Spējas var elastīgi nodrošināt un atbrīvot, dažos gadījumos automātiski, lai ātri mērogotos uz āru un uz iekšu, proporcionāli pieprasījumam. Patērētājam nodrošināšanai pieejamās iespējas bieži šķiet neierobežotas un tās var jebkurā laikā izmantot jebkurā daudzumā. [10, 11]
Izmērīts serviss	Mākoņu sistēmas automātiski kontrolē un optimizē resursu izmantošanu, izmantojot mērīšanas iespējas kāda abstrakcijas līmenī, kas atbilst pakalpojuma tipam (piemēram, krātuve, procesora jauda, tīkla joslas platums un aktīvie lietotāju konti). Resursu izmantošanu var uzraudzīt un kontrolēt un ziņot, nodrošinot pārredzamību gan izmantotā pakalpojuma sniedzējam, gan patērētājam. [10, 11]

*1.1. Tabula **Mākoņdatošanas raksturlielumi***

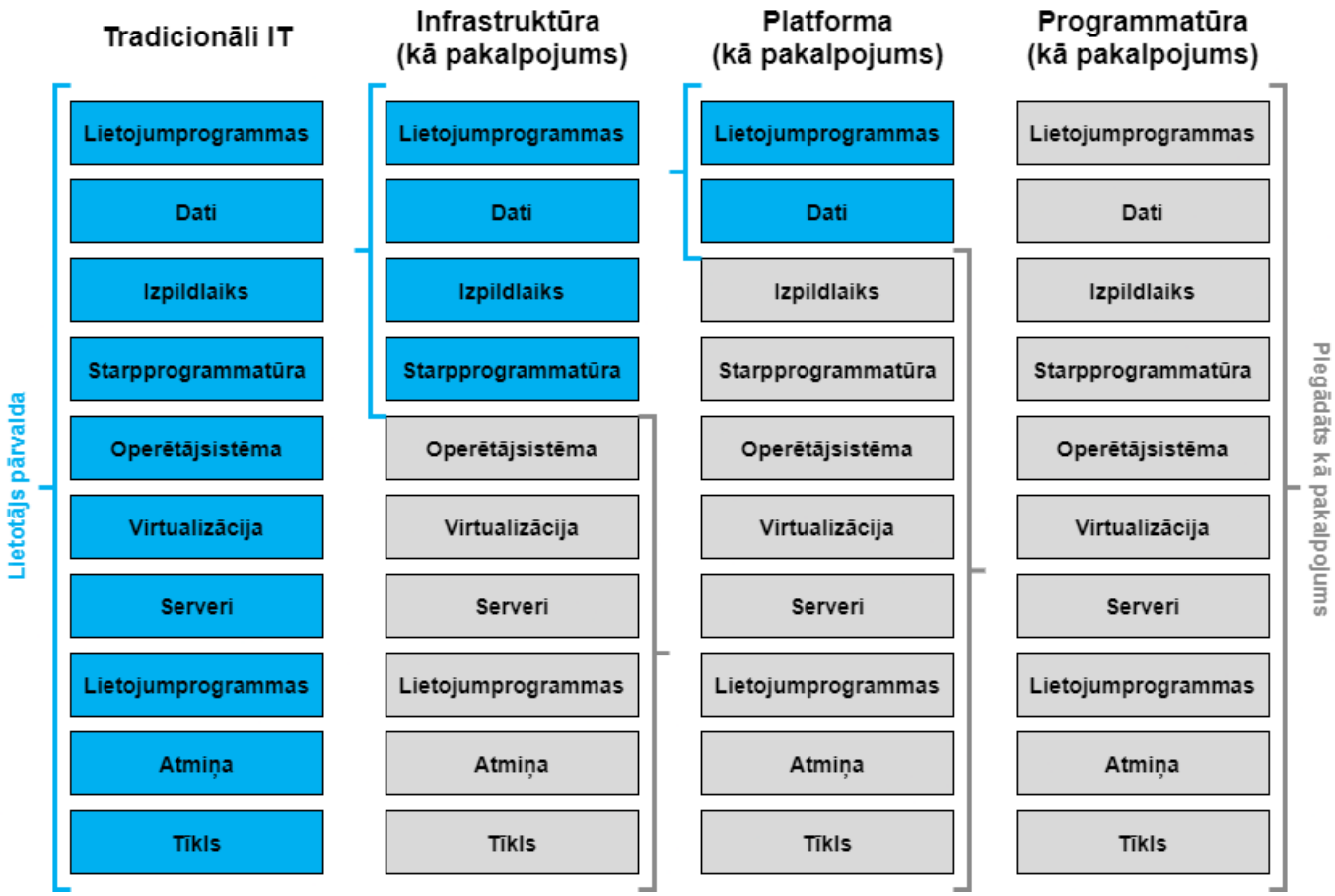
1.7 Galvenie mākoņpakalpojumu modeļu tipi

Skaitļošanas vai tīkla resursus, lietojumprogrammas vai jebkāda cita veida IT pakalpojumus, ko lietotājam piedāvā mākonis, sauc par mākoņpakalpojumu. Mākoņpakalpojumi ir sākot ar vienkāršām lietojumprogrammām, piemēram, kā e-pastiem, kalendāriem, tekstu apstrādes un fotoattēlu kopīgošanas, līdz dažāda veida sarežģītām uzņēmumu lietojumprogrammām un skaitļošanas resursiem, ko galvenie mākoņpakalpojumu sniedzēji piedāvā kā pakalpojumus.

Atkarībā no piedāvāto pakalpojumu veida mākoņpakalpojumus var iedalīt trīs galvenajās kategorijās, kas ir: Programmatūru kā pakalpojumu (SaaS), platformu kā pakalpojumu (PaaS) un infrastruktūru kā pakalpojumu (IaaS). Papildus šiem pamatpakalpojumiem tiek piedāvāti vairāki mākoņu atbalsta pakalpojumi, piemēram, drošība kā pakalpojums un identitātes un piekļuves pārvaldība kā pakalpojums. Katru pakalpojumu kategoriju var izmantot neatkarīgi vai izmantot kombinācijā kopā ar citiem. [9]

Katrs modelis nodrošina abstrakcijas līmeni, kas samazina patērētāju patērēto enerģiju un laiku, veidojot un izvietojot sistēmas. Tradicionālā lokālā datu centrā, IT komandai ir jāveido

un jāpārvalda visu sistēmu darbība, kas, piemēram, ir serveru uzturēšana, programmatūru integrācija un regulāra atjauninājumu veikšana, drošības nodrošināšana augstā līmenī un daudzas citas lietas. Tādēļ lai mazinātu patērēto laiku un enerģiju šo sistēmu darbības nodrošināšanai, nāk palīgā mākoņpakalpojumi. Katrs modelis nodrošina automatizācijas un abstrakcijas līmeni, kas palīdz nodrošināt lielāku veiklību mākoņpakalpojumu patērētājiem, lai viņi varētu veltīt vairāk laika savām biznesa problēmām un mazāk laika un enerģijas infrastruktūras pārvaldībai. [10]



1.4. att. ***Mākoņpakalpojumu slāņu salīdzinājums***

1.7.1 Programmatūra kā pakalpojums (SaaS)

Programmatūra kā pakalpojumu mākoņus sauc arī par programmatūras mākoņiem. SaaS modelī lietojumprogrammas uztur mākoņpakapojuma sniedzējs, kurš apstrādā visu infrastruktūru, visu lietojumprogrammu loģiku, visus izvietojumus un visu, kas attiecas uz produkta vai pakalpojuma piegādi. Tas viss novērš nepieciešamību instalēt un darbināt lietojumprogrammas lokāli lietotāja datorā, tādējādi atbrīvojot lietotājus no aparatūras un programmatūras uzturēšanas un atjaunināšanas. Lietotājam ir tikai jākonfigurē daži lietojumprogrammu parametri un jāpārvalda lietotāji. Programmatūras licences nepieder

lietotajiem. Lietotājiem ir tikai jāmaksā par izmantoto pakalpojumu atkarībā no to lietošanas. Daži ļoti izplatītu SaaS lietojumprogrammu piemēri ir – klientu attiecību pārvaldība (CRM), uzņēmuma resursu plānošana (ERP), algu aprēķināšana, grāmatvedība un citas biznesa programmatūras. [9, 10]

1.7.2 Platforma kā pakalpojums (PaaS)

PaaS modelī platformu un rīkus lietojumprogrammu izstrādei un starpprogrammatūras sistēmām uztur mākoņpakalpojuma sniedzējs, un tie tiek piedāvāti lietojumprogrammu izstrādātājiem, ļaujot tiem izstrādāt kodu un izvietot to bez tiešas mijiedarbības ar pamatā esošo infrastruktūru. PaaS nodrošina lielāko daļu rīku un iespēju, kas nepieciešamas lietojumprogrammu un pakalpojumu veidošanai un piegādei, piemēram, darbplūsmas iespējas priekš lietojumprogrammu dizaina veidošanas, izstrādes, testēšanas, izvietošanas un mitināšanas.

PaaS pakalpojumi ir pilnībā pieejami no interneta. PaaS pakalpojuma sniedzēji pārvalda lietojumprogrammu platformu un nodrošina izstrādātājus ar rīku komplektu, lai paātrinātu izstrādes procesu. Tas nozīmē, ka izstrādātāji atsakāš no zināmas elastības pakāpes ar PaaS, jo tos ierobežo pakalpojuma sniedzēja piedāvātie rīki un programmatūras kaudzes. Izstrādātājiem ir arī maz vai vispār nekontrolē zemāka līmeņa programmatūras vadīklas, piemēram, atmiņas piešķiršanu un kaudžu konfigurācijas (piemēram, pavedienu skaitu, kešatmiņās daudzumu). Pakalpojuma sniedzēji to visu kontrolē un var pat ierobežot, cik lielu skaitļošanas jaudu pakalpojumu patērētājs var izmantot, lai piegādātājs varētu nodrošināt platformas mērogošanu visiem vienādi.

Viena no PaaS priekšrocībām ir tā, ka šīs platformas integrējas ar daudziem trešo pušu programmatūras risinājumiem, kurus bieži dēvē par spraudņiem, papildinājumiem vai paplašinājumiem. Šeit ir daži paplašinājumu kategoriju piemēri, kurus var atrast visvairāk nobriedušajos PaaS risinājumos:

- Datubāzes
- Žurnāla veidošana
- Monitorings
- Drošība
- Kešošana
- Meklēšana
- E-pasts

- Analītika
- Maksājumi

Izmantojot API, lai piekļūtu daudziem trešo pušu risinājumiem, izstrādātāji var nodrošināt neveiksmju novēršanu, augsta līmeņa pakalpojumu līgumus (SLA) un gūt milzīgu ātrumu tirgū un izmaksu efektivitāti, jo nav jāpārvalda un jāuztur tehnoloģija aiz API. Tas ir liels PaaS plus, kurā izstrādātāji var ātri apkopot nobriedušu un pārbaudītu trešo pušu risinājumu kolekciju, vienkārši izsaucot API un nav jāiziet iepirkuma process, kam seko katra trešās puses rīka ieviešanas process. PaaS ļauj uzņēmumiem koncentrēties uz pamatkompetencēm un integrēties ar vislabākajiem instrumentiem tirgū. [9, 10]

1.7.3 Infrastruktūra kā pakalpojums (IaaS)

IaaS mākonī neapstrādāta datoru infrastruktūra, piemēram, serveri, centrālais procesors, atmiņa, tīkla aprīkojums un datu centra iespējas, tiek piegādātās kā pakalpojums pēc pieprasījuma. Uz šīs pakalpojumu kolekcijas, kurai var piekļūt un automatizēt no koda vai tīmekļa pārvaldītas konsoles, izstrādātājiem ir jāizstrādā visas lietojumprogrammas, un administratoriem ir jāinstalē, jāpārvalda un jāatjaunina trešo pušu risinājumi, bet tiem nav jāpārvalda fiziska infrastruktūra. Tas nozīmē, ka izmantojot IaaS, virtuālā infrastruktūra ir pieejama pēc pieprasījuma, un tā var sākt darbu dažu minūšu laikā, izsaucot API vai palaižot to no tīmekļa pārvaldības konsoles. IaaS nodrošina virtuālo datu centru iespējas, lai pakalpojumu lietotāji varētu vairāk koncentrēties uz lietojumprogrammu izveidi un pārvaldību, un mazāk tērēt resursus uz datu centru infrastruktūras pārvaldīšanas.[9, 10]

1.8 Mākoņu izvietošanas modeļi

Pamatojoties uz to, kurš un kur mākoni ir izvietojis, kam pieder un to pārvalda, un kas ir tā galvenie lietotāji, mākoņi tiek iedalīti četrās kategorijās: publiskais mākonis, privātais mākonis, kopienas mākonis un hibrīds mākonis.[9]

1.8.1 Publiskais mākonis

Publiskais mākonis ir visizplatītākais un visplašāk pazīstamais mākoņa izvietošanas modelis. Šī mākoņa infrastruktūra ir paredzēta atklātai izmantošanai plašākai sabiedrībai, kas nozīmē, ka pakalpojums ir atvērts ikvienam – uzņēmējdarbībai, rūpniecībai, vadībai, bezpeļņas organizācijām un privātpersonām. Mākoņa infrastruktūra tomēr pieder un to pārvalda mākoņpakalpojuma sniedzējs. [9, 11]

1.8.2 Privātais mākonis

Privātu mākoni izvieto, nodrošina un kontrolē uzņēmums, kas atrodas aiz ugunsmūra savām vajadzībām. Daži uzņēmumi, kuri nevēlas nonākt publiskajos mākoņos, jo pastāv bažas par tiem un atbilstības prasībām, daži uzņēmumi izmanto savas mākoņdatošanas vides ekskluzīvai (un viņu biznesa partneru) lietošanai. Tādējādi, iegūstot savu mākoni, viņi iegūst darbības efektivitāti, efektīvi izmanto esošos resursus, ja tādi ir, un pilnībā kontrolē mākoni lietojumprogrammas un datus par mākoni. [9]

1.8.3 Kopienas mākonis

Kopienas mākonis ir pazīstams kā nozares mākonis vai vertikāls mākonis. Tas ir optimizēts un speciāli izvietots lietošanai noteiktā nozares nozarē vai lietotāju grupā, lai tas atbilstu īpašām prasībām, lai risinātu viņiem izšķirošās problēmas. [9]

1.8.4 Hibrīds mākonis

Hibrīds mākonis ir divu vai vairāku atšķirīgu mākoņu infrastruktūru (privāta, kopienas vai publiska) sastāvs. Šajā modelī uzņēmumi izmanto gan publiskos, gan privātos mākoņus – izvietojot savus mazāk kritiskos, zema riska pakalpojumus publiskajā mākonī un uzņēmējdarbībai kritiskās pamatprogrammas iekšējā privātā mākonī. Hibrīds modelis ļauj selektīvu implementāciju kas ļauj adresēt problēmas, drošību, atbilstību un kontroles zaudēšanu, kā arī ļauj pieņemt publiskus mākoņus, kas piedāvā izmaksu priekšrocības un citas lietojuma iespējas. [9]

2. PROBLĒMAS APSKATS

2.1 Problēmas apraksts

Mūsdienās mākoņpakalpojumi ir kļuvuši par neatņemamu uzņēmumu infrastruktūras daļu, bet kaut gan, tie uztur lielu daļu mūsdienu interneta resursu, to veiktspējas potenciāls netiek pilnībā izmantots. Pēc vairāku pētījumu secinājumiem, ir zināms, ka mākoņu resursu vidējā noslodze ir salīdzinoši zema, kas ir aptuveni 25-30% no CPU jaudas un 40-50% no RAM resursiem. [15, 16, 17]. Bet pēc "Brīvu mākoņresursu pieejamība daru analīzei" kursa darba (turpmāk tekstā – "bāzes darbs") rezultāta var secināt, ka veiksmīgi izmantot visus resursus dažādās situācijās ir sarežģīti un bieži neiespējami, ja resursdatora fiziskie resursi tiek izdalīti vairāk kā tie ir pieejami. Uz šī darba arī tiek balstīts turpmākais pētījums un risinājums [18]

Šo zemo noslodzi var paskaidrot ar vairākiem faktoriem, kas ir pīķa apstrāde, risku pārvaldīšana, nākotnes pieprasījumu apstrāde kā arī arhitektūras dizains.

Pīķa apstrāde ideja balstās uz to, ka mākoņu infrastruktūrai ir jābūt nodrošinātai tā, lai tā varētu apmierināt augstu pieprasījumu daudzumu. Vienīgi šādu situāciju nenotiek bieži, tādēļ uz mākoņu serveriem bieži ir neizmantoti resursi. Risku pārvaldīšanas rezultātā mēdz nodrošināt sistēmas ar daudz vairāk resursiem nekā tām ir vajadzīgs, ar domu, lai sniegtais pakalpojums vai produkts vienmēr būtu pieejams. Līdzīgi arī ir ar nākotnes pieprasījumu apstrādāšanas nodrošināšanu, domājot ka nākotnē tiks sniegtais pakalpojums piegādāts vairāk klientiem, kā rezultātā sistēmu tiek nodrošinātas ar ļoti lielu rezervi, kas netiek izmantota. [16]

Lai menedžētu visus šos resursus, parasti tiek izmantoti dažādi virtualizācijas risinājumi no jau atzītiem līderiem mākoņpakalpojumu sfērā, piemēram, kā Amazon AWS. Visi šie mākoņpakalpojumu sniedzēji izmanto sevis izstrādātus, privātus virtualizācijas risinājumus, kas nodrošina mākoņu darbību. Tādēļ, lai apskatītu problēmu un izvirzītu risinājumu, tiks izmantots komerciāls virtualizācijas risinājums VMware vSphere.

Sistēmas izveide tiek realizēta apvienojot VMware vSphere, Kubernetes, Python, Docker tehnoloģijas. VMware vSphere tika izvēlēs, jo šis risinājums ir viens no vadošajiem virtualizācijas risinājumiem, kurš ir komerciāli pieejams un darbs, uz kura bāzes tiek balstīts jaunais risinājums, arī tika veidots ar vSphere tehnoloģiju. Papildus tika izvēlētas Kubernetes dēļ to vieglās mērogojamības, tas ir, jaunu procesu ātru uzstādīšanu kā arī noņemšanu. Docker tika izvēlēts kā konteinerizācijas rīks, kurš roku rokās strādās ar Kubernetēm, un saturēs uzstādāmo procesu komplektāciju. Python tika izvēlēt, priekš kontroles loģikas izveides, jo tas neprasītu daudz resursus, kā arī tam ir viegli integrējami API gan priekš vSphere klienta, gan priekš Kubernetēm.

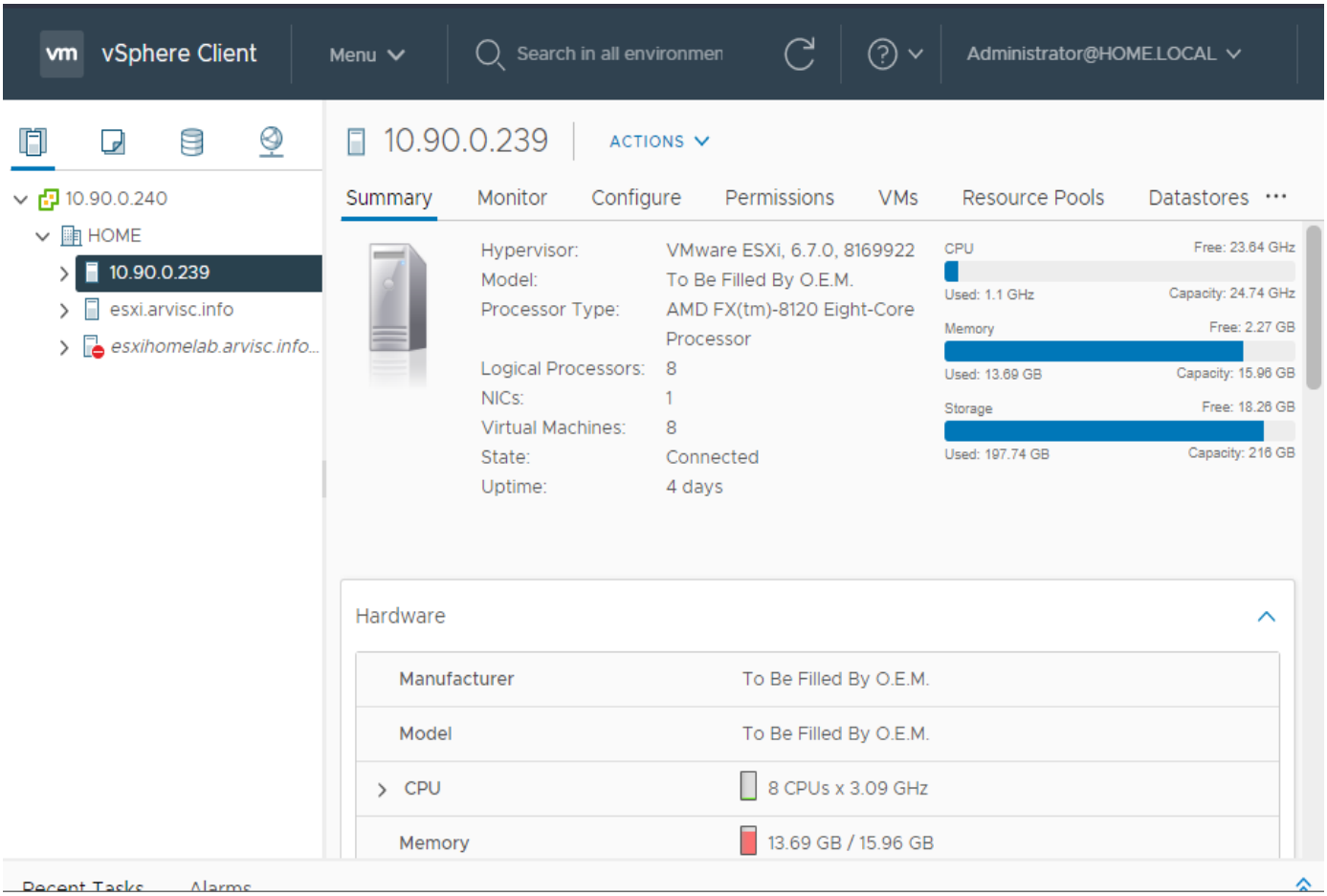
3. SISTĒMAS UN RISINĀJUMA IZPĒTE UN PIELIETOŠANA

3.1 Sistēmas uzstādīšana

Problēmas apskatīšanai un risinājuma izveidošanai, tika uzstādīta testa sistēma. Šīs testa sistēmas pamatā tika uzstādīta VMware ESXi 6.7.0 virtualizācijas operētājsistēma ar sekojošiem parametriem:

- 8 kodolu AMD FX(tm)-8120 procesors
- 16GB RAM atmiņas
- 240GB SSD atmiņas

Tālāk šīs resursdators tika pieslēgts VMware vSphere mākoņu virtualizācijas platformai, kas atrodas uz cita iekšējā tīklā atrodošos servera. Tas tiek darīts, lai apskatītu mākoņu līmeņa virtualizācijas programmatūru, kā arī tiktu klāt papildus konfigurācijas iespējām, kuras nav pieejamas uz VMware ESXi operētājsistēmas. Zemāk (skat. 3.1. att.) var redzēt, resursdatora informāciju apskatot to VMware vSphere lietotnē.



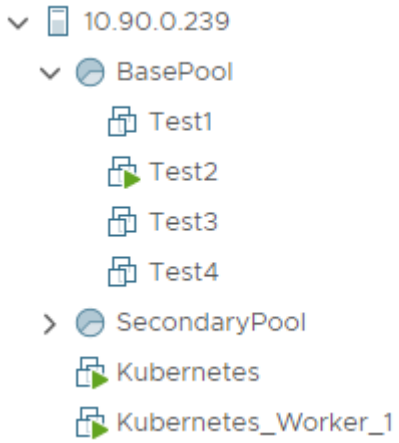
3.1 attēls ***Resursdatora skats vSphere klientā***

Kā redzams 3.1 attēlā, servera pieejamie CPU resursi netiek uzskaitīti kodolos, bet gan ar kodolu darba frekvenci GHz. Tas tiek darīts tādēļ, ka hipervizors nepiešķir virtuālajām

mašīnām fiziskus kodolus, bet gan liek domāt ka tām ir fiziski kodoli, kaut gan realitātē, tiek piešķirti virtuāli kodoli, kuri tiek piešķirti GHz formā. Piemēram, ja uz izveidotās testa sistēmas virtuālajai mašīnai tiek piešķirti divi kodoli, tad realitātē, hipervizors tai piešķīra pieeju 6.18GHz no kopējās jaudas. Testa sistēmas CPU kapacitāte ir mērāma 24.74GHz.

3.1.1 Virtuālās mašīnas

Testa veikšanai tika veikts pieņēmums, līdzīgi kā "Brīvu mākoņresursu pieejamība datu analīzei" kursa darbā, ka ir kaut kādas virtuālās mašīnas, kas simulē 30 procentu slodzi uz resursdatora. Kopumā tāpat kā atsauces darbā tika uzstādītas 4 virtuālās mašīnas pamata slodzes simulēšanai. Datu apstrādei atšķirībā no pamata darba, tika izveidots divas papildus virtuālās mašīnas, kuras veiktu lielu datu apstrādes simulēšanu. Tās atšķiras ar to, ka tās tika veidotas Kubernetes kopa. Zemak (skat. 3.2. att.) var redzēt visu izveidoto vm sarakstu.



3.2. att. ***Saraksts ar darbā izmantotājām virtuālajām mašīnām***

Uz katras virtuālās mašīnas tika uzstādīta viena un tā pati Linux Ubuntu 64bit 20.04 versija un uz attēlā redzamajām Test1 – Test4 mašīnām papildus stress-ng pakonte, ar kuru veica slodzes veidošanu.

3.1.2 Kubernetes virtuālās mašīnas

Kubernetes mašīnas tika veidotas balstoties uz produkcijas vides modeļa. Uz visām Kubernetes mašīnām tika uzstādīts Docker konteinerizācijas risinājums, priekš Kubernetes konteineriem. Pēc tam tika uzstādītas pašas Kubernetes ar sekojošām komandām izpildes secībā:

1. curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
2. sudo apt-get install curl
3. sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"

4. sudo apt-get install kubeadm kubelet kubectl
5. sudo apt-mark hold kubeadm kubelet kubectl
6. sudo swapoff –a
7. sudo hostnamectl set-hostname master-node vai worker01

Šīs komandas tika izmantotas uz visām Kubernetes mašīnām, jo instalācija ir vienāda. 1.-5. punkta komandas uzstādīja Kubernetes vides uz servera. 6. punkta komanda tiek izmantota, jo Kubernetes nav spējīgas pacelt procesus, ja uz tās mašīnas ir ieslēgta mijmaiņas atmiņa. [20]

Tālāk tika veiktas šādas komandas tikai galvenajai Kubernetes mašīnai, kas mūsu gadījumā saucās Kubernetes :

1. sudo kubeadm init --pod-network-cidr=10.244.0.0/16
2. mkdir -p $HOME/.kube
3. sudo chown $(id -u):$(id -g) $HOME/.kube/config
4. sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

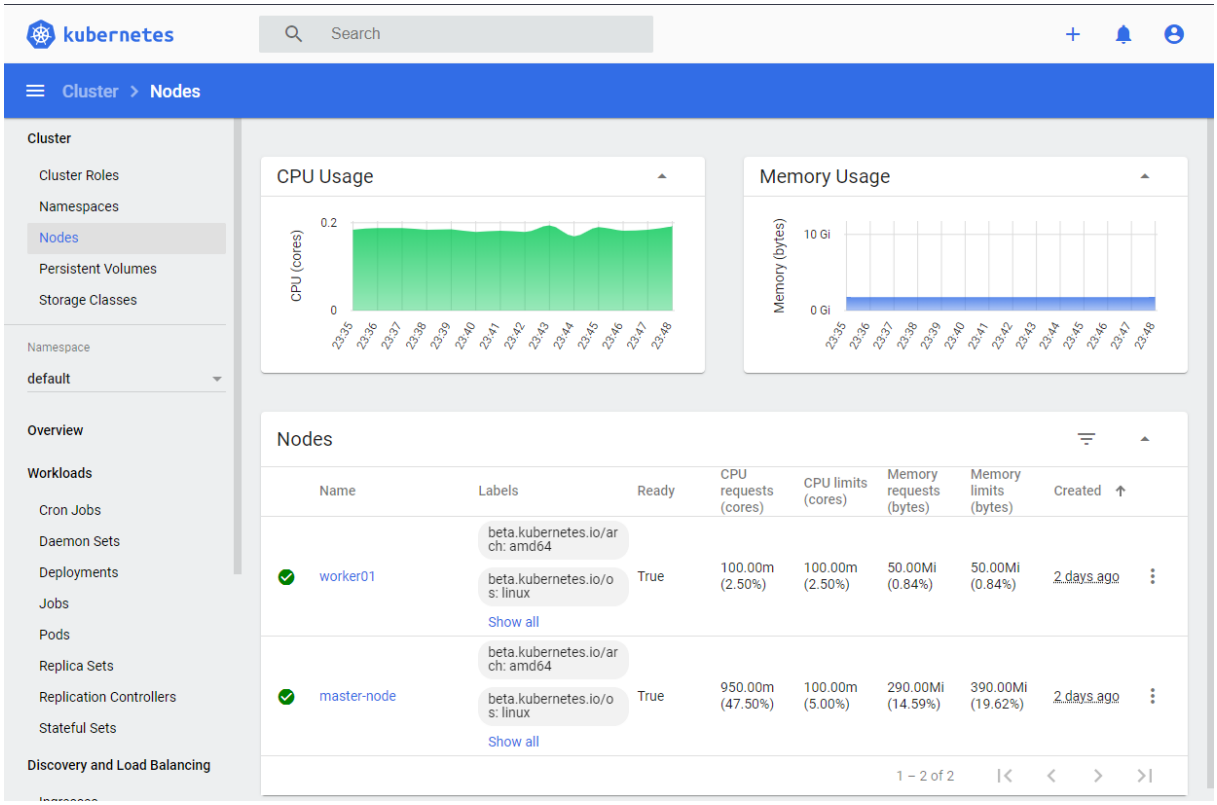
Ar šīm komandām tika izveidots galvenais Kubernetes pārvaldnieks un izveidots iekšējs Kubernešu tīkls, pa kuru Kubernetes galvenais VM sarunāsies ar citiem piesaistītiem Kubernetes strādniekiem.

Bez tā, lai pievienotu pēc tam pārējās Kubernetes virtuālās mašīnas kuras, tālāk sauksim par strādniekiem, izmantoja šādu komandu:

1. kubeadm join --discovery-token abcdef.1234567890abcdef --discovery-token-ca-cert-hash sha256:1234..cdef 1.2.3.4:6443

Ar šīm komandām tika uzstādīta visa Kubernetes kopa, ar kuru var sākt strādāt. Papildus visam šim, tika uzstādīts arī tīmekļa lietotnes risinājums, lai vieglāk un pārskatāmāk būtu saprast, kas notieku visā Kubernetes kopā, bet tas nav svarīgi tālāk izstrādātajā risinājumā.

Zemāk (skat. 3.3 att.) redzams kā izskatās Kubernetes tīmekļa pārvaldnieks ar uzstādīto galveno mašīnu (master-node) un strādnieku (worker01)



3.3 att. ***Kubernetes kopas tīmekļa pārvaldnieks***

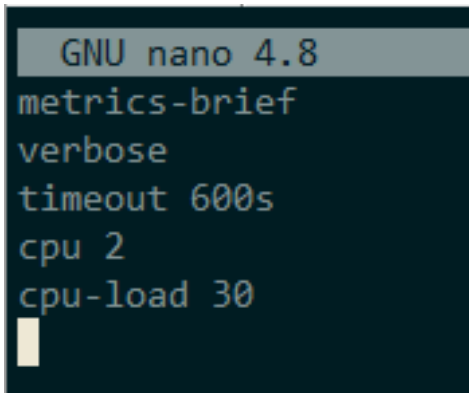
3.2 Problēmas scenārijs

Lai apskatītu kā un cik efektīvi ir iespējams izmantot brīvos CPU resursus, tika veidots scenārijs ņemot pa pamatu bāzes darbā piedāvāto scenāriju[18]. Tika izveidotas un uzstādītas 4 galvenās darba mašīnas, kurās ar stress-ng pakotnes palīdzību simulētu vienmērīgu slodzi, kura aptuveni būtu 30% no CPU jaudas. To panāk ar 2 kodolu piešķiršanu katrai galvenajai mašīnai un simulējot uz katra kodola 30 % slodzi. Kubernetes galvenajai mašīnai tiek iedoti 2 kodoli, jo tas ir ieteicamais minimums, neatkarīgi no tā, ka viņas daudz resursu nepatērē. Bet strādnieka mašīnai tiks pamainīti kodoli starp no 4 līdz 8 kodoliem. Papildus tam, slodze Kubernetes kopā tiks regulēta ar specifisku, paša veidotu Python skripta risinājuu. Galvenais kritērijs pēc kura skatīsies vai drīkst sākt blakus datu apstrādi, būs CPU gatavības rādītājs.

Otrs scenārijs tika veidots līdzīgi pirmajam, bet uzstādot par divām galvenajām mašīnām mazāk, tādējādi testējot, kā izstrādātais risinājums strādā vidē, kur visi CPU resursi nav izdalīti 1:1 galvenajam mašīnām (tiek izdalīti 4 kodoli nevis visi 8 kodoli). Nemainot fiziskās konfigurācijas testa mašīnām, tām tika palielināts simulējamais jaudas līmenis no 30% uz 60%, lai tāpat kā pirmajā scenārijā šīs divas mašīnas kopā simulētu 30% noslodzi uz resursdatora.

3.3 Slodzes simulēšana

Tā kā uz visām mašīnām tika uzstādīts Ubuntu 20.04 serveris, uz visām ir viegli pieejama stress-ng pakotne, kas ļauj veikt dažādus sistēmas stresa testus, sākot no CPU līdz HDD slodzes simulēšanai. Tāpat kā bāzes darbā priekš pamata mašīnām tika izmantota tāda pati darba datne, kurā tika norādīti visi parametri, pēc kuriem jāveic ir slodzes simulēšana. Mainīgie, kuri tika izmantoti ir – Metrics-brief, verbose, timeout, cpu un cpu-load. Metrics-brief un verbose mainīgie dod simulētās slodzes atskaites, ar datiem kā veikto operāciju skaits. Tālāk timeout nozīmē to, cik ilgi tiek simulēta slodze. Šī testa gadījumā tas tika palielināts uz 600 sekundēm no 360 sekundēm bāzes darbā, lai labāk varētu saprast kādu ietekmi rada lielu datu apstrāde, kura tiek veikta paralēli uz lielāku laiku, kā arī tas ir vajadzīgs, lai noskaidrotu, cik lielu labumu dod Python izveidotais risinājums. Cpu mainīgais norāda to, cik CPU kodolus izmantot testā. Un pēdējais izmantotais mainīgais cpu-load, kas paraksta to, cik lielu slodžu procentuāli ir jāsimulē.



3.4. att. ***Strees ng darba datnes definīcija***

Tālāk slodzi Kubernetes mašīnām dos dinamiski, izmantojot Docker konteinerus ar stress-ng pakotni. Ideja ir tāda, ka skatoties pēc brīvo resursu pieejamības tiek palielināts konteineru skaits cenšoties optimāli neietekmēt galvenās mašīnas veiksmīgi, izmantojot kādu daļu brīvo resursu. Katrs no konteineriem veiks slodzes testēšanu ar 100% slodzi un viena kodola. Šo visu dara speciāli izstrādāts Python risinājums priekš šīs problēmas, kurš tiks apskatīts sīkāk, zemāk nodaļā 3.4.

Lai viegli un efektīvi sāktu stress testus uz galvenajām mašīnām, tika izmantots automatizēšanas rīks Ansible. Šis rīks ļauj automatizēt vairāku sistēmu nodrošināšanu, konfigurāciju pārvaldību, lietojumprogrammu izvietošanu un daudzas citas IT vajadzības [19] Ar šo rīku tika izmantots identisks skripts kā bāzes darbā, ar kura palīdzību uz visām galvenajām testa mašīnām tika izpildītas stress-ng komandas (skat. 3.5. att.).[18]

```
GNU nano 4.8                                                    testPlay.yml
---

- name: testPlay
  hosts: testservers
  become: yes
  become_user: root
  tasks:
    - name: launch test jobs
      command: stress-ng --log-file log_$HOSTNAME --job job1
```

3.5. att. ***Ansible-playbook datne, no bāzes darba.***

Tāpat kā bāzes darbā, uz visam galvenajām virtuālajām mašīnām tika nodefinēti un uzlikti faili, kuri tad no mašīnas ar Ansible tika palaisti vienlaikus. Kas beigas atvieglo testēšanas sākšanas darbu, kā arī, slodzes simulēšana notiek sinhroni reizē.

3.4 Python skripts

Lai labāk būtu iespējams limitēt augstu CPU resursu neefektīvu izmantošanu, tika izveidots Python skripts, kurš pilda šo funkciju. Šis Python skripts izmantoja API gan no vSphere klienta gan Kubernetes, lai tam būtu pilns pārskats par sistēmā notiekošo un tas spētu kontrolēt to, kā tiek izmantoti brīvie CPU resursi lielu datu apstrādei. [21, 22] Python skripts sastāv no šādiem failiem – main.py, kubernetesConnection.py, vSphereConnection.py, vm.py un kubernetesDeployment.py. Visus šos failus var apskatīt pielikumā (1. pielikums – 5. pielikums). Šis Python skripts tiek uzstādīts uz galvenās Kubernetes mašīnas, paralēli visiem Kubernetes menedžmenta procesiem. No šīs mašīnas tas tālāk autonomi reaģē uz situāciju resursdatorā un pieņem lēmumus par to, vai drīkst uzsākt papildus datu apstrādes procesus vai nē, kā arī ja resursdatorā pēkšņi tiek izmantoti daudz resursi, un datu analīze ietekmē šo jauno procesu darbību, tad šis skripts ņem nost datu apstrādes procesus, lai vairāk neietekmēt galveno procesu darbību.

3.4.1 Skripta darbības gaita

1. Sākot darbu skripts ar norādītajiem parametriem izveido savienojumu ar Kubernetes galveno mašīnu, izveidojot vairākus dažādus API savienojumus (skat. 3.6. att.). Katrs no šiem API nodrošina pieeju citai funkcionalitātei Kubernetes kopā. ApiClient nodrošina savienojumu ar pašu kopu. AppsV1Api tiek izmantots, lai būtu iespējams izvietot risinājuma konteinerus uz strādnieka mašīnas un tos dinamiski mērogotu uz leju un augšu. CoreV1Api tiek izmantots, lai piekļūtu sīkākai resursu patērēšanas informācijai katrā no sistēmām.

```
# Connects to kubernetes cluster apis.
def connect(self):
    self.api_client = kubernetes.client.ApiClient(self.configuration)
    self.apps_v1_api = kubernetes.client.AppsV1Api(self.api_client)
    self.core_v1_api = kubernetes.client.CoreV1Api(self.api_client)
```

3.6. att. ***Funkcija savienojuma izveidei.***

2. Tālāk tiek izveidots savienojums ar vSphere klientu, iegūstot informāciju par resursdatoru un visām ieslēgtajām virtuālajām mašīnām uz tā. Skriptam tiek atsevišķi padots saraksts ar Kubernetes mašīnu nosaukumiem, jo automātiski to noteikt ir sarežģīti, neievērojot kādu specifisku nosaukumu piešķiršanas praksi. Šis savienojums mums ir vajadzīgs, lai efektīvi varētu monitorēt katras virtuālās mašīnas pieejamos resursus kā arī procesora gatavības rādītāju. Šī informācija ir pieejama ātrākais pa 20 sekunžu intervāliem, tādēļ tiek implementētas 40 sekunžu gaidīšanas, lai nereaģētu pilnīgi uz katru datu atjauninājumu. Tas tiek darīts, lai gadījumos, kad tiek novērots pēkšņs palielinājums resursu izmantošanā, izstrādātais risinājums nereaģē uzreiz, pirms situācija nedaudz nenostabilizējas un ir saprotams, vai šis palielinājums bija tikai uz to brīdi ieslēdzot kādu jaunu procesu, vai arī kaut kas, uz ko mums vajadzētu reaģēt.
3. Pēc tam, kad visi savienojumi izveidoti un iegūti pamata dati par resursdatoru un visām tajā ieslēgtajām virtuālajām mašīnām, tālāk tiek veidots Kubernetes konteinera izvietošanas šablons (skat. 4. pielikumu). No pielikuma zemāk (skat. 3.7. att.) ir izcelta izvietošanas šablona funkcijas “create_deployment_object” sākums, kur tiek norādīta svarīgāka informācija, priekš testēšanas vajadzībām. Zem “image” mainīgā tiek norādīts Docker konteiners ar nosaukumu “alexeiled/stress-ng” šādi iegūstot to pašu stress-ng pakotni, kura tiek izmantota slodzes simulēšanai uz darba mašīnām.

Tālāk tiek norādīti resursi kuru robežās strādās uzstādītais konteineris. Kubernetes konteineri izmanto specifisku veidu, kā tie norāda izmantojamo resursu daudzumu, 800m tiek lasīts kā 800 miliCPU, kas realitātē nozīmē, ka tie ir 0.8 daļas no viena CPU. Resursos netiek norādīts pilns CPU resurss mūsu gadījumā, jo uz katras Kubernetes mašīnas ir citi sīki resursi, kas patērē CPU resursus, kā rezultātā, priekš testēšanas vajadzībām nevarētu uzstādīt tik konteineru, cik kodolu, jo Kubernetes limitu menedžments neļauj pieprasīt vairāk resursu kā ir pieejams. Pēdējais nozīmīgais mainīgais "args" ir argumentu saraksts, kur tieši var norādīt stress-ng pakotnes parametrus. Šajā gadījumā tiek norādīts tikai CPU resursu daudzums, jo pārējo menedžēs Python skripts.

```
def create_deployment_object(self, cpu_count):
    # Configureate Pod template container
    container = client.V1Container(
        name=self.pod_name,
        image="alexeiled/stress-ng",
        ports=[client.V1ContainerPort(container_port=80)],
        resources=client.V1ResourceRequirements(
            requests={"cpu": "800m", "memory": "200Mi"},
            limits={"cpu": "900m", "memory": "500Mi"},
        ),
        args=["--cpu", str(cpu_count)],
    )
```

3.7. att. ***Funkcija izvietošanas šablona uzveidošanai.***

4. Pēc šablona izveidošanas tas tiek izvietots uz Kubernetes kopas ar "create_deployment" funkciju. Kad tas notiek, tiek tikai izvietots šablons un nekas cits. Tas nozīmē ka Kubernetes zina kāda sistēma tai būs jāmērogo, kad Python skripts dos atbilstošas komandas.
5. Kad tiek izpildīts 4 punkts viss ir sagatavots darbam un resursu menedžmentam. Tālāk skripts ieiet mūžīgā ciklā, kurš menedžē resursus Kubernetes kopā. Tas strādā tā, ka no sākuma pārbauda pieejamos resursus uz resursdatora. Tālāk skatās vai netiek izmantoti par daudz CPU resursi. Ja izmantotie CPU resursi ir zem 70%, tad tālāk tiek domāts par jaunu Kubernetes konteineru izveidi. Tas ir tādēļ, lai uz sistēmas arī nebūtu pārmērīgi liela slodze, kas varētu ietekmēt sistēmas fizikālo mūžu. Pa priekšu no visām ieslēgtajām mašīnām iegūst kopīgo bildi par CPU gaidīšanas laiku. Tas notiek saskaitot visu virtuālo mašīnu gaidīšanas laiku kopā. Ja šis laiks procentuāli sasniedz 5%, tas nozīmē ka neefektīvi tiek izmantoti resursi. Jeb procesi pavada 5%

laika gaidot, kad tiem tiks piešķirti CPU resursi, jo tajā laikā kāds cits process tos izmanto. Šo laiku iegūst veicot pieprasījumu katrai resursdatora virtuālajai mašīnai. Zemāk (skat. 3.8. att.) ir redzama funkcija kura ir atbildīga par šo datu iegūšanu. Vienīgā vieta, kurai ir vērts pievērts uzmanību ir tas, kā tā tiek rēķināta. Pieprasījums atgriež sarakstu, kurā pēc specificēta laika intervāla ir vērtības ar vidējo gaidīšanas laiku pa 20 sekundēm. Piemēram, pieprasot vienas minūtes gatavības rādītāju, tas atgriezīs sarakstu ar 3, 20 sekunžu vidējām vērtībām. Lai šos datus tālā interpretētu tiek pielietota funkcija:

(CPU gatavības vērtība / (datu atjaunošanas laiks * 1000)) * 100 / VM CPU skaits = CPU gatavība %

```python
# Gets cpu ready data on VM
def get_vm_data(self, interval):
    query = self.build_query("cpu.ready.summation", interval)
    self.max_cpu_ready = (float(max(query[0].value[0].value))) / 20000 * 100 / self.cpu_core_count
    self.cpu_ready = (float(sum(query[0].value[0].value)) / (interval * 3)) / 20000 * 100 / self.cpu_core_count
    LOGGER.info(f"VM NAME: {self.vm.name}")
    LOGGER.info(f'[VM] CPU Ready. Average {self.cpu_ready:.2f} %, Maximum {self.max_cpu_ready:.2f} %')
```

3.8. att. ***Funkcija kas iegūst CPU gatavības rādītāju.***

6. Kad tiek apkopota kopējā situācija par resursdatora CPU gatavības radītāju, tiek pieņemts lēmums, vai drīkst uzstādīt datu analīzes procesu no šablona vai nē. Ja šis gatavības rādītājs ir zem 5% un nav tuvu tam rādītājam, tas ir, nav virs 4%, tad tiek uzstādīts viens konteiners kurš sāk simulēt slodzi. Ja slodze pārsniedz 5% un ja uz Kubernetes mašīnas ir vismaz viens slodzes simulēšanas konteiners, tad kāds no šiem konteineriem tiek izslēgts un noņemts no mašīnas. Šajā brīdī ir svarīgi pieminēt pieņēmumu, ka ja stresa simulēšanas vietā, tur tiktu veikta datu apstrāde, šiem procesiem ir jābūt neatkarīgiem, jo tie nedrīkst ietekmēt datu apstrādi tādā veidā, ka ja šis process tiek novākts, visi dati tiek pazaudēti, bet gan jebkurš cits konteiners, kad ir pieejami resursi, var turpināt darbu. Pēc šī lēmuma pieņemšanas tiek apturēta skripta darbība uz 40 sekundēm. Pēc pauzes, darbs tiek atsākts punktā 5 un no jauna iets visam cauri.

4. REZULTĀTI

Līdzīgi kā bāzes darbā, tika noteikts bāzes operāciju skaits uz testa mašīnām, kas simulēja 30% slodzi, lai pēc tam varam noteikt, cik lielu ietekmi atstāj papildus procesu darbināšana. Zemāk 4.1 tabulā ir redzams, kāds bija operāciju skaits, ko veica katra virtuālā mašīna simulācijas laikā. Tests tika veikts divreiz, lai pārliecinātos par datu precizitāti. Slodzes grafiks redzams 7. pielikumā. CPU gatavības rādītājs bāzes testos bija vidēji 0.18%.

4.1 Tabula

Pirmā scenārija bāzes rezultāti

Virtuālās mašīnas nosaukums	Darbību skaits sekundē (ops) Pirmais tests	Darbību skaits sekundē (ops) Otrais tests	Vidējais (ops)
Test1	43904	44224	44064
Test2	43968	44187	44077
Test3	44026	44077	44051
Test4	44057	44288	44072

4.1 Pirmā scenārija rezultāti

Kā tika aprakstīts problēmas scenārijā, tiek veikta slodzes testēšana, liekot klāt Kubernetes mašīnu risinājumu. Pirmais tests tiek veiks, kad Kubernetes strādnieka mašīnai tiek piešķirti 4 kodoli.

4.2 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 4 kodoliem.

Virtuālā mašīna	Darbību skaits sekundē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU slodze vidējais (%)
Test1	38147	-13	0.77%	46.66%
Test2	38080	-13		
Test3	38424	-13		
Test4	38348	-13		

4.3 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 6 kodoliem.

Virtuālā mašīna	Darbību skaits sekundē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU vidējā slodze (%)
Test1	35704	-19	0.91%	47.58%
Test2	36026	-18		
Test3	35904	-19		
Test4	35699	-19		

4.4 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 8 kodoliem

Virtuālā mašīna	Darbību skaits sekundē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU vidējā slodze (%)
Test1	39872	-9	1.29%	40.83%
Test2	40379	-8		
Test3	40284	-9		
Test4	40429	-8		

CPU gatavības vidējais rādītājs visos testos bija stipri zem 5% un nebija tuvu. Tas ir tāpēc, ka aprēķinātais rādītājs Python skriptā aplūkoja gatavību katrai virtuālajai mašīnai atsevišķi. Piemēram, ja virtuālā mašīna A uzrāda ka CPU gatavība ir 10%, un tai ir 2 kodoli no 8, tad realitātē visa resursdatora gatavība nav 10% bet tuvāk 2%. Bet pēc ieviestās loģikas risinājums reaģē, ja uz vienas mašīnas ierauga 10%, rīkojoties atbilstoši tā, lai mazinātu šo procentu šai mašīnai.

Apkopojot datus par 4, 6 un 8 kodolu Kubernetes ar Python skripta menedžmenta rezultātiem, var novērot to, ka rezultāti ops ziņā ir līdzīgi starp visiem testiem kur tika iesaistīts Kubernetes risinājums. Var novērot ka katrā konfigurācijā ietekme uz test mašīnām ir dažādam kas ir vidēji -13% testā ar 4 kodoliem, -19% ar 6 kodolu un -9% ar 8 kodolu. Šeit var novērot ka pārejot no 4 kodolu uz 6 kodolu zaudējam visvairāk performances testa mašīnās, bet tajā pašā laikā iegūstam lielāko CPU noslodzi, kaut gan tikai par 1% labāku nekā 4 kodolu konfigurācijā. Tas nozīmē, ka visdrīzāk šī papildus CPU noslodze netika pilnvērtīgi izmantota datu analīzes mašīnās, bet gan izmantota lai menedžētu resursus starp eksistējošajām mašīnām. Jo 4 kodolu konfigurācijā ir 3:2 kodolu attiecība (VM kodolu skaits : resursdatoram pieejamo kodolu skaits), bet 6 kodolu konfigurācijā ir 7:4 ~ 3.5:2. Tā rezultātā ir redzams, ka kaut gan CPU slodze ir lielāka, tiek iegūts sliktāks rādītājs test mašīnās, jo resursi ir jāizdala pa vairāk virtuālajiem kodoliem. Tas nozīmē ka efektīvākā konfigurācija šajā gadījumā ir 4 kodolu konfigurācija. 8 kodolu konfigurācijā toties ir cits skats. Tur performances ziņā tiek zaudēti tikai 8% salīdzinot ar 4 kodolu konfigurāciju, kur tiek zaudēti 13%. Bet CPU noslodze arī ir mazāka kopumā – 40.8%. Tas tāpēc ka izstrādātais risinājums šajā konfigurācijā bieži sastapās ar 5% CPU gatavības rādītāju, kas neļāva tam likt daudz papildus slodzes, kas nozīmē, ka konfigurācijā, kad tiek ļauts analīzes uzdevumiem izplesties pa visiem brīvi pieejamajiem resursiem, rodas lielas neefektivitātes un šī konfigurācija nav optimāla.

Apskatoties šo testu slodzes grafikus (skat. 8. 9. 10. pielikumus) var novērot, ka katra testa sākumā slodze ir vienmērīga, un pēc kādas minūtes vai divām sāk strauji manīties uz leju un augšu. Tas notika kad tika pacelti vairāki Kubernetes stresa konteineri, un pašās Kubernetēs

sāka parādīties neefektivitātes resursu dalīšanā. Šo straujo izmaiņu varēja arī novērot, kad tika testēta tikai viena pati Kubernetes mašīna izstrādātājā risinājumā (skat. 11. pielikumu). Tā ir viena risinājuma neefektivitāte, ko vajadzētu apskatīt sīkāk šī risinājuma optimizācijā.

4.2 Otrā scenārija rezultāti

Otrā scenārija tiek atkārtoti noteikts bāzes darbību skaits, ko testa mašīnas izpilda simulējot 30% slodzi, jo tiek samazināts šo mašīnu skaits no 4 uz 2. Kā arī katra no šīm virtuālajām mašīnām simulē 60% slodzi, lai kopsummā resursdatora slodze būtu 30%.

1.5 Tabula

Otrā scenārija bāzes rezultāti

Virtuālā mašīna	Darbību skaits sekundē (ops) Pirmais tests	Darbību skaits sekundē (ops) Otrais tests	Vidējais (ops)
Test1	89600	89024	89312
Test2	89481	89494	89487

4.6 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 4 kodoliem.

Virtuālā mašīna	Darbību skaits sekundē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU vidējā slodze (%)
Test1	77411	-13%	0.65%	47%
Test2	77401	-14%		

4.7 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 6 kodoliem.

Virtuālā mašīna	Darbību skaits sekundē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU vidējā slodze (%)
Test1	79696	-11%	0.98%	47.95%
Test2	79478	-11%		

4.8 Tabula

Rezultāti, izmantojot Kubernetes risinājumu ar 8 kodoliem.

Virtuālā mašīna	Darbību skaits sekunē (ops)	Starpība starp bāzes testu (%)	CPU gatavības vidējais (%)	CPU vidējā vidējais (%)
Test1	77402	-13%	1.78%	41%
Test2	77353	-14%		

Otrajā scenārijā, pirmajā testā vidējā starpība ir -13% uz 4 kodolu Kubernetes konfigurāciju, otrajā testā -11% ar 6 kodolu Kubernetes mašīnu un trešajā testā arī -13% uz 8 kodolu konfigurācijas. Pēc rezultātiem var novērot, ka daudz maz optimāli varam izmantot 47%

CPU resursu, gan pirmajā scenārijā, gan otrajā, kaut gan zaudējam 11% efektivitātes galvenajās mašīnās, šis procents ir salīdzinoši zems un lielākajā daļu gadījumu neievērojams, ja netiek veikti laika sensitīvu procesu veikšana. Var arī novērot, ka starp 6 un 8 kodolu konfigurācijām palielinās vidējais CPU gatavības rādītājs, bet krītas vidējā CPU slodze uz resursdatora, kas nozīmē ka uz šī resursdatora šajā konfigurācijā palielinoties CPU gaidīšanas laikam virs 1% jau ir jūtami resursu zaudējumi, no izmantoto resursu daudzuma viedokļa. Bet kopumā var redzēt, ka tāpat kā pirmajā scenārijā, šāds risinājums neizmanto visu potenciālu, kad tiek piešķirti vairāk CPU kodoli, kas gan nenozīmē ka tas ir slikti. Jo mūsu mērķis nav izmantot pilnīgi visus CPU resursus, bet gan izmantot to efektīvi, nepalielinot CPU gatavības rādītāju, kā arī smagi neietekmējot citu virtuālo mašīnu darbu.

Kopumā var novērot, ka izstrādātais risinājums strādā nodrošinot daļu resursu datu analīzes veikšanai, smagi neietekmējot citas, uz resursdatora atrodošās sistēmas, kad tiek izmantotas konfigurācijas, kuras nepiešķir vairāk kā 75% resursdatora CPU kodolu jaudas.

5. SECINĀJUMI

Mākoņpakalpojumi katru dienu kļūst populārāki un vairāk izmantoti, kas nozīmē, ka ar vien vairāk resursi tiek uzturēti ar mākoņpakalpojumu palīdzību. Bet kaut gan pieprasījumam augot, gadi ir rādījuši, ka vidēji resursu noslogojums ir 25-30%, kas nozīmē, ka liela daļa resursu nemaz netiek izmantoti.

Turpinot par pamatu ņemto kursa darbu "Brīvu mākoņresursu pieejamība datu analīzei", tika papildus apgūta konteinerizācija kā arī Kubernetes, kuras kopā ar Python skriptu tika apvienotas vienā risinājumā, izveidojot IaaS un PaaS slāņu risinājumu. Šis skripts kontrolē datu analīžu procesu sākšanu, ja ir brīvi resursi, un to beigšanu, ja citām virtuālajām mašīnām bija vajadzīgi šie resursi. Risinājums tika balstīts uz to, ka brīvu mākoņresursu pieejamais daudzums ir aptuveni 70% un tie ir potenciāli izmantojami datu analīzei, un tiek fokusēts uz privātām mākoņu infrastruktūrām, kuru sistēmu pamatā ir VMware risinājumi. Veidojot šo risinājumu, tika nonākts pie secinājuma, ka jāveido divi testēšanas scenāriji. Pirmais scenārijs bija tāds, kur tika izveidotas 4 virtuālas mašīnas bāzes 30% slodzes veidošanai un 2 Kubernetes virtuālās mašīnas, pārbaudot kā izstrādātais risinājums dala resursus, kad tam ir pieejams dažāds brīvo resursu daudzums un galvenās virtuālās sistēmas saņem CPU kodolus 1:1 attiecībā. Pēc tam ar otra scenārija palīdzību, tika pārbaudīts, kas notiek, kad bāzes slodzes veidošanai ir 2 virtuālās mašīnas un to saņemto CPU kodolu attiecība ir 0.5:1, un kā risinājums darbojās ar dažādu daudzumu brīvo resursu.

Pēc eksperimenta rezultātiem var secināt, ka izstrādātais risinājums labi limitē CPU gatavības rādītāju. Bāzes darba eksperimentā šis rādītājs sasniedza 70%, kas ir ļoti augsts rādītās un translējot 20 sekunžu intervālos, tas nozīmē ka process gaidīja pēc resursiem 14 sekundes katrā 20 sekunžu intervālā. Šajā darbā izstrādātais risinājums tieši risināja šo problēmu un samazināja CPU gatavības rādītāju līdz 1.2% uz visu resursdatoru. Tas nozīmē, ka izveidotais risinājums neļauj neregulēti uzsākt datu analīzes procesus iegūstot gandrīz 60% efektivitāti. Šāds rādītājs tika novērots gan scenārijā, kur jau izdalīto kodolu attiecība ir vismaz 1:1, kā arī sistēmā, kur izdalīto kodolu attiecība ir 0.5:1.

Bet stingri limitējot un regulējot, kā tiek izmantoti brīvie resursi tika veiksmīgi izmantoti 17% no 70% atlikušo brīvo resursu. Šis skaitlis ir mazāks kā bāzes darba 30%, bet ņemot vērā iegūto uzlabojumu CPU gatavības rādītāja samazināšanā, 17% arī ir labs ieguvums.

Kopumā var teikt, ka sistēmās, kur noslodze ir 30% uz visiem resursdatora CPU kodoliem, ar risinājuma palīdzību ir iespējams izmantot papildus 17% no CPU jaudas datu analīzes uzdevumiem, neradot lielas neefektivitātes šo resursu dalīšanā, bet nevar šos rezultātus

iegūt neatstājot aptuveni 12% efektivitātes zaudējumus uz visām citām resursdatora virtuālajām mašīnām ar izstrādāto risinājumu.

Lai uzlabotu risinājumu ir redzamas divas vietas, kur tas varētu būt iespējams. Pirmā vieta ir uzlabot Python skripta algoritma procesu mērogošanu, lai labāk novērtētu vai tiešām var uzstādīt jaunu procesu, ja sistēma tuvojas CPU gatavības 5% robežai. Otrs ir izvērtēt visas pieejamās Kubernetes iespējas un tās CPU menedžmenta algoritmu, lai noskaidrotu, kāpēc rodas neefektivitātes sasniedzot noteiktas vietas (skat. 11. pielikumu) un atrisinātu tās.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Stephen R. Smoot and Nam K. Tan. Private Cloud Computing: Consolidation, Virtualization, and Service-Oriented Infrastructure [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/detail.action?docID=788011&pq-origsite=primo
2. Dac-Nhuong Le , Raghvendra Kumar, Gia Nhu Nguyen, and Jyotir Moy Chatterjee. Cloud Computing and Virtualization [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/reader.action?docID=5320954
3. Kas ir virtualizācija [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://opensource.com/resources/virtualization
4. Red Hat materiāli par virtualizāciju [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.redhat.com/en/topics/virtualization
5. Kas ir virtualizācija no Red Hat [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.redhat.com/en/topics/virtualization/what-is-virtualization
6. VMware virtuālās mašīnas [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.vmware.com/topics/glossary/content/virtual-machine
7. ScienceDirect apraksts par hipervizoriem [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.sciencedirect.com/topics/computer-science/hypervisors
8. TechTarget Otrā tipa hipervizora definīcija [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://searchservervirtualization.techtarget.com/definition/hosted-hypervisor-Type-2-hypervisor
9. San Murugesan and Irena Bojanova. Encyclopedia of Cloud Computing [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/detail.action?pq-origsite=primo&docID=4526670#
10. Michael J. Kavis. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS) [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/detail.action?docID=1605593&pq-origsite=primo
11. Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

12. Daniel Krisch and Judith Hurwitz. Cloud Computing for dummies [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/reader.action?docID=6260952
13. Raksts par Docker [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html
14. Gabriel N. Schenker. Learn Docker – Fundamentals of Docker 18. x: Everything You Need to Know about Containerizing Your Applications and Running Them in Production [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://ebookcentral.proquest.com/lib/lulv/detail.action?docID=5371682&pq-origsite=primo
15. Marcus Carvalho, Walfredo Cirne, Francisco Brasilerio, John Wilkes. Long-term SLOs for reclaimed cloud computing resources [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://dl.acm.org/doi/10.1145/2670979.2670999
16. Jean-Emile Dartois. Leveraging Cloud unused heterogenous resources for applications with SLA guarantees [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://hal.inria.fr/tel-03009816/document
17. Jean-Emile Dartois. Using Quantile regression for reclaiming unused Cloud Resources while achieving SLA [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://hal.inria.fr/hal-01898438/file/Using_Quantile_Regression_for_Reclaiming_Unused_Cloud_Resources_with_SLA_Guarantees.pdf
18. Cimermanis Ralfs. (2021) Kursa darbs "Brīvu mākoņresursu pieejamība datu analīzei", Latvijas Universitāte, Datorikas fakultāte, 33 lpp.
19. Ansible dokumentācija [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://www.ansible.com/overview/how-ansible-works
20. Kubernetes dokumentācija [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://kubernetes.io/docs/home/
21. Repozitorijs VMware Python API [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://github.com/vmware/pyvmomi
22. Repozitorijs Kubernetes Python API [tiešsaistē]. – [atsauce 27.05.2021]. Pieejams: https://github.com/kubernetes-client/python

PIELIKUMS

1.Pielikums

Python skripta main.py datne

```
import vSphereConnection
import kubernetesConnection
import os
import sys
import json
import logging
import logging.config
import time

# setup logging
# create log folder if does not exist
if not os.path.exists("./log"):
    os.mkdir("./log")
logger = logging.getLogger(__name__)
log_config_file = 'logging.json'
try:
    with open(log_config_file, 'rt') as f:
        config = json.load(f)
    logging.config.dictConfig(config)
except Exception as e:
    logging.basicConfig(level=logging.DEBUG)
    logging.error('Loading logging configuration file failed.', exc_info=e)

logger.info("************************************************************")
logger.info(f"Python: {sys.version}")
logger.info("Application started")

# vSphere host connection variables
vSphere_host = "10.90.0.240"
vSphere_user = "Administrator@home.local"
vSphere_pwd = 'PASSWORD'

# Sets data interval in minutes for CPU usage request
interval_time = 1

# Define all kubernetes variables
kubernetes_host = "http://localhost:8080"
kubernetes_api_key = "API KEY"
kubernetes_deployment_name = "stress-test-deployment"
kubernetes_pod_name = "stress-test"
kubernetes_worker_node_names = ["worker01"]
kubernetes_vm_names = ["Kubernetes_wroker_1", "Kubernetes"]
kubernetes_worker_names = ["Kubernetes_wroker_1"]
kubernetes_pod_stress_cpu_count = 1
kubernetes_pod_replicas_count = 0

time_to_sleep = 40

# Create connection to kubernetes cluster
kubernetes = kubernetesConnection.KubernetesConnection()
kubernetes.set_configuration(kubernetes_api_key, kubernetes_host)
kubernetes.connect()

# Create connection to vSphere client
vSphere_connection = vSphereConnection.VSphereConnection(vSphere_host,
vSphere_user, vSphere_pwd)
```

```
vSphere_connection.get_host()
vSphere_connection.get_vms(kubernetes_vm_names)
vSphere_connection.get_kubernetes_workers(kubernetes_worker_names)

# Get host cpu stats
vSphere_connection.get_host_stats()

# max allowed readiness as percentage
max_allowed_readiness = 5
# sets high unreachable start
last_known_limit = 1000
limit_time_stamp = time.time()

# Create kubernetes deployment object
kubernetes.set_deployment_object(kubernetes_deployment_name,
kubernetes_pod_name, kubernetes_pod_stress_cpu_count)

# Try to delete deployment on kubernetes cluster
try:
    kubernetes.delete_deployment()
    # time.sleep(20)
except:
    pass

# Create deployment on kubernetes cluster
kubernetes.create_deployment()

# time.sleep(20)
start_time = time.time()
# Start kubernetes resource management
try:
    while True:
        for kubernetes_worker in vSphere_connection.kubernetes_workers:
            # Resets limit so new pods can be generated
            if time.time() - limit_time_stamp > 60:
                last_known_limit += 1

            # Update host cpu stats
            vSphere_connection.update_host_stats()
            readiness = 0

            # Check if there is no high CPU usage
            if vSphere_connection.stats["cpu_usage"] < 70:
                # Counts readiness time for all kubernetes VMs
                for kubernetes_vm in vSphere_connection.kubernetes_vms:
                    kubernetes_vm.get_query_params()
                    kubernetes_vm.get_vm_data(interval_time)
                    readiness += kubernetes_vm.cpu_ready

                # Counts readiness time for all non-Kubernetes VMs
                for default_vm in vSphere_connection.default_vms:
                    default_vm.get_query_params()
                    default_vm.get_vm_data(interval_time)
                    readiness += default_vm.cpu_ready

                logger.info(f"Total readiness: {readiness:.2f}")

                # Checks if readiness is below maximum allowed
                if readiness < max_allowed_readiness:
                    if max_allowed_readiness - readiness > 1:
                        # Maximum stress pod count is same as core count,
as single pod has limit of one core.
```

```
                        if kubernetes_pod_replicas_count <
kubernetes_worker.cpu_core_count and \
                                kubernetes_pod_replicas_count <
last_known_limit - 1:
                            kubernetes_pod_replicas_count += 1

kubernetes.update_scale_deployment(kubernetes_pod_replicas_count)
                            logger.info(f"Scaled up kubernetes pods.
Current pod count: {kubernetes_pod_replicas_count}")
                            # Wait for system to normalize
                            time.sleep(50)
                        else:
                            logger.info("Maximum available pods deployed")
                else:
                    # Check if last know pod limit is higher than pod count
                    if last_known_limit > kubernetes_pod_replicas_count:
                        last_known_limit = kubernetes_pod_replicas_count
                        limit_time_stamp = time.time()

                    # If there are any pods left, then deploy one less
                    if kubernetes_pod_replicas_count != 0:
                        kubernetes_pod_replicas_count -= 1

kubernetes.update_scale_deployment(kubernetes_pod_replicas_count)
                        logger.info(f"Scaled down kubernetes pods, pods
remaining: {kubernetes_pod_replicas_count}")
                        # Wait for system to normalize
                        time.sleep(50)
                    else:
                        logger.warning("No kubernetes pods are running, but
system has high readiness level")

        logger.info(f"START SLEEP {time_to_sleep} SEC")
        time.sleep(time_to_sleep)
except:
    pass

# Close connection to vSphere client
vSphere_connection.disconnect()
# Kills kubernetes deployment
kubernetes.delete_deployment()
```

2.Pielikums

Python skripta vSphereConnection.py datne

```
from pyVim.connect import SmartConnectNoSSL, Disconnect
import vm

class VSphereConnection():
    def __init__(self, host, user, pwd):
        self.host_ip = host
        self.user = user
        self.pwd = pwd
        self.host = None
        self.connection = None
        self.kubernetes_vms = []
        self.default_vms = []
        self.kubernetes_workers = []
        self.stats = {}
```

```
        self.connect()

    # Connects to vSphere Client
    def connect(self):
        self.connection = SmartConnectNoSSL(host=self.host_ip,
user=self.user, pwd=self.pwd)

    # Disconnects form vSphere client
    def disconnect(self):
        Disconnect(self.connection)

    # Gets host machine
    def get_host(self):
        self.host =
self.connection.content.rootFolder.childEntity[0].hostFolder.childEntity[0]
.host[0]

    # Gets all online VM and splits them into Kubernetes and Non-kubernetes
VMs
    def get_vms(self, kubernetes_vm_names):
        for vm_instance in self.host.vm:
            if vm_instance.runtime.powerState == "poweredOn":
                is_kubernetes_vm = False
                for name in kubernetes_vm_names:
                    if vm_instance.name == name:
                        self.kubernetes_vms.append(vm.VM(vm_instance,
self.connection))
                        is_kubernetes_vm = True
                        break
                if not is_kubernetes_vm:
                    self.default_vms.append(vm.VM(vm_instance,
self.connection))

    # Gets all kubernetes worker nodes in one place
    def get_kubernetes_workers(self, kubernetes_worker_names):
        for vm_instance in self.kubernetes_vms:
            for name in kubernetes_worker_names:
                if vm_instance.vm.name == name:
                    self.kubernetes_workers.append(vm_instance)
                    break

    # Gets host cpu stats
    def get_host_stats(self):
        self.stats = {
            "overall_cpu_usage":
self.host.summary.quickStats.overallCpuUsage,
            "max_capacity": (self.host.hardware.cpuInfo.hz *
self.host.hardware.cpuInfo.numCpuCores) / 1000 / 1000,
        }
        self.stats["cpu_usage"] = self.stats["overall_cpu_usage"] /
self.stats["max_capacity"] * 100
        self.stats["free_cpu_resources"] = 100 - self.stats["cpu_usage"]

    # Gets only cpu usage stats
    def update_host_stats(self):
        self.stats["cpu_usage"] = self.stats["overall_cpu_usage"] /
self.stats["max_capacity"] * 100
        self.stats["free_cpu_resources"] = 100 - self.stats["cpu_usage"]
```

Python skripta kubernetesConnection.py datne

```
import kubernetes.client
import kubernetesDeployment
import logging

LOGGER = logging.getLogger(__name__)

class KubernetesConnection():
    def __init__(self):
        self.configuration = kubernetes.client.Configuration()
        self.api_client = None
        self.apps_v1_api = None
        self.deployment_object = None
        self.core_v1_api = None

    def set_configuration(self, api_key, host):
        self.configuration.api_key['authorization'] = api_key
        self.configuration.host = host

    def connect(self):
        self.api_client = kubernetes.client.ApiClient(self.configuration)
        self.apps_v1_api = kubernetes.client.AppsV1Api(self.api_client)
        self.core_v1_api = kubernetes.client.CoreV1Api(self.api_client)

    def set_deployment_object(self, deployment_name, pod_name, cpu_count):
        self.deployment_object =
kubernetesDeployment.KubernetesDeployment(deployment_name, pod_name)
        self.deployment_object.create_deployment_object(cpu_count)

    # Create deployment
    def create_deployment(self):
        resp = self.apps_v1_api.create_namespaced_deployment(
            body=self.deployment_object.deployment, namespace="default"
        )

        LOGGER.info("[INFO] deployment `stress-test-deployment` created.")
        LOGGER.info(f"NAMESPACE, NAME, REVISION, IMAGE")
        LOGGER.info(f"{resp.metadata.namespace} {resp.metadata.name}
{resp.metadata.generation} {resp.spec.template.spec.containers[0].image}")

    # Update scale for deployment (Deployed pod count)
    def update_scale_deployment(self, replica_count):
        # Update container image
        self.deployment_object.deployment.spec.replicas =
int(replica_count)

        # patch the deployment
        resp = self.apps_v1_api.patch_namespaced_deployment(
            name=self.deployment_object.deployment_name,
namespace="default", body=self.deployment_object.deployment
        )

        LOGGER.info(f"deployment {self.deployment_object.deployment_name}
updated.")
        LOGGER.info(f"NAMESPACE, NAME, REVISION, IMAGE")
        LOGGER.info(f"{resp.metadata.namespace} {resp.metadata.name}
{resp.metadata.generation} {resp.spec.template.spec.containers[0].image}")

    # Delete deployment
```

```
    def delete_deployment(self):
        # Delete deployment
        resp = self.apps_v1_api.delete_namespaced_deployment(
            name=self.deployment_object.deployment_name,
            namespace="default",
            body=kubernetes.client.V1DeleteOptions(
                propagation_policy="Foreground", grace_period_seconds=5
            ),
        )
        LOGGER.info(f" deployment {self.deployment_object.deployment_name}
deleted.")
```

4.pielikums

Python skripta kubernetesDeployment.py datne

```
from kubernetes import client

class KubernetesDeployment():
    def __init__(self, deployment_name, pod_name):
        self.deployment_name = deployment_name
        self.pod_name = pod_name
        self.deployment = None

    def create_deployment_object(self, cpu_count):
        # Configureate Pod template container
        container = client.V1Container(
            name=self.pod_name,
            image="alexeiled/stress-ng",
            ports=[client.V1ContainerPort(container_port=80)],
            resources=client.V1ResourceRequirements(
                requests={"cpu": "800m", "memory": "200Mi"},
                limits={"cpu": "800m", "memory": "200Mi"},
            ),
            args=["--cpu", str(cpu_count)],
        )

        # Create and configure a spec section
        template = client.V1PodTemplateSpec(
            metadata=client.V1ObjectMeta(labels={"app": self.pod_name}),
            spec=client.V1PodSpec(containers=[container]),
        )

        # Create the specification of deployment
        spec = client.V1DeploymentSpec(
            replicas=0, template=template, selector={
                "matchLabels":
                    {"app": self.pod_name}})

        # Instantiate the deployment object
        self.deployment = client.V1Deployment(
            api_version="apps/v1",
            kind="Deployment",
            metadata=client.V1ObjectMeta(name=self.deployment_name),
            spec=spec,
        )
```

Python skripta vm.py datne

```
from pyVmomi import vmodl, vim
from datetime import timedelta
import logging

LOGGER = logging.getLogger(__name__)

class VM():
    def __init__(self, vm_data, connection):
        self.vm = vm_data
        self.connection = connection
        self.content = None
        self.vchtime = None
        self.perf_list = None
        self.perf_dict = {}
        self.max_cpu_ready = None
        self.cpu_ready = None
        self.cpu_core_count = vm_data.config.hardware.numCPU

    # Get all parametrs from VM for Query
    def get_query_params(self):
        self.content = self.connection.RetrieveContent()
        self.vchtime = self.connection.CurrentTime()
        self.perf_list = self.content.perfManager.perfCounter
        for counter in self.perf_list:
            counter_full = "{}.{}.{}".format(counter.groupInfo.key,
counter.nameInfo.key, counter.rollupType)
            self.perf_dict[counter_full] = counter.key

    # Check if passed counter is found in vm dictionary
    def stat_check(self, counter_name):
        counter_key = self.perf_dict[counter_name]
        return counter_key

    # Build query and get response
    def build_query(self, counter_id, interval):
        counter = self.stat_check(counter_id)
        perf_manager = self.content.perfManager
        metric_id = vim.PerformanceManager.MetricId(counterId=counter,
instance="")
        start_time = self.vchtime - timedelta(minutes=interval + 0.2)
        end_time = self.vchtime - timedelta(minutes=1)
        query = vim.PerformanceManager.QuerySpec(intervalId=20,
entity=self.vm, metricId=[metric_id],
                                                 startTime=start_time)
        perf_results = perf_manager.QueryPerf(querySpec=[query])
        if perf_results:
            return perf_results
        else:
            LOGGER.info('ERROR: Performance results empty.  TIP: Check time
drift on source and vCenter server')
            LOGGER.info('Troubleshooting info:')
            LOGGER.info('vCenter/host date and time:
{}'.format(self.vchtime))
            LOGGER.info('Start perf counter time   :
{}'.format(start_time))
            LOGGER.info('End perf counter time     :  {}'.format(end_time))
            LOGGER.info(query)
            exit()
```

```
    # Gets cpu ready data on VM
    def get_vm_data(self, interval):
        query = self.build_query("cpu.ready.summation", interval)
        self.max_cpu_ready = (float(max(query[0].value[0].value))) / 20000
* 100 / self.cpu_core_count
        self.cpu_ready = (float(sum(query[0].value[0].value)) / (interval *
3)) / 20000 * 100 / self.cpu_core_count
        LOGGER.info(f"VM NAME: {self.vm.name}")
        LOGGER.info(f'[VM] CPU Ready. Average {self.cpu_ready:.2f} %,
Maximum {self.max_cpu_ready:.2f} %')
```

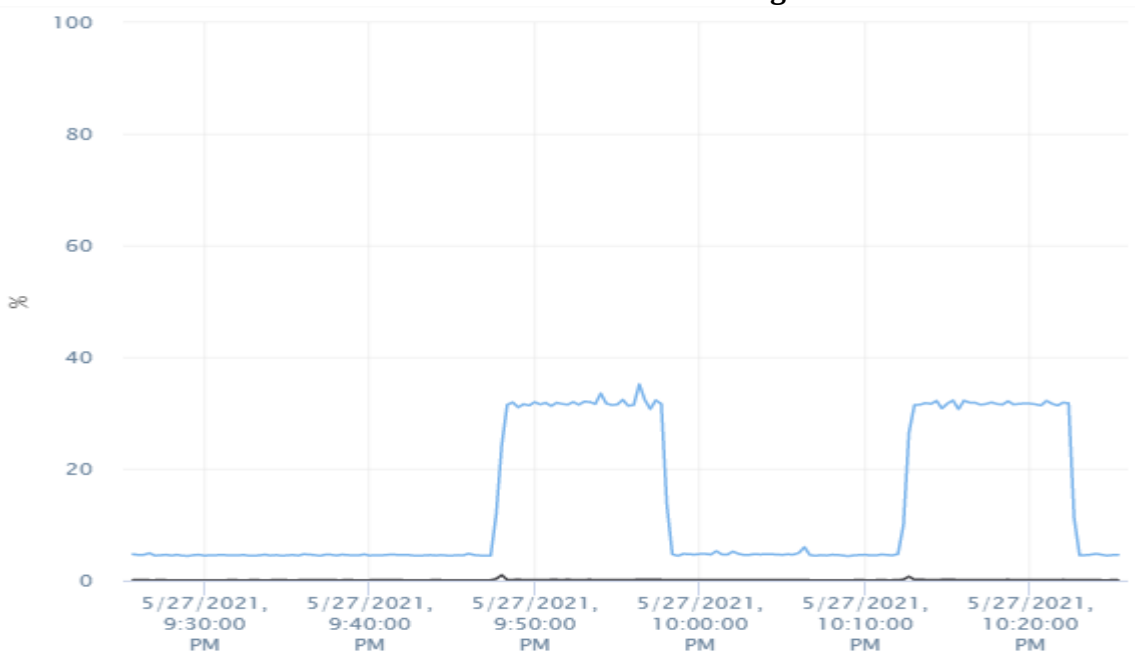
6.pielikums

Bāzes rezultāti no bāzes darba "Brīvu mākoņresursu pieejamība datu analīzei"[18]

Mašīna	Pirmais tests (ops)	Otrais tests (ops)	Vidēji (ops)
Test1	32138	31872	32005
Test2	32064	31808	31936
Test3	32064	32097	32080
Test4	32128	32124	32126

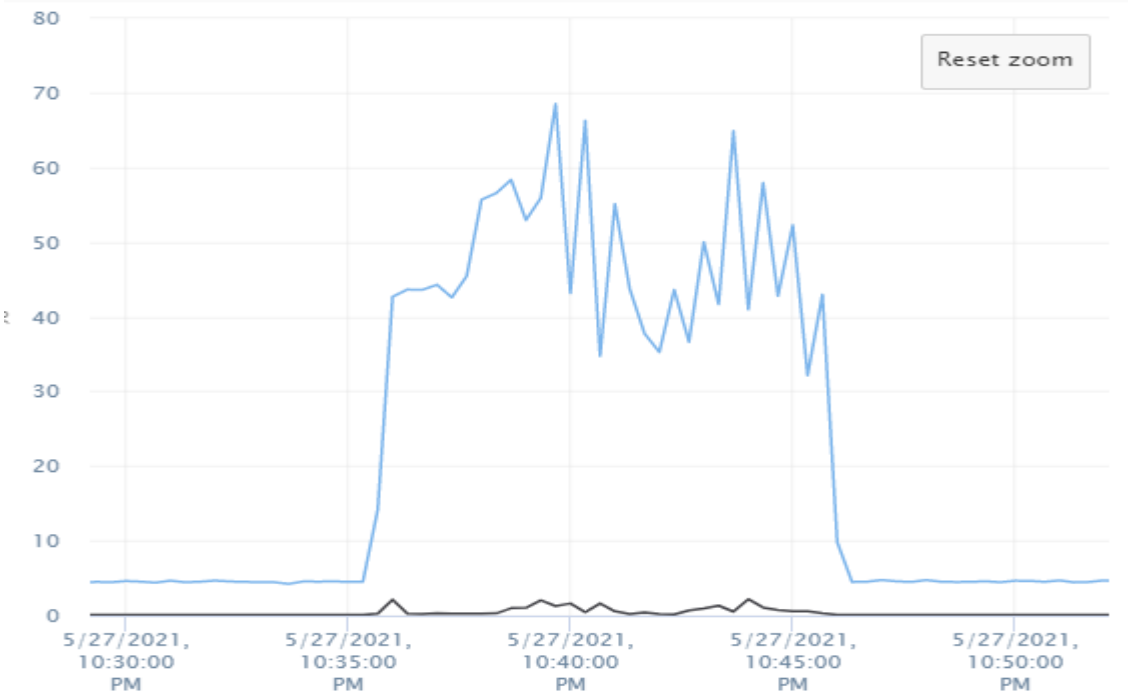
7.pielikums

Bāzes testa resursdatora slodzes grafiks



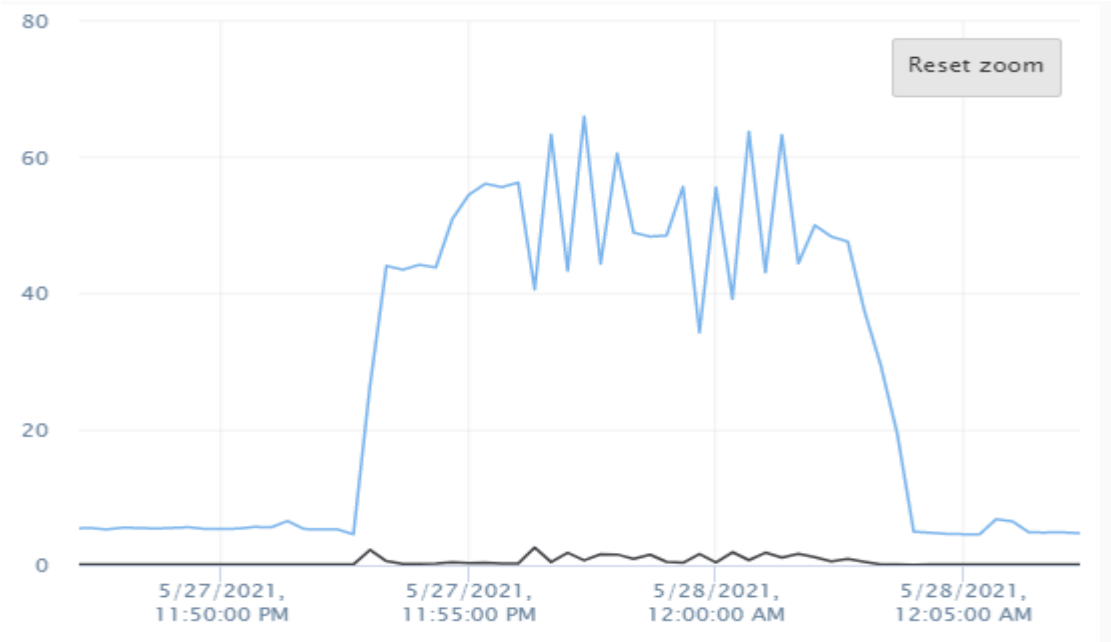
8.pielikums

Pirmā testa, pirmā scenārija resursdatora slodzes grafiks (Kubernetes ar 4 CPU)



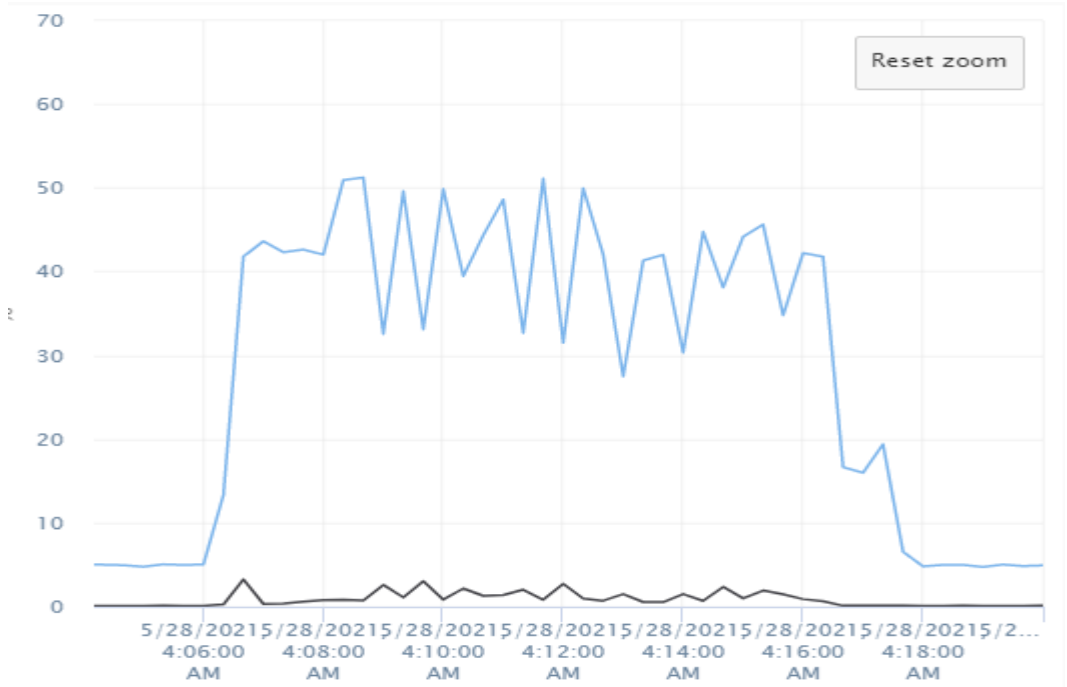
9.pielikums

Otrā testa, pirmā scenārija resursdatora slodzes grafiks (Kubernetes ar 6 CPU)



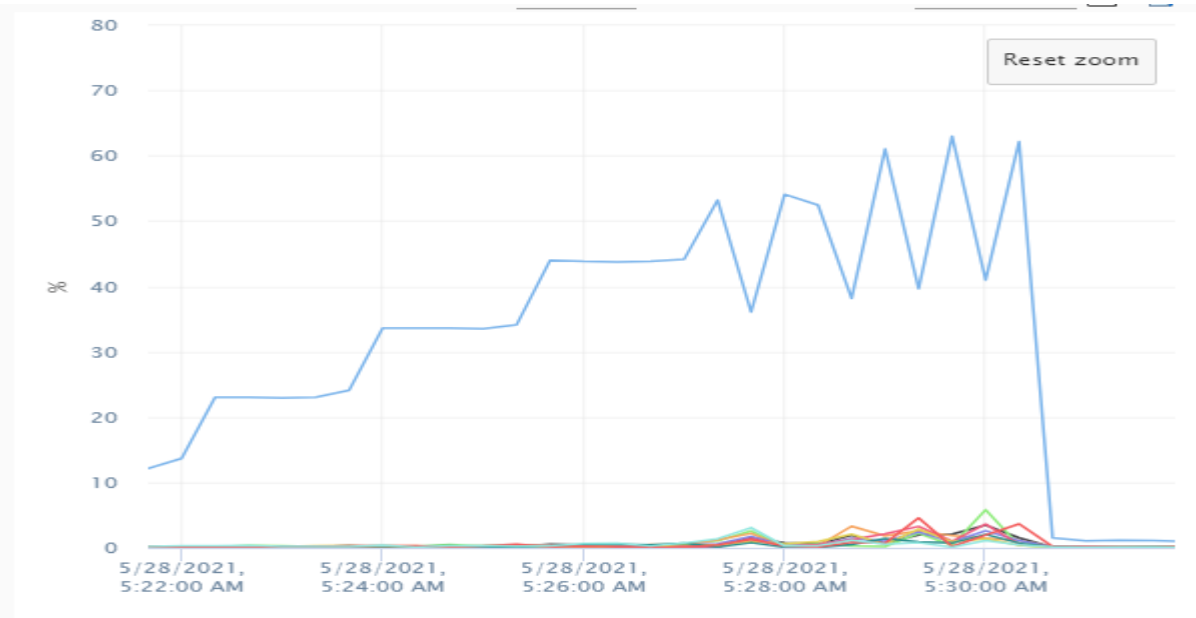
10.pielikums

Trešā testa, pirmā scenārija resursdatora slodzes grafiks (Kubernetes ar 8 kodoliem)



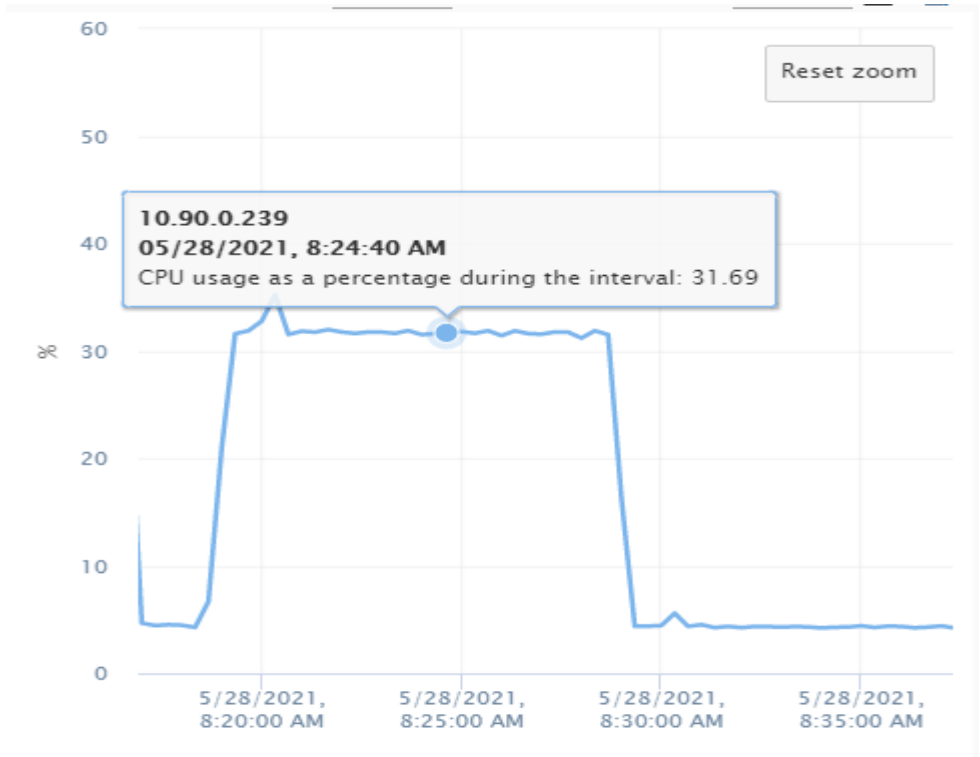
11.pielikums

Python/Kubernetes izveidotā skripta darbība bez 30% bāzes slodzes



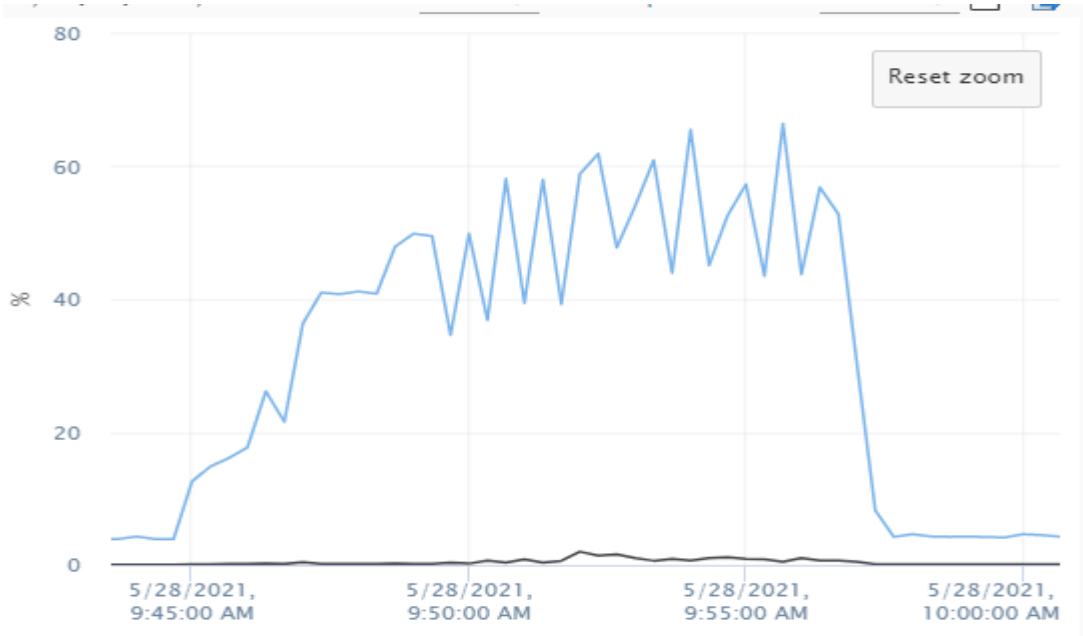
12.pielikums

Otrais scenārijs bāzes rezultātu noteikšanas slodze



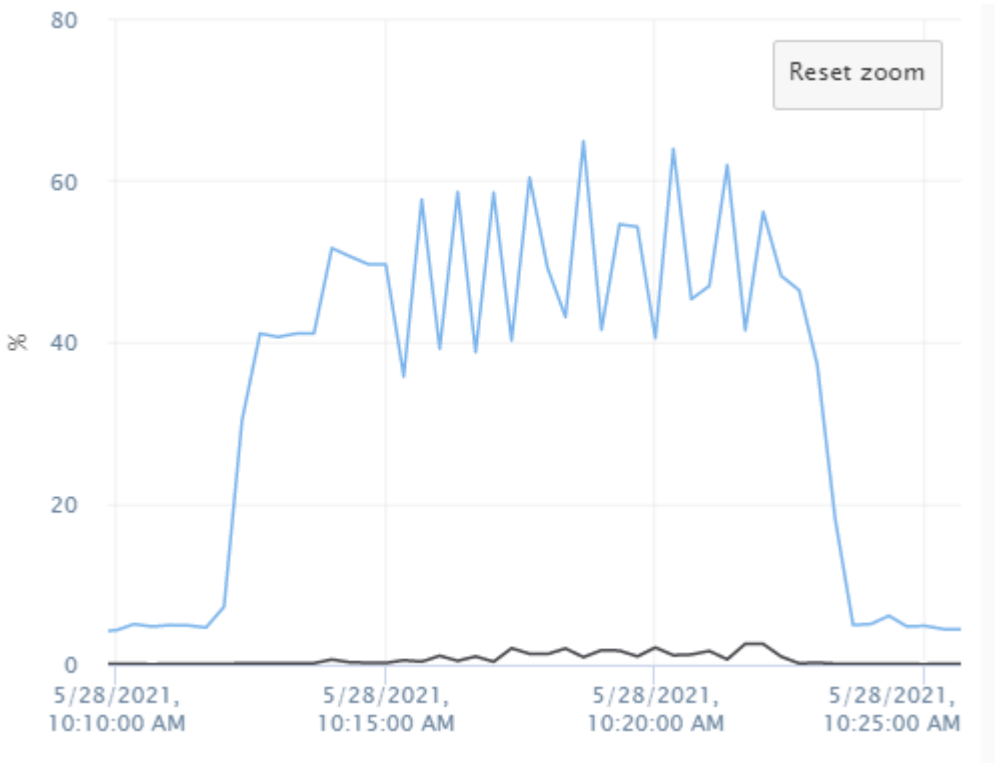
13.pielikums

Otrā scenārija, pirmā testa resursdatora slodzes grafiks (kubernetes 4 kodoli)



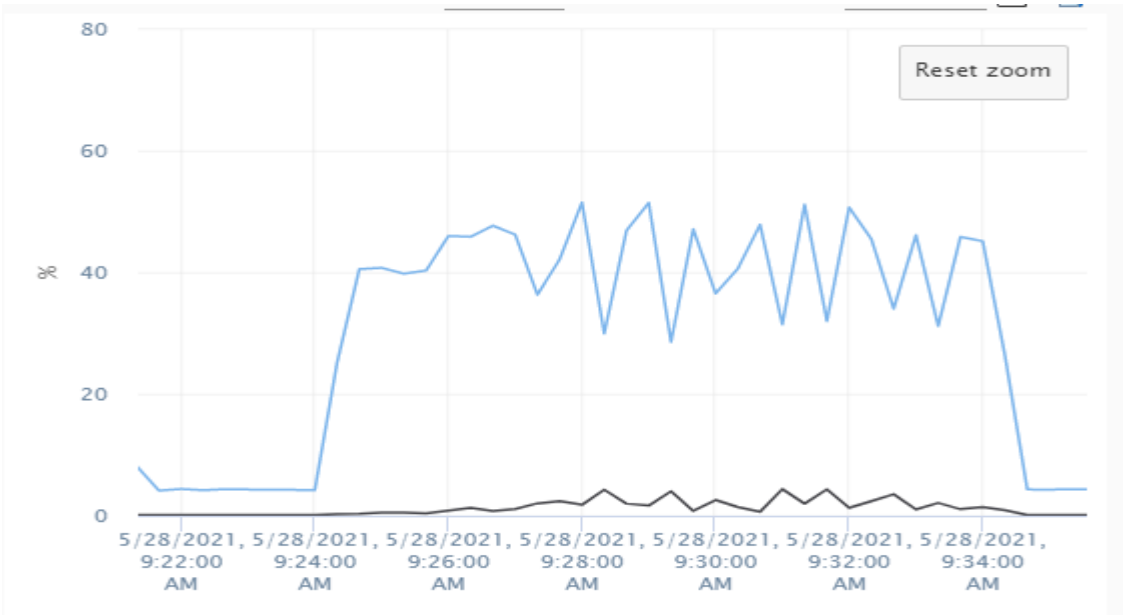
14.pielikums

Otrā scenārija, otrā testa resursdatora slodzes grafiks (kubernetes 6 kodoli)



15.pielikums

Otrā scenārija, trešā testa resursdatora slodzes grafiks (kubernetes 8 kodoli)



Otrā scenārija, pirmā testa Python skripta log faila dotās informācijas paraugs (Kubernetes 4 kodoli)

```
- Scaled up kubernetes pods. Current pod count: 4
- START SLEEP 40 SEC
- VM NAME: Kubernetes_wroker_1
- [VM] CPU Ready. Average 1.89 %, Maximum 2.14 %
- VM NAME: Kubernetes
- [VM] CPU Ready. Average 2.64 %, Maximum 3.17 %
- VM NAME: Test2
- [VM] CPU Ready. Average 0.18 %, Maximum 0.24 %
- VM NAME: Test1
- [VM] CPU Ready. Average 0.22 %, Maximum 0.27 %
- Total readiness: 4.92
- START SLEEP 40 SEC
```

Bakalaura darbs „Brīvu mākoņresursu pieejamība datu analīzei” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti.

Autors: Ralfs Cimermanis 31.05.2021.

Rekomendēju darbu aizstāvēšanai

Vadītājs: profesors Dr.sc.comp. Jānis Zuters 31.05.2021.

Recenzents: docents Anastasija Ņikiforova

Darbs iesniegts Datorikas fakultātē 31.05.2021.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

__.06.2021. prot. Nr. ___.

Komisijas sekretārs: docents Dr.sc.comp. Aivars Niedrītis